Light software services for dynamical partial reconfiguration in FPGAs

Yan Xu

▶ To cite this version:

Yan Xu. Light software services for dynamical partial reconfiguration in FPGAs. Micro and nanotechnologies/Microelectronics. Université de Grenoble, 2014. English. NNT : 2014GRENT010 . tel-01060171

HAL Id: tel-01060171
https://theses.hal.science/tel-01060171

Submitted on 3 Sep 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Nano-Electronique et Nano-Technologies**

Arrêté ministériel : 6 Août 2006

Présentée par

Yan XU

Thèse dirigée par **Frédéric Pétrot**

préparée au sein **Laboratoire TIMA**
et de **l'École Doctrale EEATS**

Gestion Logicielle Légère pour la Reconfiguration Dynamique Partielle sur les FPGAs

Thèse soutenue publiquement le **13 Mars 2014**,
devant le jury composé de :

M. Bertrand GRANADO
Professeur à l'UPMC, Président

M. Loïc LAGADEC
Professeur à l'ENSTA Bretagne, Rapporteur

M. Fabrice MULLER
Maître de Conférence à Polytech'Nice, Rapporteur

M. Fabrice LEMONNIER
Directeur du Lab HPC chez Thales Research and Technology, Examinateur

M. Benoit MIRAMOND
Maître de Conférences à l'ETIS, Examinateur

M. Frédéric PÉTROT
Professeur à Grenoble INP, Directeur de thèse

M. Olivier MULLER
Maître de Conférences à Grenoble INP, Co-Directeur de thèse

Acknowledgement

This work is carried out during the years 2010-2014 at TIMA Laboratory, SLS group. It is an honour for me to finally have this chance to thank those who made this thesis possible.

My deepest gratitude goes to my advisor Mr. Olivier MULLER, who made available his support in a number of ways. Just after my arrival, Olivier provided me with a series of good quality tutorials which helped me quickly integrate into the group and set up a good working manner. During the four years, I was always appreciating the constructive and interesting discussions with him. He constantly encouraged me to develop my own ideas and to have more self-confidence. When I lost myself into certain details, his strong sense of priorities and capacity of structuring ideas prevented me many times from sticking into the mud of trivialities. I highly valued his practical writing advices for my manuscript, and his useful communication tips for my defence presentation. Especially, I am grateful for his patience and efforts for aiding me in coming out of my shell of shyness. In addition, Olivier spared his precious time to help me with all the administrative procedures, which allows me to concentrate to the research work.

I would like to express my warmest acknowledgement to Professor Frédéric PÉTROT for directing my thesis. Although we saw each other only occasionally, he guaranteed his presences at every key moment along the years when I was pursuing my doctorate. Thanks to his extensive expertise, I could obtain useful references when I urgently needed to apprehend or integrate some concepts. His rigorous scholarship and optimistic attitude always made me go out of the meeting room with a better-organized mind and a lighter heart.

Mr. Fabrice MULLER, Mr. Loïc LAGADEC, Mr. Benoit MIRAMOND, Mr. Bertrand GRANADO are thanked sincerely for accepting being the members of my jury, and in particular the two reviewers for their constructive remarks.

I owe many thanks to Xavier GUÉRIN, Damien HEDDE and Pierre-Henri HORREIN for their generous technical support; to Nicolas FOURNEL and Frédéric ROUSSEAU for their insightful suggestions; and to all members of the SLS group for their frank feedbacks.

The debt I owe is not only intellectual but also moral. Mr. Gang FENG, thank you for years of concern, from the recruitment at the very beginning to my last defence. Mrs. Sophie MARTINEAU, I cherish your lovely French courses in Monday afternoons, full of laughters. Hai YU, your warm reception released me a lot from the tension of being in a foreign country. Adrien PROST-BOUCLE, Florentine DUBOIS, Greicy COSTA-MARQUES, Maryam BAHMANI and Zuheng MING, your friendship is irreplaceable. I will always remember the good time we had together.

My special thank goes to Quentin MEUNIER and his family. Quentin, you are a patient listener and always turning my head to the bright side. Each step of my advance was accompanied by your persistent encouragement. The continuous tender care from your family gives me feeling of belonging in France. I really appreciated for all you people have done.

Last but not less important, I wish to send my ocean-deep feeling to my family and friends in China, where my life energy resides. Every Chinese word coming from you, through a letter or a phone call, was calming and evoking the nostalgic wave of mine. Thank you for your companion and support in distance. Mother and Father, you said nothing but gave me wings when I wished to fly. I just want you to know that, no matter how far am I, you are my roots forever.

Contents

1 Introduction 1

2 Problem Statement 5
- 2.1 Task Management 6
- 2.2 Resource Management 7
- 2.3 Communication Management 8
- 2.4 Flexibility Issues 8
- 2.5 Summary 9

3 State of the Art 11
- 3.1 Background on Model of Computation and Reconfigurable Circuit . 11
  - 3.1.1 Computational Model 11
    - 3.1.1.1 Streaming Model 12
    - 3.1.1.2 Kahn Process Network Model 12
    - 3.1.1.3 Synchronous Data-Flow Model 12
    - 3.1.1.4 Multi-Threaded Model 13
  - 3.1.2 Reconfigurable Architectures 14
    - 3.1.2.1 Granularity of FPGA Devices 14
    - 3.1.2.2 Reconfiguration Types 14
    - 3.1.2.3 Coupling FPGAs with GPPs 15
  - 3.1.3 Gap Between Computational Models and Reconfigurable Architectures 15
- 3.2 Integration Strategies 17
  - 3.2.1 Integration at Task Level 17
    - 3.2.1.1 FOSFOR 17
    - 3.2.1.2 ReConfigME 18
    - 3.2.1.3 SPORE 18
    - 3.2.1.4 Flextiles 19
  - 3.2.2 Integration at Process Level 19
    - 3.2.2.1 BORPH 19
  - 3.2.3 Integration at Thread Level 20
    - 3.2.3.1 HybridThread 20
    - 3.2.3.2 ReconOS 20
    - 3.2.3.3 FUSE 21
    - 3.2.3.4 SPREAD 22
    - 3.2.3.5 Virtual Memory System 22

3.2.4 Integration at Instruction-Set Processor Level 23
3.2.4.1 Chimaera 23
3.2.4.2 XiRisc 23
3.2.4.3 MOLEN 24
3.2.4.4 RISPP 24
3.3 Conclusion 25

4 An Abstraction Layer for Dynamic Reconfiguration 27
4.1 Hypotheses on the System 27
4.1.1 Targeted Hardware Architecture Template 27
4.1.1.1 Cells and the Homogeneous Choice 28
4.1.1.2 Reconfiguration Controllers 28
4.1.1.3 General-Purpose Processors 29
4.1.2 Software Environment Assumptions 29
4.1.2.1 Explicit Partition 29
4.1.2.2 Cooperative Multitasking 29
4.1.2.3 Pre-synthesized User Library 30
4.2 A Motivating Example 30
4.2.1 Development and Maintenance of the Code 31
4.2.2 Flexibility of the Code 31
4.2.3 Working in the Multi-User Context 32
4.2.4 Motivation of our Work 32
4.3 Proposal: An Abstraction Layer Wrapping Hardware Components . 32
4.3.1 Observed Interactions 32
4.3.2 Hardware Component: the Key Element for Interfacing with Application Layer 33
4.3.2.1 Hardware Component Properties 33
4.3.2.2 Hardware Component Operations 35
4.3.3 Hardware Component Manager: A Centralized Reconfigurable Hardware Resource Manager 37
4.3.3.1 Conceptual Model of HCM 37
4.3.3.2 An Implementation of the HCM 39
4.4 Summary of the Chapter 41

5 A Scalable Communication Mechanism for Dynamic Reconfiguration Platforms 43
5.1 Communication Problems Brought by Dynamically Reconfigurable Platforms 43
5.1.1 A Motivating Example 44
5.1.2 Analysis of the Motivating Example 46
5.1.2.1 The Existence of Tasks 46
5.1.2.2 The Access to Tasks 47
5.2 MWMR Channel Analysis in a Dynamic Reconfigurable Context . . 48
5.2.1 MWMR Channel Description 48
5.2.2 Why the MWMR Channel is Chosen 50
5.2.2.1 Shared Hypotheses 50
5.2.2.2 Useful Features 50

5.2.2.3 Reasonable Technical Requirements 52
5.2.3 The Problems Unsolved by the MWMR Channel 52
5.3 Proposed Communication Mechanism Based on MWMR Channels and HCM 54
5.3.1 Proposed Architecture 54
5.3.2 Communication Services 56
5.3.3 Use on the Motivating Example 58
5.4 summary of this chapter 59

6 Experiments **61**
6.1 Proof-of-concept Integration in an OS 61
6.1.1 The Implementation of the HCM Integration 62
6.1.2 HWC Services Description 64
6.1.3 Experiment 1: Feature Validation on Simulator-based Test Environment 65
6.1.3.1 Test Platforms 65
6.1.3.2 Application 66
6.1.3.3 Results Analysis 67
6.1.4 Experiment 2: Integration Cost 69
6.1.4.1 Test Platform and Application 70
6.1.4.2 Execution Result 70
6.1.4.3 Results Analysis 72
6.2 Communication Mechanism Validation 76
6.2.1 Experiment 3: Original MWMR Channel Migration 76
6.2.2 Experiment 4: Dynamicity Management with the HCM and Modified MWMR Channels 78
6.3 Conclusion 82

7 Conclusion **83**
7.1 Contribution 83

List of Tables

4.1 Hardware Component Properties 34
4.2 Hardware Component Services 36

6.1 Parameters of test platforms 66
6.2 Basic components for test applications 66
6.3 Time Overhead of OS Extension Services 73
6.4 Memory Footprints of Systems 74

List of Figures

1.1 Evolution of Internet and cellphone users between 1980 and 2010 . . 1
1.2 Evolution of the amount of data produced every year between 2005 and 2020 2

2.1 Schematic view of a basic dynamically reconfigurable system 5
2.2 Task Graph Example 6

3.1 The Coupling between Reconfigurable Fabrics and General Purpose Processors (adapt from [HD07]) 16
3.2 Gap between Computation Models and Reconfigurable Architectures 16

4.1 The targeted hardware architecture template 27
4.2 Task Assignment in a GPP/FPGA Hybrid Platform 31
4.3 The Rewritten Motivated Example 36
4.4 Hardware Component Manager in System 37
4.5 The State Machine of a Cell 38
4.6 Internal Structure of the HCM Implementation 40

5.1 Passive Task Model 44
5.2 Communication Problems Caused by Dynamic Reconfigurable Platform 45
5.3 Basic Properties of a MWMR Channel 49
5.4 Different Physical Task Graphs Using the Same MWMR Channels . . 51
5.5 The Problems that MWMR Channels Left Unsolved 53
5.6 Proposed Communication Architecture 55

6.1 Global View of Software Organization 62
6.2 One Implementation of HCM-integrated OS 63
6.3 Simple image processing scenario 67
6.4 Code without HCM 68
6.5 Code with HCM 69
6.6 The Underneath Processing of Scenario in Different Platforms 69
6.7 Time Consumption During the Lifetime of an HA 71
6.8 Maximum Frequency that the HCM can Reach Depending on the Cell Number 75
6.9 HCM Resource Utilization Depending on the Cell Number 75
6.10 Platform for Testing the Original MWMR channel 77

6.11 MWMR Channel Validation in Various Data Producer-Consumer Pair Cases 79
6.12 The Platform for Testing the Dynamic Communication Mechanism 80
6.13 The Platform for Testing the Dynamic Communication Mechanism 80
6.14 Underlying Cell Usage for Each Application Processing the Two First Images 81

List of Used Acronyms

General Acronyms

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DPR	Dynamic Partial Reconfiguration
DR	Dynamic Reconfiguration
FIFO(s)	First-in First-out queue(s)
FPGA(s)	Field Programmable Gate Array(s)
GPP	General-Purpose Processor
HPC	High Performance Computers
HLS	High-Level Synthesis
HW/SW	Hardware/Software
MPI	Massage Passing Interface
MWMR	Multi-Writer Multi-Reader
OS	Operating System
P-C pair	Producer-Consumer pair
SDF	Synchronous Data Flow

User-Defined Acronyms

ARD	Allocation Request Dispatcher
CTM	Cell Track Maintainer
DRM	Dynamic Resource Manager
HCM	Hardware Component Manager
RF(s)	Reconfigurable Fabric(s)
RHR	Reconfigurable Hardware Resource

Chapter 1

Introduction

THE digital revolution, which started somewhere around 1970 and is still undergoing today, has modified deeply and in many ways the society we are living in. The number of users of electronic devices has not stopped increasing since then. As an example, figure 1.1 shows that the number of cellphone users has gone from 11.2 millions in 1980 to 4 billions in 2010, while the number of Internet users has passed from less than 100,000 in 1980 to more than 1.8 billion in 2010.

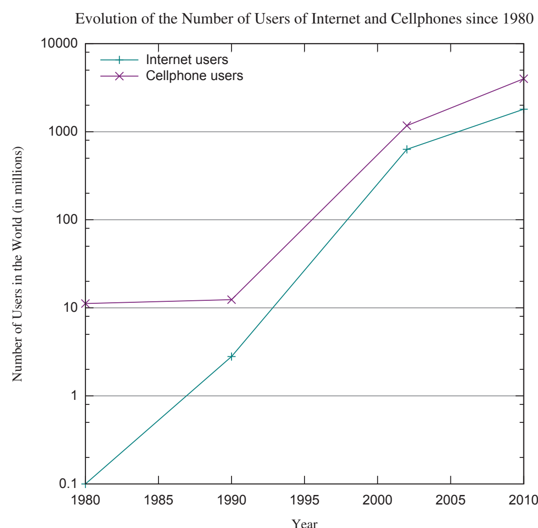


Figure 1.1: Evolution of Internet and cellphone users between 1980 and 2010

Similarly, the number of electronic devices has known an incredible growth over the last 30 years, going from 1 million in 1980 to more than 6 billions in 2012. As another example, Oracle is proud to say that more than 4 billion devices run java in the world at every java update.

Not only the digital revolution is related to the omnipresence of electronic devices, but it has also been accompanied by a fast evolution of each individual system's performance. The most famous quoted measure is the number of transistors integrated on a single chip, which has continuously doubled every 2 years since 1965, commonly known as Moore's law.

One of the challenges addressed by this integration growth is the face to the always increasing computation demand. Although this demand has a lot of sources (e.g. games), one of the biggest is data processing. In fact, the amount of data produced worldwide in 2010 has reached 1.2 zettabytes (10^{21} bytes), and even if only about 5% of this data is structured – i.e. can be analyzed by a machine –, it still represents a very high computing demand [idc11]. According to the same sources, this trend is not going to stop in the near future, since the International Data Corporation foresees the world to produce more than 40 zettabytes per year in 2020, as shown on figure 1.2.

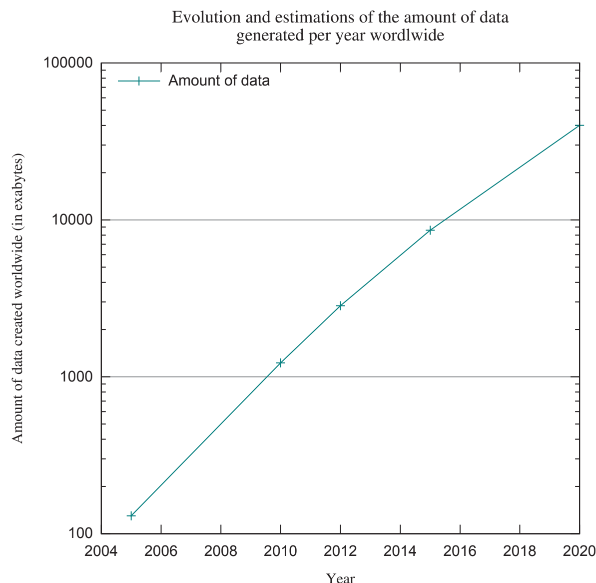


Figure 1.2: Evolution of the amount of data produced every year between 2005 and 2020

However, if Moore's law combined to the increase of the number of devices provides a support to face this demand, it still does not resolve it all as two major problems come up with the transistor integration growth.

The first one may seem paradoxical since it deals with the utilization of the transistors themselves. Around 2004, 50 years of exponential improvement in the sequential performance of processors ended [OH05]. This led manufacturers to design multicore chips, which has been the first answer to the utilization of available circuit surface. The second problem is related to the power consumption of actual and future chips, since it estimated that by 2015, 300mm² chip will consume more than 1kW [Bor07]. One part of this increase is due to what is called the *power wall*, or the trend to consume an exponentially increasing power when increasing linearly the frequency. The other part is related to Pollack's rule, which states that the performance increase of a circuit achieved via architectural improvements is roughly proportional to the square root of the increase of complexity – the latter being directly linked with power consumption. Once again, an answer is to turn towards multi- and manycore chips, so as to enhance the performance per watt ratio.

Finally, another problem related to Moore's law is the end of it; we know that

physical constraints will make technology below 1nm hardly feasible, and until a major breakthrough in technology occurs – likely the end of CMOS –, new ways to improve performance at fixed cost and power must be found.

Using FPGAs to Improve Performance

A general-purpose microprocessor, by executing various software on a fixed hardware, can achieve any logical or mathematical operation that a programmer can conceive, but the performance is not always satisfactory. In contrast, Application Specific Integrated Circuits (ASICs), by dedicating hardware circuits to a particular task, can result in a smaller, cheaper, faster chip that consumes less power than a programmable processor. However, a slight change in its functionality may lead to a complete redesign and rebuilt of the circuit, which is very expensive.

In order to find the right balance, the idea of adapting hardware to the application after build time was formed in the 1960s [Est60]. However, this idea could not become a reality before long.

Field Programmable Gate Arrays (FPGAs) are circuits which were originally designed in the mid-1980's to validate Register Transfer Level (RTL) models before using them to make an ASIC chip. By configuring its behaviour with the means of a "bitstream", a FPGA can achieve any functionality in hardware. These bitstreams are created from RTL models *via* different tools. Using FPGAs is thus an alternative to RTL simulation, with the advantage of providing a very high speed compared to the former. It is, as such, a RTL emulation technique.

However, this also allows triggered a regain of interest in the idea of adapting hardware to the application after its build. Quickly enough, FPGAs became used as a part of a System-on-Chip.

With this hypothesis, circuit design now possesses an additional step in the speed vs generality trade-off: a functionality implemented by a RTL model on a FPGA is orders of magnitude faster (up to 3 can be observed in literature) than a software implementation, what also results on large power savings compared to a processor executing equivalent code. On the other side, the circuit used for it has neither to be designed, nor is definitely attached to this functionality. As such, a FPGA is a good compromise between a processor executing a piece of software and an ASIC. The drawbacks are the opposite: a RTL model is still harder to write than a high level function; besides, a RTL model on a FPGA will still run one order of magnitude slower than an ASIC backend of this model.

Exploiting Reconfiguration Inside FPGAs

To even gain more in flexibility, researchers and industrials did not content themselves with integrating FPGAs in system-on-chip. In the late 1990's, a technique was proposed to allow a part of the FPGA to be reconfigured while the rest of the system is still running and computing, thus opening a new era in which the adaption of hardware to the application can even be done at runtime. This technique was called *Dynamic Partial Reconfiguration* (DPR).

Dynamic reconfigurable FPGAs achieve both inherent computing efficiency and potentially infinite execution space by breaking the barriers between hardware and software. Unfortunately, compared to CPU-based solutions, FPGA-based solution suffer from low productivity. The latter is measured by the time required to arrive to a solution and indirectly by flexibility (scalability, portability and reusability).

This thesis investigates dynamic reconfigurable FPGA, and more specifically the ways to ease the life of application developers who program a system containing such FPGAs. In particular, this work will try to improve the flexibility of such systems in terms of scalability, portability and reusability.

The rest of the thesis is organized as follows: chapter 2 introduces the problematic we considered for this work; chapter 3 presents related works considering the integration of FPGAs into conventional systems-on-chip; chapter 4 details the abstraction layer proposed by the author to solve the portability problems stated earlier; chapter 5 presents the communication mechanisms used between the different parts of the systems; chapter 6 summarizes the experiments fulfilled during this thesis to validate our proposal; finally, chapter 7 concludes and discusses possible future works.

Chapter 2

Problem Statement

THIS chapter introduces the general context of our study, centered around dynamic reconfigurable (DR) systems, i.e. systems in which a part of the hardware can be adapted to applications at run-time, while the rest of the system keeps executing as normal.

Figure 2.1 is a schematic diagram of a typical architecture employed by DR systems, in which the general purpose CPUs and reconfigurable fabrics (RF) are loosely-coupled by an interconnect.

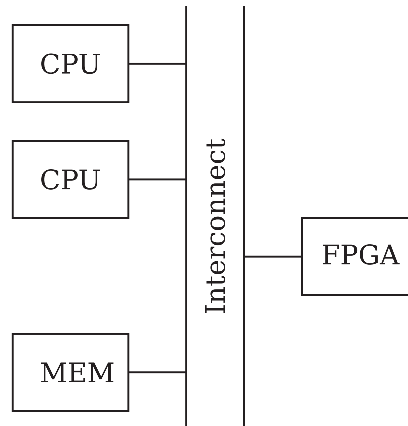


Figure 2.1: Schematic view of a basic dynamically reconfigurable system

According to the integration strategy, the RF part can either act as individual data processors in a heterogeneous multi-processor system or peripheral data processing devices.

Depending on the scale of the DR system, the RFs can be implemented by one partial reconfigurable FPGA or several FPGAs. The number of CPUs is not limited to one neither. The CPUs can be conventional micro-processors, multi-core processors, or soft cores in FPGAs.

In our hypothesis, applications can be seen as communicating tasks, as shown in Figure 2.2. A task can be a data producer (T1), a data consumer (T3) or both (T2) at the same time. Tasks can be executed by CPU or be mapped to RF. We

can see that the CPU/RF hybrid DR architecture provides a quite promising computational power by breaking the barrier between hardware (hardware) and software (software), but it is exactly such a mixture that makes programming somehow complicated. Taking the dynamic nature and flexibility demand into consideration, handling such systems is not an easy job.

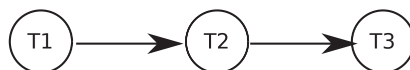


Figure 2.2: Task Graph Example

There are numerous issues in programming models, resource management, communication infrastructure to deal with, and many design choices to make before finally arriving to a solution.

2.1 Task Management

Noticing the possible parallelism implied by the loose coupling between the CPUs and RFs, and the potential multi-CPU, multi-FPGA context, wise application programmers would probably consider dividing their algorithms into multiple computation tasks, which can independently execute during a relatively long period of time and which can effectively communicate with each other when necessary. The tasks definition and their assignment to an execution unit have a great influence on the system's performance, thus should be treated with care.

One particularity of the hybrid DR systems is that a task can either be executed by a CPU, called a software task; or be implemented on a RF, then called a hardware task. A hardware task can be considered as the hardware accelerator of its software counterpart. Normally, an application designer knows best whether a task should be allocated to a kind of execution unit or another; therefore, when the whole system is totally predictable, the HW/SW partitioning can be done statically before the application is executed. However, there are cases in which the behavior of the application depends on information only available at run-time. Accordingly, the executing unit of a specific task must be decided based on system state at that moment. Dedicating the precious RF to a task which never runs is a waste of resource and an inefficient way of using the DR architecture. This on-demand task allocation increases the difficulty of task management, because the procedure of launching a hardware task is not the same as the one of a software one.

The hardware tasks can be prepared in advance in several ways:

- from a Register Transfer Level Hardware Description Language, such as VHDL or Verilog,
- by commercial Electronic Design Automation tools,
- or, from high level language, as C or C++, by High-Level Synthesis (HLS) tools.

Some just-in-time compile-synthesis methods are in research to provide an on-line hardware task generation. Whichever the way, the bitstreams of hardware tasks must be loaded to the FPGA before these hardware tasks actually run. They can run until all expected computations are finished, or be swapped in-and-out and complete all predefined operations in several time slices. Whether preemptive or not, there must be a way to recognize the tasks' state, i.e. that the requested hardware task is available on FPGA (first appearance or reloaded), and that the hardware tasks reach their end or the point to be swapped out.

2.2 Resource Management

The DR feature adds another dimension of system resource management. The FPGA is not only a spatial resource, but also temporally shared by different hardware tasks. Therefore, resource management mainly consists in managing the RFs in the platform, through a reconfiguration process, by allocating room for hardware tasks and for a given time on the RFs. Apart from this reconfiguration process, the configurations (usually called bitstreams) of hardware tasks corresponding to the RFs should also be stored somewhere in memory. The identification and transfers of these bitstreams to the RF also need to be handled. In addition, FPGAs can be configured through several kinds of interface, such as Joint Test Action Group (JTAG) port, synchronized serial data/clock interface, or Internal Configuration Access Port (ICAP) in some self-reconfigurable FPGAs. All these reconfigurable interfaces need to be controlled and thus require the system to have this ability.

From the view of resource management, the execution of a hardware task can be considered as a request of part of the RF on the FPGA. Some of the questions to address during the lifetime of a hardware task are the following:

- How to organize RFs into zones in which we can put different hardware tasks?
- How to choose a specific zone to be reconfigured for the required hardware task from the numerous available zones?
- What to do when there are no more zones available?
- How to make a zone reusable once the hardware task on it has finished its execution?

In a multi-threaded context, several applications run independently. It is neither necessary nor possible to know the executing state of other applications. There must be a mechanism to guarantee the coexistence of hardware tasks belonging to different applications. For example, the requests sent simultaneously from different applications should not be lost; or a hardware task configured on the RF by one application should not be replaced by a request of another application even without a proper run. When in a multi-FPGA context or when an FPGA contains several configuration ports, parallelizing several reconfigurations is feasible. In that case, the distribution of reconfiguration requests to the different ports is also a problem to consider.

2.3 Communication Management

As mentioned at the beginning of this chapter, our programming model is supposed to be communicating tasks, which form the data producer-consumer (P-C) pairs (Figure 2.2). In a conventional architecture, once the task graph is settled, the communication can be managed as the data synchronization between P-C pairs. However, the communication in a DR system could be much more complicated.

Firstly, each task could either be implemented in its software version or hardware version. Consequently, a basic P-C pair evolves into four combinations: software/software, software/hardware, hardware/software and hardware/hardware. In a software version, data is usually stored in arrays, which are referred to by a pointer; while in a hardware version, data is probably stored in an unaddressable First-In First-Out (FIFO) queue on the RF. The communication mechanism should be able to cover the different natures of communicators.

Secondly, depending on the availability of resources on the RF, one task could have more than one instance. As a result, a basic P-C pair may leave as single-producer/single-consumer, or it may expand to multiple-producers/single-consumer, single-producer/multiple-consumers or multiple-producers/multiple-consumers. Thus, the number of instances of a hardware task may not be predicable. The communication mechanism should be able to cover an arbitrary number of communicators in both sides of a P-C pair, so that the computational power is not wasted due to disconnections among tasks.

Finally, since the hardware tasks need a period of time to be reconfigured on the RF and since they may be taken off from the RF, the communicators of a basic P-C pair do not always exist. The communication mechanism should also be able to guarantee that the data will not be sent to or taken from nowhere, while keeping the property that different running tasks should not break their boundaries to get information of existence of other tasks.

2.4 Flexibility Issues

Usability, portability and scalability are the terms we use when talking about flexibility. In the above sections, we discussed the management of tasks, reconfigurable resources and communication separately, but in practice they are often tightly connected due to FPGA architectures. In the conventional way of using FPGAs, it is the application programmer who decides which task is put on which part of FPGA, at which moment, through which reconfiguration port, from which place the bitstream of the task should be taken from in the memory, and how the ready task should be accessed.

The strong relation between the platform and the application imposes on the application programmers a deep knowledge about the underlying hardware, what asks extra efforts apart from their own job – programming to implement algorithms. The usability of DR CPU/FPGA hybrid architecture thus need to be improved.

In addition, platform information is planted into applications, what makes such applications not portable at all. A slight change of the platform, such as a damage

of one part of the FPGA, requires a complete recoding of the application; not to mention the porting of an application to other FPGA devices.

Besides, such applications are not scalable: the communication graph is firmly based on task distribution and task distribution is mixed with resource management of the current platform. As a result, when resources are added to the platform, the applications cannot benefit from the potential increase in computational power without a rewrite, which is both not trivial and error prone.

To solve the flexibility issues, the only possible choice is to hide the platform details to application programmers. All the problem is to decide to which degree this hiding should occur. Is it better to achieve a virtualization, meaning the application programmers are not aware of the underlying platform at all; or just an abstraction, meaning the application programmers can get some information from and give order to the underlying platform in a much easier way. The former gives the application programmer more freedom, while the latter promises better performance gained from more control. Once decided, the other problems arising are the following:

- How to provide this hiding;
- how to separate the communication from the task distribution;
- how to separate the management of tasks and resource.

2.5 Summary

After discussing the different aspects of DR CPU/FPGA hybrid system, we can identify the following problems:

1. How are tasks allocated to different resources in a CPU/FPGA hybrid system where dynamic partitioning is allowed?
2. How the reconfiguration related resources (RFs, bitstreams, reconfiguration ports) are managed, so that the DR processing can be well maintained even in a multi-threaded, multi-FPGA environment?
3. How to design a communication mechanism which can recognize the existence of dynamically appearing communicators, no matter what the nature and number of communicators at both sides of a P-C pair?
4. How to ease the life of application programmers by separating the management of tasks, reconfiguration resources and communication, so that they can write more flexible applications?

Chapter 3

State of the Art

THIS chapter presents existing work relevant to our study. The different domains covered deal with the parallel programming models, and the reconfigurable architectures which have been proposed in the literature. The focus is made on the gap between both domains, and the attempt to integrate reconfiguration into existing programming models.

3.1 Background on Model of Computation and Reconfigurable Circuit

Current applications often impose conflicting requirements on computing system designers. For example, the system should be efficient to implement a specific application, while it should be generic enough in order to adapt to different applications; or the computing power is expected to greatly increase, while stricter constraints in terms of area, footprint and power consumption are placed. In order to meet such design requirements, various efforts are made in the community. In this section, we are going to review mainly in two directions: the computational model and the reconfigurable architecture.

3.1.1 Computational Model

To make developers more productive in describing their applications, various computational models have been proposed to abstract the hardware system, giving the developers the representations of algorithms and data structures which are easier to manipulate than underlying hardware details. One tendency of the computational model development is to explore the parallelism inside an application. To do this, the most discussed subjects are recognizing the independent computation parts and managing the communications amongst them. Some frequently used computational models are described in the following text. It is to note that research efforts also exist to bring interoperability between models [DCL13].

3.1.1.1 Streaming Model

Data stream is a model often used in real time data-intensive applications, such as network monitor, telecommunication data management, on-line video player, etc.

In [Mut03], the authors describe data stream as a ”sequence of digitally encoded signals used to represent information in transmission, where TCS balance should be struck when input data comes at a very high rate”. Here, T stands for transmitting the entire input to the program, C for computing sophisticated functions on large pieces of the input at the rate it is presented, and S for storing, capturing temporarily or archiving all of it on the long term. The authors of [BBD+02] point out the common characters of data stream models. For instances, the input data is not available for random access from disk or memory, but rather arrive as continuous data items. The system has no control over the order or the moment of the arriving data items. Compared with the potentially unbounded size of data stream, the number of data items that can be temporarily stored is relatively small. Besides, once an item in a data stream has been processed, it cannot be retrieved easily.

The parallelism in streaming application can be explored by separating the application in several step functions running at the same time, and keeping the output data rate of one step function the same as the input data rate of next step functions.

3.1.1.2 Kahn Process Network Model

The Kahn Process Network (KPN) is a widely accepted model for parallel programs, used for example in [APDG05] and [CDHL12]. The formal description of KPN model is firstly given in [Kah74] as a programming language. If we try to describe it in simple words, the KPN model could be seen as concurrently running processes which communicate through unidirectional FIFOs. A process in the KPN model either stays blocked by the unavailable input, or it computes and products output data. The process transmits information within an unpredictable but finite amount of time. The data writing in an output FIFO is non-blocking, which means in theory that the FIFO should acts as if it had an infinite size.

The above characteristics indicate that the KPN is suitable for modeling only the deterministic parallel programs. Here the word ”deterministic” can be explained as the following: whatever the possible execution order of running processes is, the final outputs and the data history on channels of one program remain unchanged.

For applications satisfying the KPN model hypotheses, their concurrency and communications are explicitly handled. KPN Processes are only data-dependent, none of control variables are shared. In practice, the FIFO size need to be bounded by considering the data rate and possible executing time range. Some run-time monitoring mechanism might be need to notify and handle the overflow exceptions [Par95].

3.1.1.3 Synchronous Data-Flow Model

The Synchronous Data-Flow (SDF) model [LM87] is considered to be a natural paradigm for describing many digital signal processing applications. In the SDF

model, algorithms are described as directed graphs where the nodes represent computations and the arcs represent data paths. Any node can fire (perform its computation) whenever input data are available on its incoming arcs. When the node fires, a fixed number of data samples (or tokens) are consumed or produced on each arc respectively.

The SDF model is sometimes considered as a restriction of the KPN model. However, thanks to its static property (the fixed number of tokens on each arc), the SDF model has certain appealing characteristics. For instance, it is suitable for optimization techniques, namely minimizing bounded buffer sizes and static scheduling at compile time [BLM96].

Based on the primary SDF model, many efforts have been made to extend this model to be able to describe more generic programs, such as the following:

- the cyclo-static data flow model [BELP95], in which the rules of firing are allowed to be changed cyclically;

- The scenario-aware data flow model [TGB+06], in which the data rate and execution time can be parameterized according to the scenario occurrence captured by a stochastic approach;

- The control-operations-integrated SDF model, where the control informations are exchanged amongst processes to synchronize their execution and configuration [Bui13].

3.1.1.4 Multi-Threaded Model

The multi-threaded programming model brings the abstraction of *threads* to the programmer, a thread being a sequence of instruction executed in order. In this model, the threads communicate through shared memory, meaning that a thread can access to any memory location without constraint. This is why the threads come along with some synchronization primitives: at least locks, and often other abstractions like semaphores and barriers. Usually, this kind of model also comes up with some other support, e.g. memory allocation primitives.

Contrary to the previous models, multi-threaded programming lets the programmer a lot more freedom, but this freedom comes with a counterpart: the programmer has to guarantee himself the correctness of the synchronization to avoid deadlocks, while allowing enough concurrence between threads to get an effective parallelization. The latter is non-trivial at all, and this is why embedded programs are usually not written directly using this model. However, this model is well adapted to write a higher-level programming model such as the ones described above.

The most commonly known multi-threaded programming interface is the POSIX specification which provides all these basic blocks. Its most widespread implementations are Linux and BSD.

3.1.2 Reconfigurable Architectures

[HD07] provides an introduction to the entire range of issues relating to reconfigurable computing. In this book, various reconfigurable architectures are presented as the solution of the conflict between the increasing performance demand and the stricter constraints of area, memory footprint and energy consumption. Nowadays, the most commonly used reconfiguration device is the FPGA. The following discussion are mainly about the granularity of the FPGA device, the configuration type and the coupling with general purpose processors (GPP).

3.1.2.1 Granularity of FPGA Devices

FPGA can be seen as an array of logic block islands surrounded by general routing resources. The complexity and size of the logic blocks are referred to as the granularity of the blocks. The range of the granularity can vary.

[KTR08] reviews a spectrum of FPGAs. The logic blocks range from the very-fine-grained ones made of transistors, NAND gates; to medium-grained ones made of multiplexers, lookup tables; and to coarse-grained ones made of PAL-style wide-input gates, or even small processors.

The area, speed and power consumption are analyzed based on the granularity of the FPGA. In general, the fine-grained FPGAs benefit from convenient bit-level manipulations, but suffer from lower productivity, larger area, slower clock rates and higher power consumptions when complicated functions are demanded. The facts that developers have to construct functions at bit-level and that many areas are dedicated for interconnection result to a lower area-efficiency and a longer configuration time. The coarse-grained FPGAs, such as DART [DCPS02], are the other side of the story, as they are suited to implement relatively complicated operations, but very fine value operations lead to unnecessary area and speed overheads. The detailed trade-off can be found in [Ahm01].

3.1.2.2 Reconfiguration Types

The logic blocks and routing resources in FPGAs are controlled by reprogrammable memory locations. Boolean values held in these memory bits control whether certain wires are connected and what functionality is implemented by a particular piece of logic. A specific sequence of 1s and 0s for a particular memory locations in hardware is called a configuration, or referred to as a bitstream in the rest of the thesis. The process of loading bitstreams to the hardware memory locations is called reconfiguration. Depending on the time and influence to the system, we can divide the reconfigurations into two kinds: static ones and dynamic ones.

Static reconfigurations refer to the reconfigurations which can only take place during the system initiation and in which any change of configuration requires a halt of the whole system. If we limit our discussion only at the FPGA scope, the static reconfiguration examples could be the single-context FPGAs. The memory locations of such FPGAs can only be sequentially accessed, so that a large single bitstream for the whole chip has to be reloaded even when the functionality of a very small part needs to be modified.

At the opposite, *dynamic reconfiguration* means that a part of the fabric can be reconfigured while the rest of the system keeps on running. The example of dynamic reconfiguration could be multi-context FPGAs which allow background loading of inactive planes when an active plane is in execution. Another example is the partial reconfigurable FPGAs, whose memory locations can be randomly accessed, so that reconfiguration of one part of the chip and the computation of the rest are allowed to overlap in time. It is a special case of dynamic reconfiguration, called dynamic partial reconfiguration (DPR).

The main difference between static and dynamic reconfiguration is whether the functionality of a system hardware can evolve at runtime. It is not strictly bound to a special reconfigurable architecture. For instance, in a multi-FPGA system composed of single-context FPGAs, if the reconfiguration of a particular FPGA does not effect the execution of other FPGAs, the multi-FPGA system is still a dynamic reconfigurable system.

Another concept related with DPR is self-reconfiguration. It refers to a reconfigurable architecture in which the partial reconfigurations are controlled at runtime inside the FPGA device itself (either by a CPU core or by a dedicated controller).

3.1.2.3 Coupling FPGAs with GPPs

Frequently, the reconfigurable fabrics (RFs) are coupled with GPPs to set up a reconfigurable system. Such systems tend to make good use of both computation structures. High density of parallel data processing are often mapped to RFs, while certain control-intensive operations are left to GPPs. According to the position of RFs in the memory hierarchy, [HD07] summed up the coupling between RFs and GPPs as shown in Figure 3.1.

Tightly coupled RFs can be seen for example in PRISC architecture [RS94], Chimaera architecture [YMHB00], and OneChip architecture [CC01]. Garp architecture [CHW00] is an example of loosely coupled RF. The coprocessor RF can be found in the RaPiD architecture [CFF+99] and various commercial systems.

Each of the above coupling styles has its advantages and drawbacks. Normally, the tighter RFs and GPPs are coupled, the lower is the communication overhead between the two computation structures. The looser RFs and GPPs are coupled, the more complicated functions can be put on the RFs, thus greater parallelism can be achieve between the two computation structures. Before choosing an appropriate coupling style, the developer needs to carefully analyze the nature (control-intensive or data-intensive) of an application.

3.1.3 Gap Between Computational Models and Reconfigurable Architectures

As shown in Figure 3.2, we have reviewed some commonly used computational models and some key characteristics of the reconfigurable architectures in the last two subsections. However, for application programmers who probably are not experts of hardware, the link between the sea of reprogrammable arrays and their familiar models for describing algorithms is not so obvious. To be able to quickly

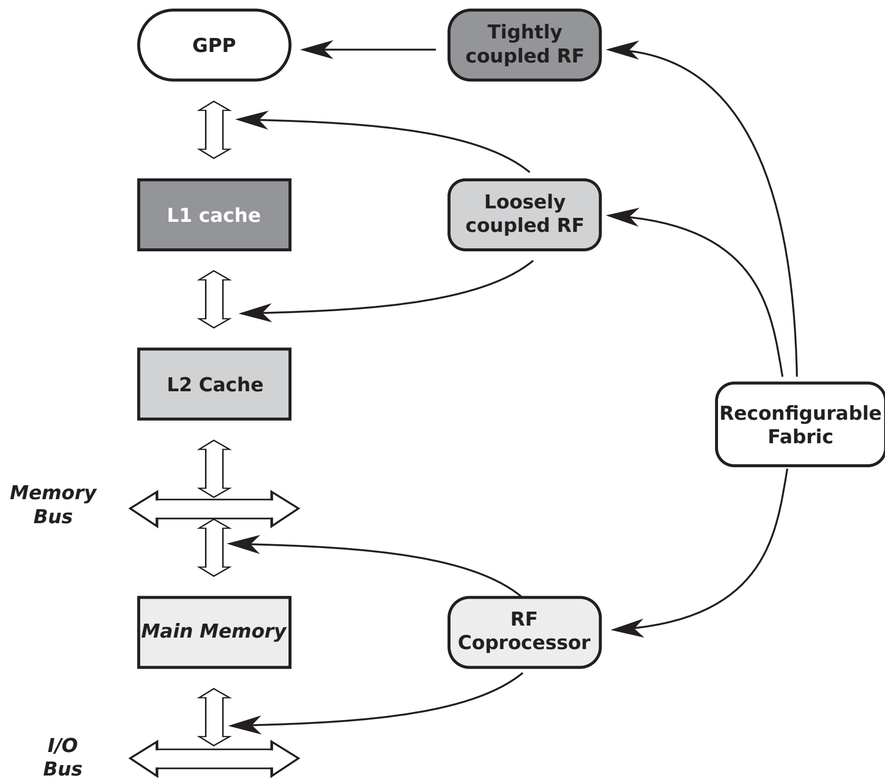


Figure 3.1: The Coupling between Reconfigurable Fabrics and General Purpose Processors (adapt from [HD07])

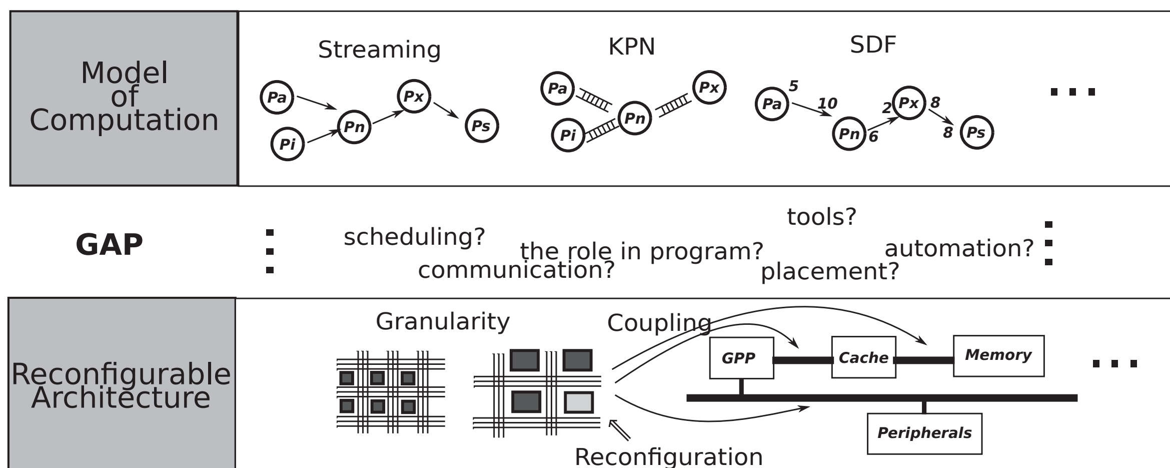


Figure 3.2: Gap between Computation Models and Reconfigurable Architectures

come to a solution, they need the appropriate design methodologies to answer the new problems introduced by reconfigurable architectures. Examples of such problems are: what is the role of RFs in an application? How do they communicate with the GPPs? Which place should a computation be mapped to? When should a computation take place? Are the programmer supposed to take care of all above

aspects? Is that possible to have some automation?

There is clearly a gap between the computation models and the reconfiguration architecture, which largely influences the productivity of the application programmer. In the following two sections, we are going to review some efforts to fill the gap. The related works are organized from two angles: how the RFs are integrated in a computing system, and how these extra reconfigurable resources are managed at runtime.

3.2 Integration Strategies

In this section, we are going to review some works bridging the semantic gap between the algorithm and the reconfigurable hardware architecture, emphasizing how hardware implemented computations are encapsulated and notified by programming environments.

The related works are categorized into four groups according to different integration strategies. We can see that the way of integration (at task level, process level, thread level or instruction-set level) also influences the way of interactions between a hardware implemented computation and the other parts of the programming environment.

3.2.1 Integration at Task Level

Integrating hardware implemented computations as tasks is the integration means which is the closest to the algorithm level. Such an integration usually relies on a specific programming paradigm or a domain-specific language. The hardware implemented computations are recognized as independent tasks (they may also be named actors, nodes or components) in the programming paradigm, where the communications amongst tasks are explicitly decided in most of the cases.

To achieve a task level integration, a high-level abstraction API has to be provided to adapt to the corresponding programming model or domain-specific language. Sometimes, the high-level abstraction API may be built up on thread implementations. Some examples of task level integration are described as follows.

3.2.1.1 FOSFOR

The FOSFOR project[GKM+12] targets applications written via a SDF model. The author intends to propose a full development flow to reduce the programming complexity of such application on a Heterogeneous System-on-Chip architecture.

At high level, the applications are described as a graph of SDF actors which receive and send certain amount of data tokens through virtual channels. The actors description is made from the standard graphical language UML (Unified Modeling Language). Their interfaces are described using Interface Description Language (IDL3).

At implementation level, the actors are refined as threads. software threads are managed by the RTEMS Operating System (OS), which is able to run on a multiprocessor target by using its *Multi-Processor Communication Interface*; while hardware

threads are managed by a flexible hardware OS, which provides the services including fast system call, thread management, semaphores, simple memory allocation and mailbox. The virtual channels and their corresponding managers are implemented as a middleware on top of the OS, explicitly transferring the exchanged data and implicitly maintaining state coherency through software and hardware.

The preemptive states have been added in the hardware thread Finite State Machines. However, the current scheduling strategy is a Highest-Priority First algorithm sorting statically partitioned thread. The context switch and relocation of hardware threads are left as open issues. The authors also plan to develop a load-balancing HW/SW partition mechanism and model transformation techniques to automatically generate the actors' code.

3.2.1.2 ReConfigME

ReConfigME [WK01, WKJ06] is a set of OS services, which are developed for allocating pure hardware implemented applications onto shared reconfigurable platforms, and for managing shared-memory data communications for each application.

All applications handled by ReConfigME must respect a data flow programming paradigm. Each application should be structured as a data flow graph, which consists of computation nodes and logic connections amongst the nodes. As inputs of ReConfigME, each node is described by an EDIF file, and a JAVA class file is provided to define how EDIF files are connected together.

The ReConfigME is responsible to place the input application, generating the corresponding bitstreams and realizing the actual configuration procedure. An application can be configured to the platform only when all of its nodes can be placed on the FPGA, or the application is put into a waiting queue. In other words, a whole application is the minimum unit to be allocated on the reconfigurable fabrics. The decision of user to load and unload an application is transferred to ReConfigME through a user command-line interface.

As for the aspect of communication, each application is assigned to a dedicated segment of on-board memory. The nodes of the application access the segment of memory through a memory controller. A remote host can send the stimulating data to and collect the final result from the memory segment through a network protocol.

The ReConfigME framework does not support dynamic reconfiguration or partial reconfiguration. When a new application is about to be added to the platform, the clock of FPGA is stopped. The bitstream of the whole FPGA is read out, merged with the bitstream of the newly-added application, then reloaded back to FPGA.

3.2.1.3 SPORE

Simple Parallel platfOrm for Reconfigurable Environment (SPORE) [FMG13] is a general theoretical platform which adapts a High Performance Computers (HPC) topology, with nodes composed of computing processing elements and a communication-dedicated element.

Close to OpenCL [ope11], applications in SPORE are viewing as a gathering of independent kernels communicating with each other. All the kernels have both

a software and a hardware implementation, the one eventually used is chosen at execution time depending on available resources and process criticality.

To achieve this HW/SW online codesign, the author proposed a virtualization mechanism to automatically manage kernels and their access. Sequential action sets are carried out by a kernel implementation as the way to deal with the starting and parameters setting of the kernel implementation.

Currently, SPORE has two implementations. One is software HPC platform, which use MPI [For12] for inter-node communication; the other is hardware stream dynamic platform, which is rather dataflow-oriented. Both platform are evaluated in terms of resource usage and job running time. The evaluation revealed that memory is a bottleneck issue led by the symmetric multiprocessing characteristic of their platform. The author intended to develop the third SPORE implementation, in which both the nodes and kernel implementations can have hybrid HW/SW nature.

3.2.1.4 Flextiles

In the Flextiles project [LMA+12], the application is described as a set of actors, that respects a dataflow model of computation. This on going project aims at providing a virtualization layer in order to mask the underlying heterogeneity of the reconfigurable architecture. Their virtualisation layer will provide self-adaptation capabilities by dynamically relocation of application tasks to software on the manycore or to hardware on the reconfigurable area. Beside the code location, the virtualization layer should also manage on the fly the storage and communication paths. To make this possible, the project also proposes a specific reconfigurable technology based on a virtual bitstream that allows dynamic relocation of accelerators just as software based on virtual binary code allows task relocation.

3.2.2 Integration at Process Level

3.2.2.1 BORPH

BORPH [hSAS+07] provides kernel support for FPGA applications by extending a standard Linux operating system. In BORPH, an instance of a program executing on the reconfigurable fabric is recognized as a hardware process, which is an active independent executing entity equivalent to the conventional software process.

A hardware process is an executing BORPH object file. Such a file is a binary file format that encapsulates, among other information, configuration for reconfigurable fabrics. In order to handle hardware processes, BORPH makes use of an extensible interface provided by the standard Linux kernel and integrates a new binary file format kernel module. The interface provides a system-call-consisted API for user defined binary file formats. The newly-added kernel module serves the requests passed by the extensible interface, such as allocating and configuring the necessary reconfigurable resources during process creation.

BORPH provides hardware processes a hybrid message passing system call interface for both accessing regular data files and to communicate with other processes in the system through UNIX's pipe construct. Besides, the `ioreg` virtual file system

allows passive communication from the controlling processor to gateware designs. Because of the kernel's involvement, access to FPGA resources may be initiated by any UNIX programs: from simple shell scripts to complex compiled programs.

BORPH has been implemented for multi-FPGA platforms. The kernel/user interface of BORPH also makes it possible to employ dynamic partial reconfiguration on a FPGA.

3.2.3 Integration at Thread Level

There exists solutions which integrate hardware implemented computations at thread level. Such solutions have the following features in common. They all rely on a multi-threaded programming model. The computation implemented by hardware and software are handled in a unified manner, as threads share the same memory space. To achieve this, an API for hardware thread management is normally built up on the top of an OS, providing services such as scheduling, communication, synchronization, and so forth. A thread interface is usually implemented on the reconfigurable fabric for each hardware computation, providing to the OS the thread elements to manipulate. Some examples of thread level integration are described in the following.

3.2.3.1 HybridThread

HybridThread [ASA+08] is a POSIX-compliant multi-threaded programming model across the HW/SW boundary. It is composed of several middleware services and an extended operating system.

The middleware services provide a unified API to create, control and schedule all threads. The executing unit of a specific thread is indicated as an attribute of the creation function. That is to say, the partition is explicitly done in the application code. Each hardware thread has a dedicated system interface called *hardware thread interface*, which allows the hardware thread to execute autonomously and in parallel by supporting system call mechanism and shared memory accesses.

In order to have a promising performance, parts of the OS concerning the hardware thread management are migrated into hardware, including a mutex manager, a CPU bypass interrupt scheduler, a thread scheduler and a thread manager.

A compiler is demanded to automatically and correctly translate the application from a standard high-level programming language to hardware threads that can be synthesized for a specific target platform. As far as we get from the literature, the HybridThread framework neither supports dynamic nor partial reconfiguration.

3.2.3.2 ReconOS

ReconOS [LP07] is an execution environment built on the top of existing embedded operating systems, extending shared memory multi-threaded programming model from the software domain to reconfigurable hardware.

ReconOS employs hardware threads through a dedicated API which is similar to but different from the POSIX or eCos kernel API used by software threads. The HW/SW thread interfacing problem is addressed by using the same OS objects for

thread communication and synchronization. Each hardware thread has its own software proxy thread, called delegate thread. The delegate thread is responsible to maintain the OS kernel-mapped objects used by the associated hardware thread.

Hardware thread is structured by two hardware parts: user logic and an OS interface. The user logic implements the computation demanded by the application. The OS interface provides thread supervision and control. Inside this OS interface, there are VHDL-implemented procedures which govern the system calls required by the hardware thread. These system calls are either transformed as an access to shared memory or dedicated hardware FIFO buffer, or relayed to the corresponding delegated thread to execute a software OS call on behalf of the hardware thread.

The current ReconOS prototypes hardware threads are statically configured. The support of dynamic partial reconfiguration is planned in future work by the authors [LP09].

3.2.3.3 FUSE

FUSE [IS11] is a front-end user framework intending to ease the migration of software-implemented tasks to hardware. To attain this goal, the authors proposed a specific creation/destroy API and some OS support organized in two layers.

The API acts as the wrapper of corresponding POSIX thread creation/destruction functions, augmenting their abilities by supporting hardware tasks when a platform contains hardware accelerators. The user-layer OS supports, called *top-level FUSE component*, provide a decision flow to firstly partition a task on hardware when possible and secondly to assign the task to software otherwise. Thanks to the API and top-level FUSE component, the semantic of task creation/destruction code remains identical in the application program, whatever the underlying platform is.

Although the creation of hardware tasks is encapsulated in POSIX-like thread creation function, the communication and synchronization between software and hardware are not managed at thread-level. In the FUSE framework, all tasks implemented in hardware are considered as memory-mapped I/O devices.

The kernel-layer OS supports, called *low-level FUSE component*, provides the typical file system services (open, close,read, write, iocrl and mmap) to give access to hardware tasks. The device drivers in the low-level FUSE component are customized for each specific hardware task. They can be dynamically loaded to kernel at runtime.

A hardware accelerator interface is added to each hardware task. However, unlike the hardware thread interface in HybridThread or the OS interface in ReconOS, this hardware accelerator interface does not support the encapsulated hardware task to actively communicate with the remaining parts of the system.

The current version of FUSE partly provides the dynamic feature by employing on-demand loaded device drivers to manage the already existing hardware accelerators. The fully partial dynamic reconfiguration support needs further research on reconfigurable resource management.

3.2.3.4 SPREAD

SPREAD [WZW+13] is another HW/SW multi-threaded programming model built up on the top of an extended OS. The particularity is that SPREAD was designed specially for the streaming applications and SPREAD supports dynamic computing resource allocation, and runtime HW/SW switching.

Resembling to ReconOS, SPREAD provides a dedicated API for creating, terminating or switching hardware threads. However, a stream programming library provides a set of unified communication and synchronization services for both software threads and hardware threads. The inter-thread communications are explicitly implemented through data-driven point-to-point streaming channels.

On the platform, each user-defined function is encapsulated by a *hardware thread interface*. Apart from the thread state controlling, the hardware thread interface mainly provides two full-duplex, synchronized stream interfaces, so that the hardware threads can communicate with software threads and hardware threads. Each hardware thread in SPREAD has its software delegate thread, called stub thread, which monitors the hardware thread interface and maintains the stream communication primitives located in OS kernel for the hardware thread.

A reconfigurable computing resource, a hardware thread manager and a stream manager are added to the OS kernel. They operate concurrently to implement the hardware task allocation, the HW/SW thread switch and stream redirection at runtime.

3.2.3.5 Virtual Memory System

[VPI05] introduces a hardware-agnostic multi-threaded programming paradigm. All threads in such a programming paradigm communicate implicitly through a shared virtual memory space. The programming paradigm is achieved by the support of a predefined hardware thread library and a virtual memory system.

Each hardware accelerator has a software wrapper. The software wrapper is responsible to activate the corresponding hardware accelerator, and to pass the associated virtual memory space and size of expected data to the hardware accelerator. The name of the software wrapper is used as the identifier in the hardware thread library. In the application, a software wrapper is invoked during a thread creation procedure, as the implementing function of the thread.

The virtual memory system consists of several window management units physically linked to each statically-located hardware accelerator and a virtual-memory window manager. The window management units, equivalent to memory management units for CPUs, map the virtual memory address used by the hardware accelerator onto physical memory addresses. The virtual memory window manager, as a supplement to conventional virtual memory managers, ensures memory consistency while providing the standardized OS data-transfer services to user space libraries and applications.

3.2.4 Integration at Instruction-Set Processor Level

Integrating hardware implemented computations as *customized instruction* is the integration way which the closest to the hardware architecture level. Such an integration is usually tailored to a special application domain. The most-frequently used patterns in the application are implemented as extra function units in the data-path of the processor core. The HW/SW interfacing is normally implicit to a programmer, realized by a bypass circuit which redirects corresponding data to instruction executing units on the reconfigurable fabric.

To achieve instruction-set level integration, a series of tools are normally required. The design space is usually explored based on the result of a profiling tool, which is able to find the hot spots in applications in an architecture-independent manner. The customized instructions can be identified and re-targeted to a reconfigurable fabric manually by the programmer or automatically by a compiler. The generation of customized instructions can be achieved through a separated synthesis tool chain from a HDL description. Sometimes, the synthesis is also integrated inside the compiler to generate customized instructions directly from the application. Some examples of instruction-set level integration are described in the following.

3.2.4.1 Chimaera

Chimaera [YMHB00] tightly couples a superscalar processor and a *reconfigurable functional unit* on a small and fast FPGA-like device. This unit is capable of performing 9-input/1-output operations on integer data. The data is exchanged via the *shadow register file*. An *execution control unit* communicates with the control logic of the host processor for coordinating the execution of the reconfigurable functional unit operations.

A modified version of GCC provides the compiler support for Chimaera. It automatically maps groups of instructions to the reconfigurable functional unit operations (RFUOP). At the same time, it performs instruction combination, control localization and SIMD within a register. The three RFUOP-specific optimizations offer significant performance improvements even under pessimistic assumptions.

Upon detection of an RFUOP, the execution control unit is able to initiate a trap to load the appropriate configuration at runtime. However, while the configuration is being loaded, execution is stalled. Moreover, if the working set of the RFUOPs is relatively large, the problem of thrashing in the configuration array is reported.

3.2.4.2 XiRisc

eXtended Instruction Set RISC (XiRisc) [LTC+03] is a VLIW based processor, which is enhanced with an additional pipelined runtime configurable data-path (PiCo Gate Array, or PiCoGA). The PiCoGA acts as a repository of virtual application-specific multi-cycle instructions. The PiCoGA is tightly integrated in the processor core, receiving inputs from and writing back results to the register file of the processor. Synchronization and consistency between the normal program flow and PiCoGA

elaboration is granted by hardware stall logic based on a register locking mechanism, which handles read-after-write hazards.

The critical computations that should be implemented on the PiCoGA are manually determined, based on a software profiling environment. That is to say, the programmers have to be aware of the application-specific instructions when they write the programming code. Dynamic reconfiguration is handled by a special assembly instruction, which means that the instruction decoder circuits should be modified to recognize the new added instructions.

3.2.4.3 MOLEN

The Molen architecture [SB09] consists of two parts: the GPP and the tightly coupled reconfigurable processor usually implemented on a FPGA. An arbiter performs a partial decoding of the instructions received from the instruction fetch unit and issue them to the appropriate processor. The *exchange registers* are used for data communication between the core GPP and the reconfigurable processor. Parameters are moved from the register file to the exchange registers and the results stored back from the exchange registers to the register file.

A runtime environment, including a scheduler, a profiler and a transformer, decides on which processor each instruction should be executed. In the Molen framework, a compiler assisted task scheduling takes place in two phases [SSB09].

First at compile-time, the compiler performs static scheduling of the reconfigurations requests (by SET and EXECUTE instructions) assuming a single application executing, in order to hide the reconfiguration delay by configuring the operations well in advance before the execution point. Then at runtime, the scheduler should make the decision based on the runtime statistics recorded by the profiler. The transformer has to replace the software instruction with a call of pre-synthesized hardware implementation. The SET and EXECUTION instructions are only conditions to invoke the scheduler. It is possible to run an instruction in software even though the compiler already scheduled the configuration on the reconfigurable hardware.

In the current version, the reconfigurable processor implemented instructions are not relocatable. They are all pre-synthesized with a fixed physical mapping location.

3.2.4.4 RISPP

Rotating Instruction Set Processing Platform (RISPP) [BSH08] distinguishes itself from other instruction-set integrations by separating the notion of a data path from a *special instruction* for an application. A special instruction is the combination of several data paths. A single data path can be reused by several special instructions. A data path can be used as soon as it is reconfigured. The RISPP allows a special instruction to be implemented with only a subset of its required data path, in order to improve the efficiency of the hardware usage.

Different implementation possibilities exist for each special instruction, which employ different trade-offs between the amount of required accelerating data paths and the achieved performances. These implementations are prepared at compile time. A runtime manager is used to control the special instruction executions,

upgrading the performance of a special instruction implementation at runtime by gradually loading the corresponding data paths. In the case that the reconfigurable hardware does not support a requested special instruction (either because of the lack of reconfigurable hardware or the low expectation of its performance), a trap is activated to call a corresponding pure software implementation for it.

The authors spent a large effort to automatically detect special instructions. However, the partition of special instructions into data paths are still manually developed.

3.3 Conclusion

This chapter provides a background on computational model and reconfigurable architectures. To support reconfigurable computing, a broad range of integration strategies exist. This chapter covers them thoroughly from the one closely related to the computation model to the one tightly-coupled to a specific reconfiguration architecture. It should help the reader to position contributions described in following chapters on these axis.

Chapter 4

An Abstraction Layer for Dynamic Reconfiguration

IN this chapter, we are going to introduce an abstraction layer between the application and the hardware platform. This layer separates task allocation from FPGA reconfiguration procedure, by abstracting different kinds of reconfigurable fabrics and providing a uniform allocation service to the upper layers.

4.1 Hypotheses on the System

Looking for simplicity and willing to search a solution implementable in a lightweight manner led us up to the following hypotheses on the hardware architecture and software environment, which form the base of our work.

4.1.1 Targeted Hardware Architecture Template

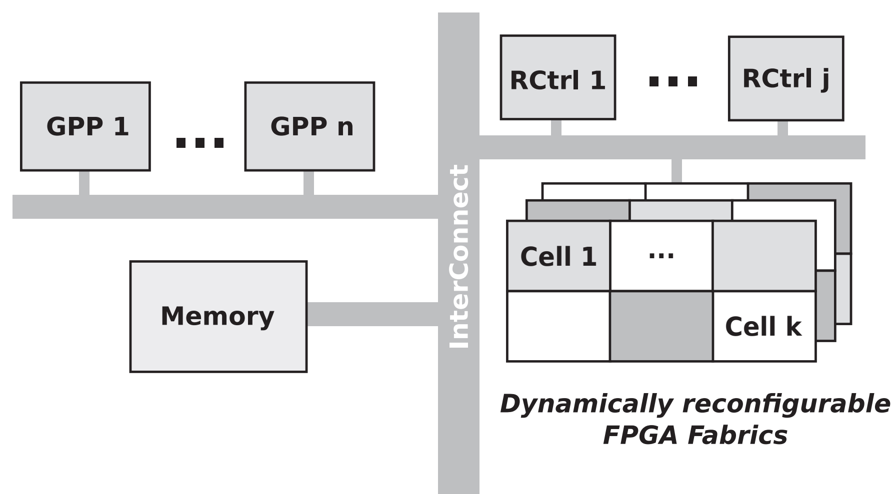


Figure 4.1: The targeted hardware architecture template

The targeted hardware architecture template is represented in Figure 4.1. The proposed architecture is based on cells located in one or several FPGAs with *dynamic partial reconfiguration* (DPR) capabilities. The cells can either be FPGA parts or a full FPGA. The architecture template also contains *reconfiguration controllers* (called RCtrl in the figure), *general-purpose processors* (GPPs) and memories.

4.1.1.1 Cells and the Homogeneous Choice

We make the assumption that the cells present in the architecture are homogeneous. This brings several advantages, that we outline now.

Bitstream Size and Structure

First, the homogeneous choice implies that two configuration files, called bitstreams, which configure different cells with different functions, are similar in terms of size and structure [KLPR05]. By structure, we mean the sequence of configuration commands and configuration memory address space. This allows a better management of bitstreams in memory.

Generation of Bitstreams and Storage Size

For each complete computing function implemented on the RF, a bitstream must be available. Normally, a bitstream is dependent on the position where the function is mapped to. However, thanks to the homogeneous cell choice, only one configuration bitstream per function has to be generated and this bitstream can be relocated on each cell of the architecture template [CMN+09] with just a bit of modification. This allows a considerable reduction of the configuration storage required.

In addition, homogeneous cells allow to ease the bitstream generation process, since a function has to be synthesized only once with area constraints associated to cell resources, which results in less synthesis number and shorter synthesis time.

Interconnect Infrastructure

Another consequence of the homogeneous cell choice is that the interconnect is static and standard interfaces can be used, as proposed for example in [HKHT05], thus simplifying at the same time the communication management.

4.1.1.2 Reconfiguration Controllers

The architecture template also integrates one or several reconfiguration controllers. Each reconfiguration controller can configure one cell at a time among the cells it can access. The reconfiguration controller is responsible for adapting a bitstream to a specific chosen cell. By doing so, a function can be relocated to any cell in the architecture template, while keeping only one bitstream in storage.

Therefore, the template can support multi-FPGA architectures and FPGA with multi-programming ports as proposed in [QSN06] – even if it does not exist in current FPGAs.

4.1.1.3 General-Purpose Processors

The architecture template also contains one or several GPPs to benefit from software modularity and ease of use. Multi-GPP is considered to ensure software performance. The GPPs can be conventional micro-processors, multi-core processors, or soft cores in FPGAs.

4.1.2 Software Environment Assumptions

Having discussed the hypotheses on the hardware template, we are now going to make assumptions about the software environment in the GPP/FPGA hybrid reconfigurable system. The assumptions mainly concern the task partition, the way to handle multitasking for functions running on the reconfigurable fabrics, and how the bitstreams are generated and recognized by the software environment.

4.1.2.1 Explicit Partition

Normally, application programmers have the best knowledge of their algorithms and the most efficient way of implementing them. Thus, we assume that an explicit partition of tasks is employed in our context, i.e. while writing applications, programmers are aware of which parts should be accelerated by putting them onto reconfigurable fabrics.

The explicit partition avoids the efforts of an attempt of resource assignment between hardware and software, and so simplifies the run-time task management infrastructure. Runtime assignment of tasks to hardware or software is out of the scope of this work.

4.1.2.2 Cooperative Multitasking

We assume that the tasks implemented on the reconfigurable fabric are cooperative. That is to say, once a hardware task begins to compute, it will be executed until the end of the task. The reason why we employ this non-preemptive multitasking way is that we would like to keep things simple.

On one hand, the context switch of hardware tasks is rather expensive. This is because it includes not only the storing and reloading the execution state of the two hardware tasks, but also a procedure of reconfiguration of the switched-in hardware task. Both of them require a period of time and space in the memory.

On the other hand, a definition of "execution state" for all hardware tasks is difficult, if not impossible. The implementation of each hardware task is designed to adapt to the function that it realizes. Some hardware task implementations contain many state machines, others are more data-dependent. Even during the different stages of a specific hardware task, the context (states of state machines, data under processing) needed to be stored may be different.

It is to note that efficient hardware context-switch solutions has been proposed in [GG09]. However, these solutions are restricted to dedicated reconfigurable fabrics.

By employing the cooperative multitasking way for the hardware tasks, we avoid the overhead and complexity of context-switch. If the task can not work in

a cooperative manner (e.g. infinite tasks), it is the responsibility of the hardware designer to implement it with preemption points.

4.1.2.3 Pre-synthesized User Library

We assume that the functions implemented on the reconfigurable fabric are pre-synthesized and recognized by the software environment as a user library.

We chose to pre-synthesize the functions running on the reconfigurable fabric mainly for performance reason, since to the best of our knowledge, fast and reliable just-in-time compile-synthesis methods have not been developed yet.

Since the preparation and the usage are separated, the bitstreams can be generated in various ways. They can be synthesized from VHDL or Verilog models written by professional hardware designers. Alternatively, a high level synthesis tool [CM08] is able to generate the bitstream from the models written in high level language (the C or C++ languages for instance). By doing so, a homogeneous programming environment is preserved, leaving application programmers free from underlying hardware details. After synthesis, the bitstreams of functions running on the reconfigurable fabric are platform dependant.

To be recognized by the software environment and used by applications, the information of pre-synthesized functions forms a user library. The library contains the information such as the identification of the function, the size and storage address of the corresponding bitstream, and the offset of the control and status registers of each function.

The size of the library is decided by the number of the pre-synthesized functions supported by the platform. If the kinds of functions are changed or the number functions is extended or reduced depending on the application needs or the platform resource condition, the pre-synthesized user library should be recompiled accordingly.

4.2 A Motivating Example

In this section, we are going to analyze a piece of application code which explicitly manages the configuration of hardware tasks. By pointing out the drawbacks of this approach, we state the motivation of our work.

Figure 4.2 illustrates the task assignment in a GPP/FPGA hybrid platform which satisfies the hypotheses in section 4.1. The platform contains a certain number of GPPs and several homogeneous cells. There are two kinds of tasks in the application. The software tasks assigned to GPPs are represented by the light-grey ellipses, while the hardware tasks assigned to cells are represented by the dark-grey ellipses. On the left, a piece of pseudo-code is listed, mapping the software task S to a GPP by a standard function call and mapping the hardware task H to FPGA by a sequence of operations on cell 2.

In this example, the drawbacks of the way of managing hardware task H are obvious. The code is hard to write and maintain, lacks flexibility, and is error-prone in a multi-user context.

```
10  void main() {
      . . .
20    /* Assigning SW task S to GPP */
21    Function_S(parameter_list_of_S);
      . . .
30    /* Assigning HW task H to FPGA */
31    while(CELL2_STATUS == OCCUPIED);
32    cfig_to_FPGA(H, CELL2, RCTRL_FOR_CELL2);
33    while(CELL2_STATUS != CFIG_FINISH);
34    active_compute_on(CELL2);
35    while(CELL2_STATUS != CALC_FINISH);
      . . .
100 }
```

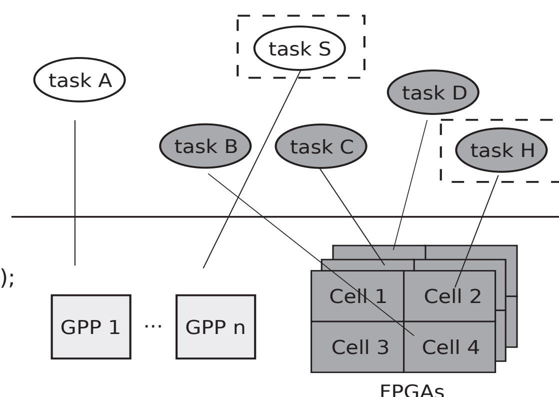


Figure 4.2: Task Assignment in a GPP/FPGA Hybrid Platform

4.2.1 Development and Maintenance of the Code

When writing the code as in the example, the programmer needs to handle the cell 2 usage himself, and to explicitly calls the reconfiguration procedure. When there are several hardware tasks to manage, the programmer has to calculate the reconfiguration delay and the calculation duration of each function in advance. Based on the timing information, the programme has to schedule the right moment to launch each cell-related operation by hand. The fact that the programmer has to arrange everything explicitly by hand makes the code hard to write and very specific.

The code in the example is hard to maintain too. First of all, the function of task H is implicitly called, hidden by the use of cell 2. For a maintainer who is not the original programmer of this piece of code, it requires much effort to understand the algorithm intention through a sequence of tedious operations of a cell in the platform. Besides, the code in the example is clearly platform dependant. The slightest problem of platform, such as the damage of some gates inside cell 2 or malfunction of its reconfiguration controller, would mean a complete failure of this piece of application. The correction of such problem demands a rescheduling of the resources, and thus a complete rewriting of the application.

4.2.2 Flexibility of the Code

Since resource management information are hard-coded inside the application, the code in the example lacks flexibility, in terms of portability and scalability. Even if new reconfigurable resources are added to the platform, such an application cannot benefit from the potential increase in computation power resulting from the update, let alone a complete change of platform.

4.2.3 Working in the Multi-User Context

The code in the example is very error-prone in a multi-user context. For example, the interval between checking the available resource (line 31) and sending a configuration request (line 32) may put the application into a race condition. If there is another application, which intends to assign to cell 2 another hardware task different from task H; and if unfortunately both applications get the information that cell 2 is not occupied at the same time, then one of the simultaneously sent configuration requests will be overwritten by the other, and the first application will be waiting for a function that will never be available.

In order to have the complete control to sequentialize the usage of resources in common, the two applications sharing cell 2 should be merged as one big application. However, in most cases, these applications are completely independent from the algorithm point of view. It is irrelevant to put such applications together.

Since different users do not know and have no need to know the reconfigurable resource usage of other users, the above strategy is a dead-end. Lacking of an arbiter of shared reconfigurable resources, no guarantee is given to any application to ensure the exclusive reconfiguration resource usage.

4.2.4 Motivation of our Work

The analyses above provide the motivation of our work: we are going to provide a mechanism which allows the programming code of the applications running on a GPP/FPGA hybrid reconfigurable platform (1) to be easy to write and maintain, (2) to be flexible in terms of portability and scalability, and (3) to be able to work in a multi-user context with the guarantee of the exclusive usage of shared reconfigurable resources.

4.3 Proposal: An Abstraction Layer Wrapping Hardware Components

In order to achieve the objectives that we set in the last section, applications need to be more clearly separated from the underlying architecture. Our proposal relies on the following observation of the interactions between the elements of the template.

4.3.1 Observed Interactions

Depending upon the elements involved in a computation implemented on hardware, different information is needed. In order to program an application, we need a function identification, information on the availability of the function on the platform, and on timings and methods to access the function. The application does not need to know the location of the function (which cell is used) or configuration procedure (which reconfiguration controller is used and how to use it).

The reconfiguration controllers are slave elements and do not require knowledge on which application asked for the reconfiguration. The only required information

is the location and size of the corresponding bitstream for a reconfiguration, and the cell which must be reconfigured.

Cells are designed to contain functions. They passively accept the configuration bitstreams during reconfiguration and report their status when computation is finished. No knowledge on configuration choices (what and when) is needed in the cells, and they do not need to initiate communication with other elements.

Based on this analysis, two layers can be distinguished. On one hand, the application layer which needs functions for computation, expects them to be available, and should be notified when the computation finishes. On the other hand, a *Reconfigurable Hardware Resource* (RHR) layer, which encompasses the reconfiguration controllers and the cells, in which the reconfiguration and computation actually happens but which does not have knowledge on the implemented functionalities. It is important to keep both layers separated, and to mask the specificities of each layer. We propose a new abstraction layer between the application layer and the RHR layer. It is designed to provide a uniform *Application Programming Interface* (API) to all applications, and a standard control interface to the RHR layer. This abstraction layer is further described in the following subsections.

4.3.2 Hardware Component: the Key Element for Interfacing with Application Layer

In order to ease the life of application programmers, we would like to provide an abstraction of the program running on the reconfigurable fabric. This abstraction, called hardware component or component, should extract only the piece of information concerning its functionality and its execution properties. With hardware components, application code can be written by a sequence of operations on the abstraction, instead of explicit management of elements in the RHR layer. Such code is not sensitive to the change of underlying platform, thus can be flexible. The properties and operations of hardware component are described in details as follows.

4.3.2.1 Hardware Component Properties

Based on the observations on the application layer, the hardware component properties are listed in Table 4.1. There follows some further descriptions about certain properties, in order to provide more details and to distinguish the potentially confusing properties from each other.

COMPONENT_ID v.s. FUNCTION_ID

Each hardware task mapped to a cell is identified as a specific hardware component, thus owning a unique `COMPONENT_ID`. The hardware component is the programming item that application programmers are able to operate.

Each hardware component is dedicated to implement a function, marked by a `FUNCTION_ID`. The function is purely software, independent of any implementing unit, either a cell or a general purpose processor.

A specific function may have several instances on different cells. In other words, hardware components with different `COMPONENT_ID` may share the same

Table 4.1: Hardware Component Properties

Property Name	Comment
COMPONENT_ID	unique identification of the hardware component in the system
FUNCTION_ID	the identification of the function of the hardware component
COMPONENT_STATE	the processing state of the hardware component
BOUNDED_CELL_ID	the identification of the cell on which the hardware component is executing
SYNC_ITEM	the synchronization item of the hardware component
PRIORITY	the priority of hardware component
INIT_FCN	the function of hardware component initialization
INIT_PARAM	parameters of hardware component initialization function
ACTIVATE_FNC	the function of hardware component activation
ACTIVATE_PARAM	parameters of hardware component activation function

FUNCTION_ID.

COMPONENT_STATE
A hardware component is an abstraction used by application programmers. COMPONENT_STATE reflects both the status of underlying reconfigurable resources (the corresponding cell, the reconfiguration controller) and the interaction with software environment (the other parts of the program). At any moment, the hardware component must be in one of the five following states:

- **not_exist**: the hardware component is not attached to any cell from the software point of view;
- **waiting_cfig_finish**: the hardware component is needed to act as a part of the application, but the configuration of the corresponding cell is still in progress on the platform;
- **cfig_finish**: the configuration of the corresponding cell is finished, but the hardware component has not yet been used to act as a part of the application;
- **computing**: the hardware component is computing, acting a part of application;
- **calc_finish**: the hardware component has completed the function which is needed as a part of the application, but it is still attached to the corresponding cell from the software point of view.

BOUNDED_CELL_ID
As we have mentioned in the observation, the location of a hardware component

(which cell is used) is not important for the application. What an application programmer really cares about is the method to access the hardware component.

BOUNDED_CELL_ID identifies the cell on which the hardware component executes. However, the value of BOUNDED_CELL_ID here is NOT indicating the physical position of the hardware component. Instead, it is used as a reference to access the hardware component.

It can be translated by the operating system to the base address of the control registers of the hardware component, or to the identifications of communication properties that the hardware component can use.

When a different cell is assigned to the hardware component according to a different reconfiguration resource usage, it is sufficient to translate the changed BOUNDED_CELL_ID according to the unchanged operating system translation mechanism. The application can be kept untouched.

It is also the reason why hardware component is an abstraction instead of a virtualization: it does not hide all information of the underlying platform, but make the platform information easier to handle.

INIT_FUNC and ACTIVATE_FUNC

Usually, a hardware device may contain several memory-mapped registers as interface with the software, in order to get control orders and to report status. When such a hardware device is implemented in a reconfigurable way, these registers should be initiated to certain values once the reconfiguration completes.

In some cases, two functions use almost the same kind and size of resources when they are implemented on hardware. Such functions can share the same bitstream in order to reduce the storage memory footprints. A parameter is needed to be given before the computation starts, in order to distinguish between the two functions, which one is going to work.

In the both above cases, the initialization procedures may be different from one hardware component to another. The INIT_FUNC allows the programmers to give each hardware component a specific initialization according to the nature of the hardware component.

While the INIT_FUNC focuses on the hardware component itself, the ACTIVATE_FUNC concerns the software environment of the hardware component. For example, synchronizing with another task or setting up communications.

4.3.2.2 Hardware Component Operations

It is true that a function implemented on the reconfigurable fabric requires a configuration procedure, which introduces many reconfigurable resource management problems. However, as we observed in section 4.3.1, the application layer only needs to know which function is asked for, when it is available, how to access this function and when the required function has completed its job. These requirements can be transformed as the operations wrapping the hardware components, as shown in Table 4.2. Only these operations appear in the application code.

The above operations can be implemented depending on different system conditions and on the design choices of the programmer. For example, we have presented

Table 4.2: Hardware Component Services

Function Name	**Comment**
hwc_create	creating a hardware component, asking for the corresponding function to be made available on the platform.
hwc_activate	checking availability of the function asking for by hwc_create; initiating the available function, including the access method of this function.
hwc_destroy	releasing software resources attached to hardware component after the function finishing its job on the reconfigurable fabric.

a proof-of-concept implementation in Chapter 6. With the operation format in the proof-of-concept implementation, the motivated example can be rewritten as in Figure 4.3.

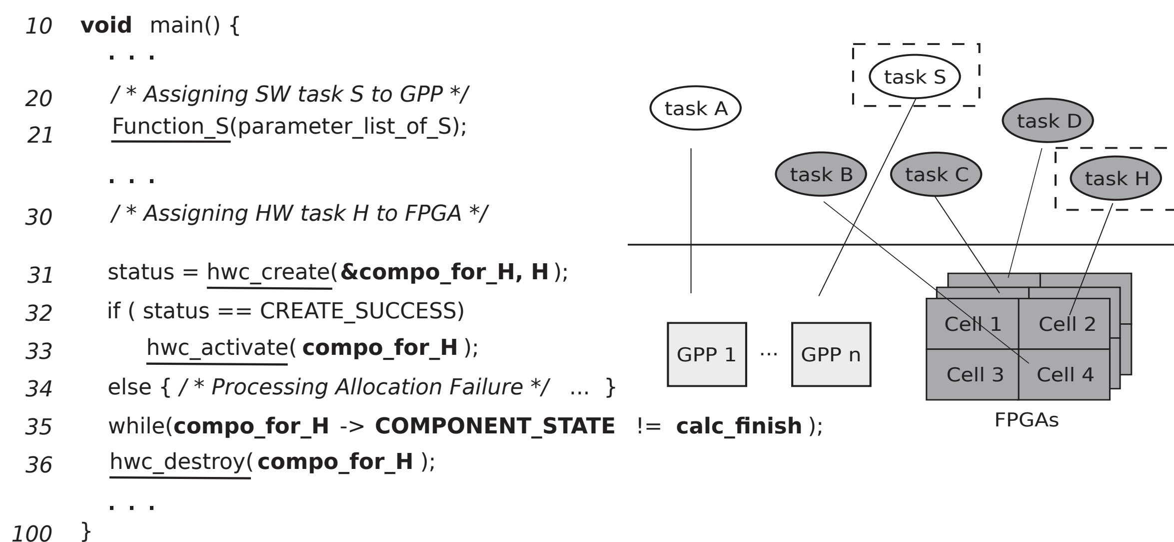


Figure 4.3: The Rewritten Motivated Example

Whatever the implementation, the hardware component and its operations provide a uniform API to the application programmer. We can see from the rewritten motivated example that thanks to this uniform API, reconfiguration management and status checking of Cell 2 is removed from the application code. The task H processing is managed through operations on the software avatar of the hardware component "compo_for_H". Compared with the original programming code of the motivated example, the rewritten code involves no platform information, thus becomes more flexible. With the hardware component centered API, the application is much easier to write, understand and maintain. Until now, we have achieved two of our motivations.

4.3.3 Hardware Component Manager: A Centralized Reconfigurable Hardware Resource Manager

Thanks to the hardware component centered API, the reconfiguration procedure has been removed from the application code. However, without reconfiguration procedure, the hardware component API is only an empty shell. In addition, looking back to our third motivation, we expect that reconfigurable hardware resources (including reconfiguration controllers and cells) can be shared correctly in a multi-user context. The elements in the RHR layer must be managed somewhere else than in the application.

In order to process information from application (through hardware component) and the RHR layer, a *Hardware Component Manager* (HCM) is integrated in the new abstraction layer. It is a centralized manager of reconfiguration controllers and cells. It is designed to manage the interfaces, to perform the placement of components on the cells, and to provide protection of cells in a multi-user context. The conceptual model of the HCM is presented in section 4.3.3.1, then an HCM implementation is described in section 4.3.3.2.

4.3.3.1 Conceptual Model of HCM

The HCM is located inside the new abstraction layer, which interfaces with the application and RHR layers. Information exchanged between the new abstraction layer and the other layers is presented in Figure 4.4.

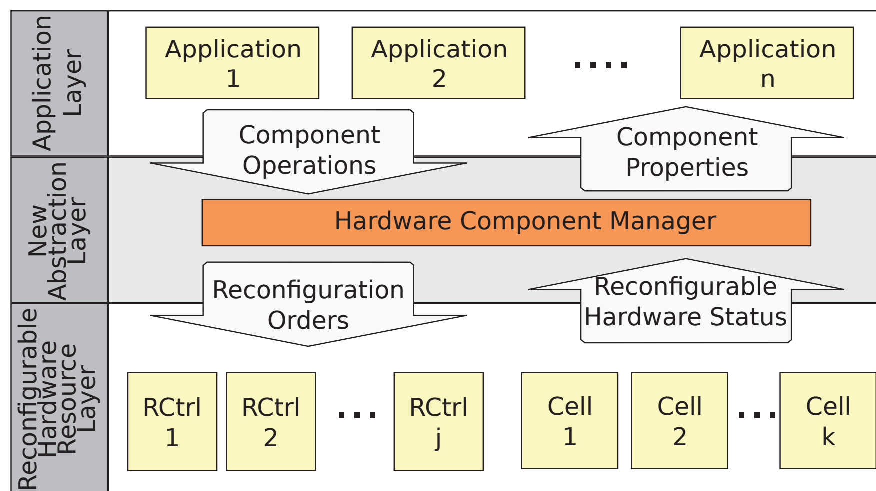


Figure 4.4: Hardware Component Manager in System

Communication between the application layer and the new abstraction layer is centered around the operations of hardware components. The applications can send the operations for making a component available in the platform for the application (*hwc_create*), enabling the component to work (*hwc_activate*), or releasing the component when the computation finished (*hwc_destroy*). These operations either request the HCM to provide a service, or wait for the HCM to report a change of status. To be more precise, the *hwc_create* operation requests the HCM with a hardware component allocation service. The concerned component identification is sent to the HCM,

which then sends back the result of the request, including the information on the availability of the component and on ways to access the allocated component. This information is contained in the component properties shown in Figure 4.4, and used by *hwc_activate* and *hwc_destroy* operations.

Communication between the HCM and the RHR is centered around reconfiguration procedures. Based on its internal decisions, described in the following paragraphs, the HCM sends reconfiguration orders to the selected reconfiguration controllers. A reconfiguration order contains the necessary platform-dependent information for a reconfiguration, such as the bitstream location or targeted cell. An order initiates the configuration process. This interface also collects status from the RHR such as the availability of a reconfiguration controller, or the status of a cell.

Having seen the communication between the HCM and two layers, we would like to detail the working mechanism of the HCM.

The HCM extracts information from allocation requests received from application through the *hwc_create* operation. Based on this information combined with internal information on the RHR usage, it selects a cell for allocation, and generates a reconfiguration order to be sent to a compatible reconfiguration controller when needed. It then reports to the application through the hardware component properties.

In order to place components while limiting the number of required reconfigurations when possible, and to guarantee the exclusive use of each cell, the HCM defines cell states and obeys a placement policy described as follows.

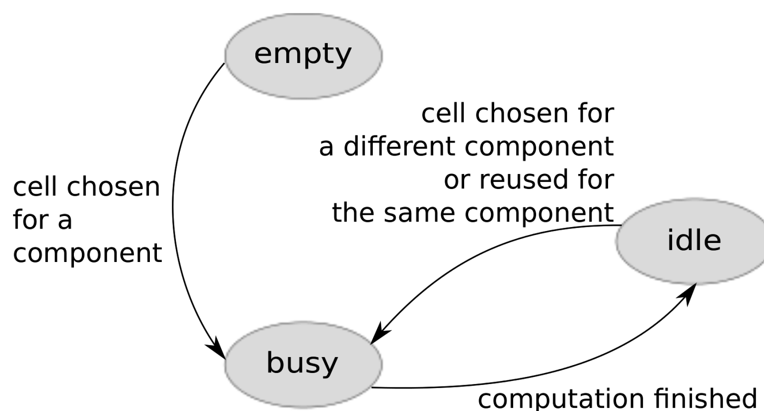


Figure 4.5: The State Machine of a Cell

Cell usage is monitored through the use of a state machine described in Figure 4.5. For each cell, there are three states recognized by a *Dynamic Resource Manager* (DRM, presented later):

- **empty**: The cell is not occupied by any component (only used at the beginning);
- **busy**: The cell is being configured, or it is occupied by a component which is busy computing;
- **idle**: The cell is occupied by a component which has finished its computation.

When the HCM searches for a cell to place the required component, only idle or empty cells can be chosen. Since we work in a non-preemptive model, a busy cell must not be reconfigured. The cell is chosen according to a placement strategy which is implementation-defined.

When a cell reports end of computation to the HCM, the latter searches the internal cell usage track to find out the corresponding hardware component. Then the HCM reports this event to the application through the hardware component properties.

4.3.3.2 An Implementation of the HCM

Figure 4.6 shows the internal structure of the HCM implementation. The HCM consists of an Application Interface (A_IF), a DRM and a Reconfigurable Hardware Resource Interface (RHR_IF).

For a given platform, the numbers of reconfiguration controllers and cells are fixed. We chose to implement the DRM in hardware, in order to have a shorter response time and to free the CPU from tedious reconfiguration control. The DRM consists of an *Available Cell Counter*, an *Allocation Request Dispatcher* (ARD), a *Cell Track Maintainer* (CTM) and an Interrupt Controller.

The A_IF is a software/hardware interface. The hardware part of A_IF contains several memory-mapped registers and an interrupt port. By accessing these registers and serving the interrupt, the software part of A_IF, called HCM driver, communicates with applications by receiving the requests translated from hardware component operations, reporting the results of these requests and the states of the hardware components, and providing inter-application protections.

When an application needs a hardware component to perform a function, the identification of the function (*fid*) is sent as a parameter of the *hwc_create* operation. After getting the necessary software resource and initiating certain properties, the *hwc_create* sends an allocation request *hcm_alloc* to the A_IF of the HCM implementation. The pointer of the hardware component is used as the parameter.

Having received the allocation request, the HCM driver does an atomic operation on the HCM_LOCK register inside A_IF (s1 in the figure 4.6). If the lock is taken by another allocation request, the driver has to wait and retry later. Otherwise, it gets the lock.

Then, the HCM driver has to check the `CELL_AVAILABILITY` register to avoid untreatable requests when no cells are available. In this case, a `NO_RESOURCE` acknowledge is returned to the *hwc_create* (s2). Otherwise, the driver fills the registers `FUNCTION_ID`, `BITSTREAM_BASE` and `BITSTREAM_SIZE` with the corresponding information extracted from the hardware component properties, in order to form an allocation command sequence.

Afterward, the `HCM_OPERATION` register is written to validate the allocation command sequence and automatically push the content of required registers into the input FIFO of the ARD (s3). The FIFO can not accept more requests than cells in the platform. At this point, the DRM is unlocked and becomes available for other applications.

The allocation request is then popped by the ARD. The dispatcher asks the CTM

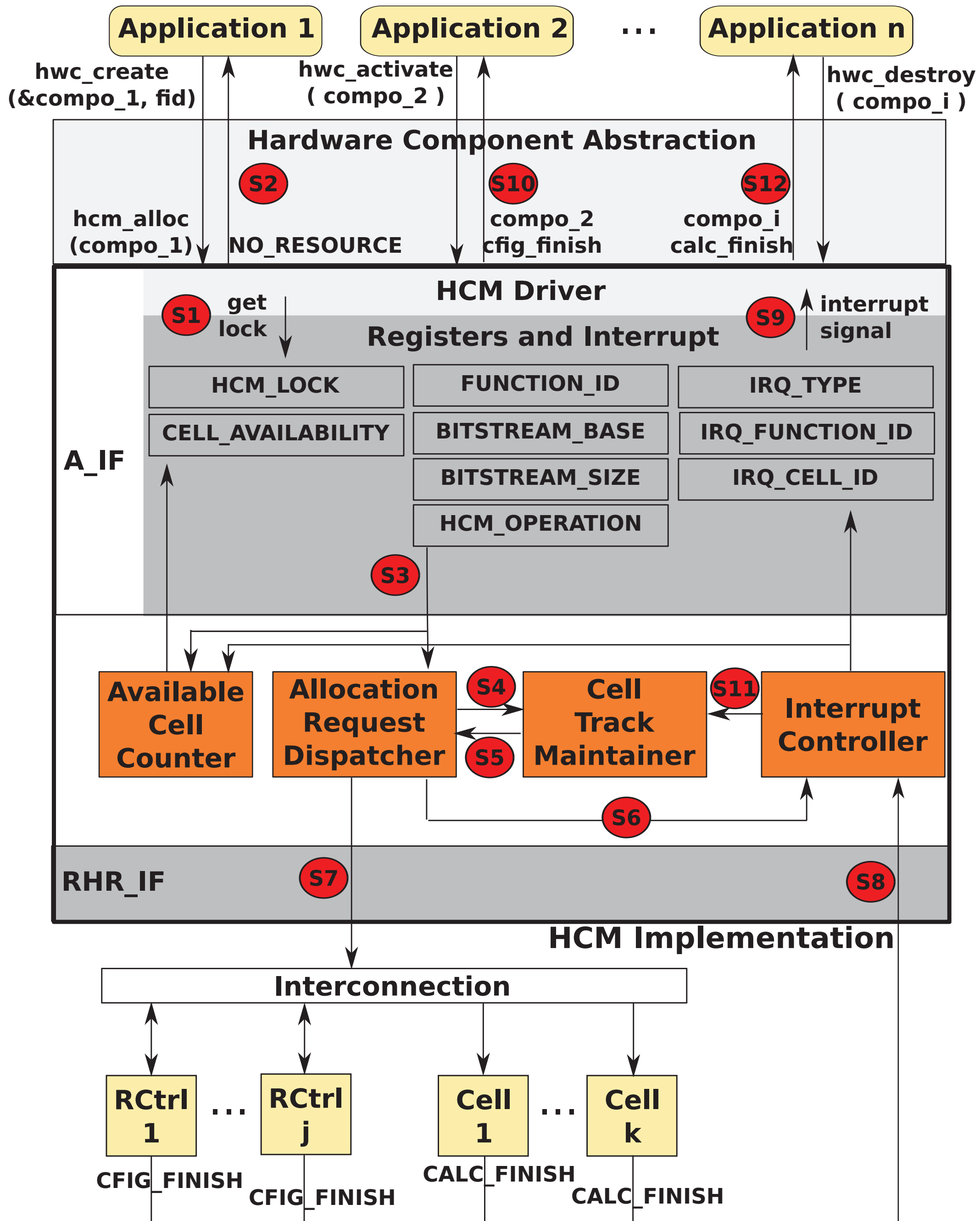


Figure 4.6: Internal Structure of the HCM Implementation

for a cell to place the required component (s4).

The CTM is composed of a cell usage table and matching circuits. The table is indexed by a global cell identification `cell_id`, i.e. one entry per cell, and each entry records the state of the cell and the `function_id` for non-empty cells. The CTM has to follow a placement policy to select a cell for the ARD. The implemented policy prioritizes cell reuse when possible, configures empty cells otherwise, and falls back to actual reconfiguration of idle cells as a default case. Once a demanded `function_id` is sent from the ARD, it is combined with state "idle" as a

comparison condition. All the entries in the table are compared with this condition. All matching entries are possible candidates for reuse. At the same time, a signal called `reusable_cell_found` is asserted to indicate the ARD that the proposed cell is reusable. Otherwise, empty (or idle) cells are selected for reconfiguration and the `reusable_cell_found` signal is deasserted. For each case, matching circuits return the lowest matching `cell_id`. Once the CTM has proposed a `cell_id` to the ARD (s5), it changes the state of corresponding entry as busy.

If the proposition is a reusable cell, the ARD sends the `function_id` and `cell_id` to the interrupt controller for issuing an interrupt later (s6); If the proposed cell needs to be reconfigured, the ARD transfers the associated reconfiguration order to a free reconfiguration controller through the RHR_IF (s7) and finishes its job.

Through the RHR_IF, the HCM can receive two kinds of interrupts: `CFIG_FINISH` from reconfiguration controllers and `CALC_FINISH` from cells (s8).

The interrupt controller collects with an input FIFO the interrupts issued by cells, reconfiguration controllers and the ARD. If this FIFO is not empty, the interrupt signal in A_IF is issued (s9). The interrupt service routine inside the HCM driver reads interrupt registers inside the A_IF hardware part.

When a component is placed on a reusable cell, or when a reconfiguration controller finishes its work, the interrupt service routine *hcm_isr* in the HCM driver chooses the first hardware component in the waiting queue which requires the function with `IRQ_FUNCTION_ID` and initializes its property `BOUNDED_CELL_ID` as the readback value `IRQ_CELL_ID`. The HCM driver then reports the availability by changing the property `COMPONENT_STATE` to `cfig_finish` (s10). The hardware component operation *hwc_activate* can then initiate the hardware component access address based on the property `BOUNDED_CELL_ID` value and system memory map.

When a cell finishes its job, the cell state inside the CTM is set to idle (s11).

In order to ensure the consistency of the `CELL_AVAILABILITY`, the available cell counter is kept up-to-date throughout the different processing steps. It is initialized with the number of cells in the platform, decreased for each `HCM_OPERATION` write and increased for each `CALC_FINISH` interrupt.

The `CALC_FINISH` interrupt is also processed by *hcm_isr*. The corresponding hardware component is found from the computing queue by the match of value `BOUNDED_CELL_ID` and `IRQ_CELL_ID`. The HCM driver reports this event by changing the property `COMPONENT_STATE` to `calc_finish` (s12).

4.4 Summary of the Chapter

In this chapter, we have proposed two important concepts. One is the Hardware Component, which is the abstraction of a program running on the reconfigurable fabric. The other is the Hardware Component Manager, which is a centralized reconfigurable hardware resource manager.

Through these two models, we have resolved the following problems:

By the hardware component abstraction, we have decoupled the execution of a program from the reconfigurable platform. This leads to two desirable results:

- The API built upon the hardware component makes the application codes

more straightforward to write, understand and maintain.

- The application code is no longer platform-dependent, thus more flexible in terms of portability and scalability.

Chapter 5

A Scalable Communication Mechanism for Dynamic Reconfiguration Platforms

In this chapter, we discuss a scalable communication mechanism for dynamic reconfiguration platforms based on the concept of FIFOs. As opposed to the single producer/single consumer FIFOs usually used in data-flow or Kahn Process Networks, FIFOs here accept several producer and consumer processes. It is important to note that from the application point of view, the channels are still single producer/single consumer FIFOs, the multiple nature being a requirement for the support of dynamic reconfiguration.

5.1 Communication Problems Brought by Dynamically Reconfigurable Platforms

In order to effectively use the FPGA/GPP hybrid platform, the applications are broken down into smaller parts, called tasks. The tasks are dynamically mapped to different processing elements and are exchanging information. The behavior of exchanged information amongst these parallel running tasks is called communication. The communication management is largely influenced by the communication participants, i.e. the tasks.

In our context, we assume that functions implemented by hardware components are passive data-driven tasks. The pseudo-code and state machine of such functions is shown in Figure 5.1.

After initialization, the start of the function depends on the availability of sufficient input data. Once this input data is processed and the result is properly obtained, a finish condition is checked. If the condition is fulfilled, the function completes its job; otherwise, the function continues to process the next set of input data. Of course, the function can be only a data producer (when the required number of input data equals to zero) or only a data consumer (when the finish condition does not include output data).

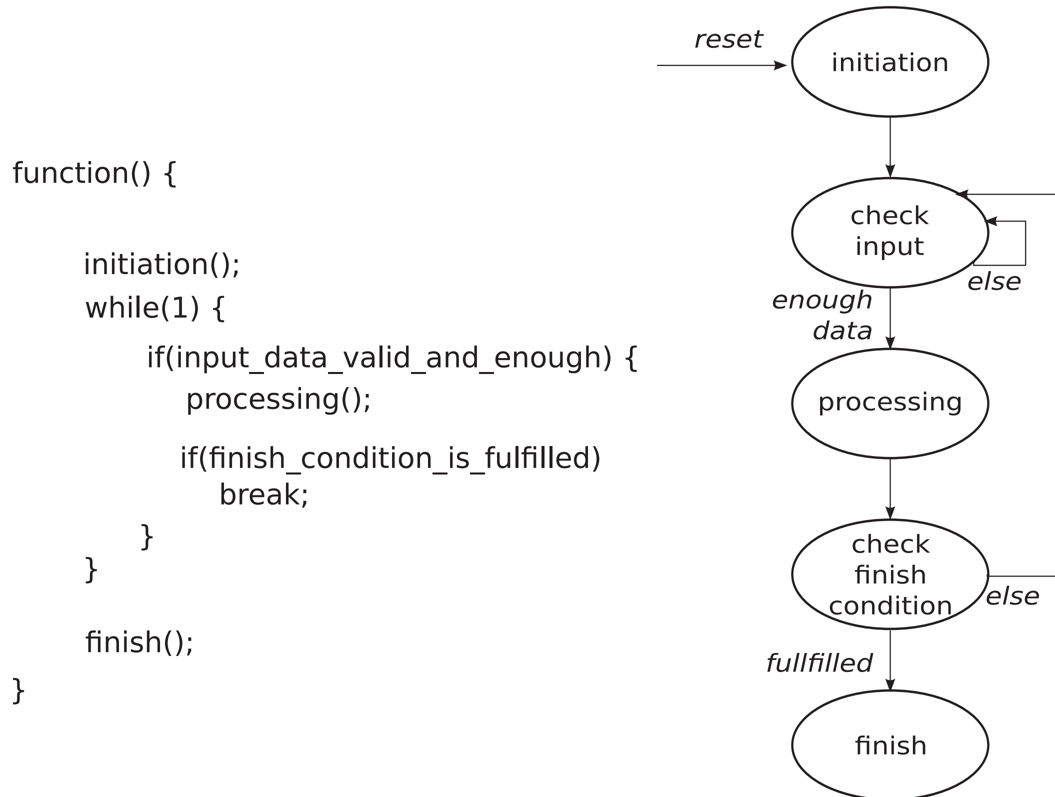


Figure 5.1: Passive Task Model

The reason why we made such an assumption is that before being employed by a specific application, a function cannot know exactly from which function the input data comes from or to which function the output data ought to be sent to. Such dependency can only be resolved when the task graph of the whole application is decided.

Based on such assumptions, the data transferring and processing phase are separated. On one hand, the design of individual tasks becomes easier, since they are well isolated by not including control related information. On the other hand, a communication mechanism is required to explicitly handle the synchronization and data exchange amongst the tasks.

5.1.1 A Motivating Example

In conventional cases, the task graph of an application is settled when the application is written. The communication scheme can thus be figured out accordingly at the same time and stay static. However, and as opposed to the hardwired case, in a dynamically reconfigurable platform, even a settled task graph only determines a logical communication scheme. The final physical communication scheme implementation depends on where exactly the tasks are mapped. Figure 5.2 illustrates such a case as a motivating example, in which we will see some communication issues which arrive because of the dynamicity of the underlying platform.

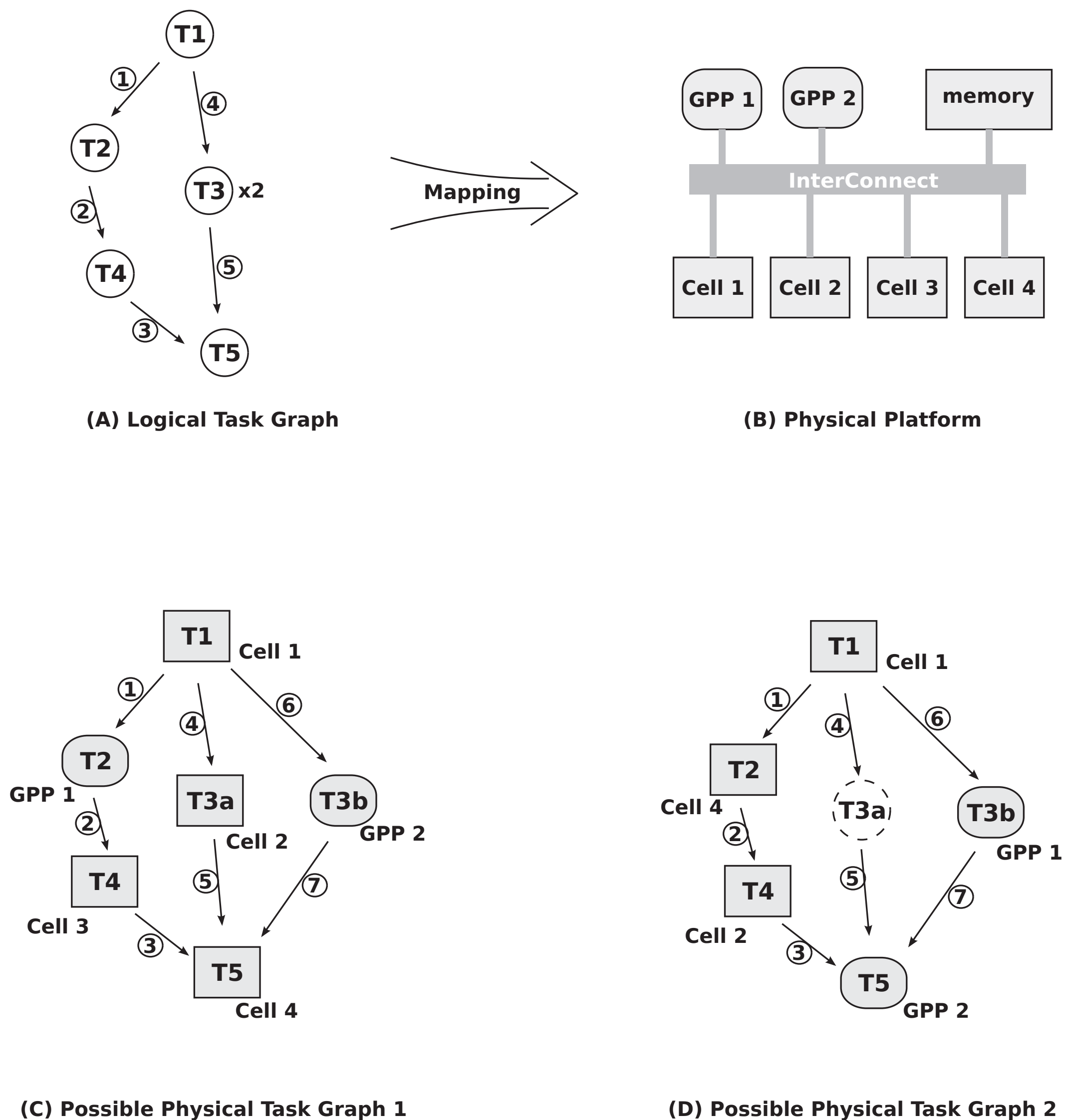


Figure 5.2: Communication Problems Caused by Dynamic Reconfigurable Platform

The subfigure A of the Figure 5.2 shows the settled logical task graph of an application. The subfigure B presents the underlying physical platform on which the logical task graph will be mapped. The subfigures C and D show two possible physical task graphs after mapping.

The logical task graph shows the data dependency amongst different tasks in a data flow application. Each task in the graph fits the passive function model given at the beginning of this section. All tasks in the graph may run in parallel. The synchronization of the tasks is achieved only by data transfered amongst them. The arrows between two tasks indicate the direction of the data flow. The task level parallelism can be performed as a pipeline (T1, T2, T4, T5 task sequence) or a task farm (T3 duplications), or the hybrid of the two (T1, T3 duplications, T5 task sequence).

The underlying platform is a GPP/FPGA hybrid platform, which contains two GPPs sharing the same memory space. The FPGA fabric has been divided into four

cells, which can be reconfigured to execute various hardware tasks at different moment. This underlying platform is shared by several applications, including the application described by the logical task graph at the left. It is possible that at a specific moment, the resources of the platform are not dedicated to only one application.

The physical task graphs show the resource usage of the application. A task in the logical task graph is now mapped to a GPP as a software task (presented by a rectangle with rounded corner in the Figure 5.2), or to a cell as a hardware task (presented by a rectangle). Otherwise, a task may be left unmapped (presented by a circle with dashed line) due to the lack of resource. The arrows are unidirectional single-input single-output FIFOs, responsible for data synchronization between every two tasks with dependency. It should be noted that the two physical task graphs in the Figure 5.2 are only two of numerous possible task mappings. The actual physical task graph depends on the number of other applications and their usage of the platform resources during the period when the concerned application is running.

5.1.2 Analysis of the Motivating Example

Observing the motivating example presented Figure 5.2, we can identify two kinds of communication problems arising due to the dynamic reconfigurable nature of the platform. One is related to the existence of the tasks, the other is related to the access of the already existing tasks.

5.1.2.1 The Existence of Tasks

In the logical task graph, we can differentiate two kinds of tasks: the surely-mapped tasks and the possibly-mapped ones. Surely-mapped tasks are the tasks that are mandatory to ensure application completion. Possibly-mapped tasks are optional tasks, that accelerate the overall processing. In the motivated example presented by Figure 5.2, T1, T2, T4, T5 and one of the T3 instances are surely-mapped tasks, the other T3 instance is a possibly-mapped task.

According to this mapping requirement, the task existence has a different meaning, resulting in different problems of task communication.

The Presence of Surely-Mapped Tasks

For a surely-mapped task, its existence in the physical task graph is guaranteed by definition. This does not mean that the task is mapped on the underlying physical platform all along the life time of the concerned application.

For example, if GPPs are used by other applications, T2 and T3b in the physical task graph (A), may be switched out for a moment. Even if hardware context switch is more difficult, the same thing can happen on tasks mapped on cells.

As these context switches are handled at run time, the application programmers can not foresee them. Therefore, the communication channels associated to a surely-mapped task have always to be ready to communicate.

A communication mechanism should be able to recognize the presence and absence of each surely mapped task on the physical platform, in order to guarantee

that the data will not be read from or sent to nowhere.

The Number of Possibly-Mapped Tasks

There are cases that the programmers expect some tasks would have certain number of instances when possible, in order to balance the performance of different parts of an application. However, since the resources of a dynamic reconfigurable platform are distributed at runtime, it is possible that not all the expected instances could be mapped to the computing units (GPPs or cells) due to the lack of resources at a specific moment. It is not fatal to the application, as long as at least one instance appears in the final physical task graph.

T3 in the motivating example is such a task. In the case of the physical task graph (A), the two expected instances in the logic task graph are mapped to a cell (T3a) and a GPP (T3b); while in the case of the physical task graph (B), the expected instance T3a does not get an implementing unit. Since the programmer is not able to and has no need to predict the usage of the resources in the dynamic reconfigurable platform when the application is written, the use of FIFO (4) and (5) becomes a problem. If the programmer does not use the two FIFOs in the application code, the computation power of T3a in the physical task graph (A) will be wasted owing to the disconnections to T1 and T5; on the contrary, the direct usage of the two data FIFOs in the application code may cause that part of output data of T1 are led to FIFO (4) and stored in there forever, which in turn makes T3b to be blocked or to run wrongly because of the missing input data.

From the T3 example, we can see that in a dynamic reconfigurable platform where the number of physical instances of a task may change, the unidirectional single-input single-output FIFO solution is difficult to manage explicitly in the application code. A communication mechanism should be developed in order to recognize and connect only the actually mapped task instances.

5.1.2.2 The Access to Tasks

Once the existence of tasks is guaranteed, the challenge of the communication mechanism is to find out the location of the tasks, and to transfer data in a proper way.

The Location of the Tasks

The locations of the tasks are important. They are the actual places where the data are produced and/or consumed. They are the sources and/or destinations where the communication system should receive data from or sent data to.

The problem brought by the dynamic reconfigurable platform is that when the application is written, the programmer cannot and has no need to predict where exactly a task will be located. In the motivating example, the fact that T3b is mapped to GPP 2 or GPP 1, and that T4 is mapped to cell 3 or cell 2, are influenced largely by the runtime status of the underlying platform.

A communication mechanism should be developed to be able to find out

the actual computing units to which the tasks are mapped, and to connect the computing units to the communication system.

The Nature of the Tasks

The data storage in software tasks usually differs from that in hardware tasks. In a software task, data are usually stored in arrays in memory, which are referenced by pointers; while in a hardware task, data are probably stored in a FIFO on the reconfigurable fabrics. As a consequence, the methods of accessing data in these two kinds of tasks should be managed differently.

Although in the last chapter, we mentioned that in our context we assumed an explicit hardware/software partition in specific applications, it is possible that the application programmers are willing to try different partitions for an application when they search for better performance balance amongst tasks. To avoid the potential complicated modifications of associated communication management in the application code, it is expected that the communication mechanism handles correctly the difference of the task nature.

For instance, in the motivating example, the task which sends data to the FIFO, called the producer, could be a software task or a hardware task (e.g. T2 for FIFO 2); so does the task which receives data from the FIFO, called the consumer (e.g. T5 for FIFO 3). The communication mechanism would become more generic if it would be able to recognize the nature of tasks and provide the proper data transfer interface accordingly.

5.2 MWMR Channel Analysis in a Dynamic Reconfigurable Context

By studying the existing communication solutions, we found a generic communication channel, called MWMR channel [Fau07]. It has some interesting features which may resolve part of the problems listed in the previous section.

At the beginning of this section, a simple description of the original MWMR channel is given. An analysis of the use of the MWMR channel in the motivated example is then carried out, distinguishing the problems remaining unsolved, which sets up the start point of our proposal.

5.2.1 MWMR Channel Description

A MWMR channel is a generic channel behaving as a FIFO, which can be accessed by multiple writers and multiple readers. Figure 5.3 illustrates the basic elements and access operations of a MWMR channel.

The data buffer and necessary control structures of the MWMR channel are located in the shared memory. Being allocated and deallocated by application programmers, the MWMR channel is a software element.

As most of the memories support only one access at a time, the MWMR channel does not support simultaneous accesses. The exclusive access amongst several

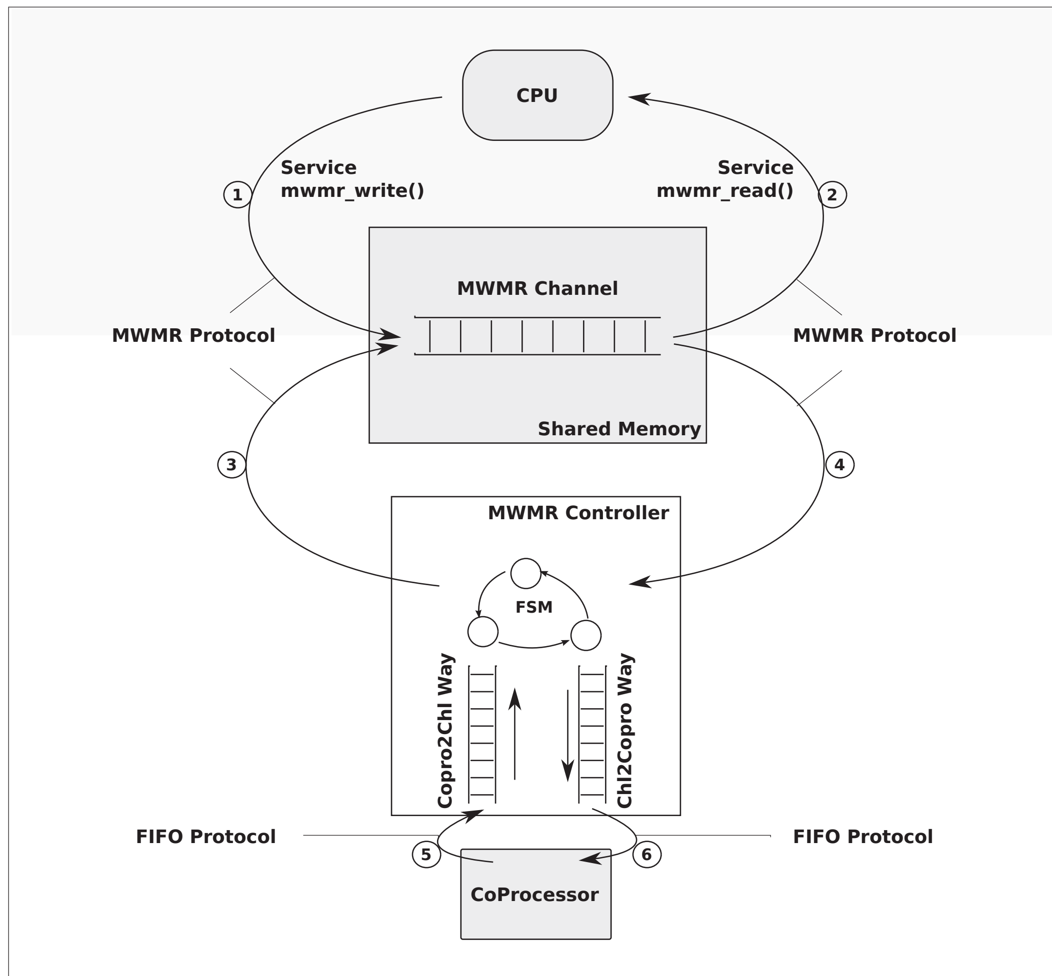


Figure 5.3: Basic Properties of a MWMR Channel

readers and writers is guaranteed by a five-step protocol described as follows:

- Get the lock protecting the MWMR channel;
- Test the status of the data FIFO of the MWMR channel;
- Transfer a burst of data between a local buffer and the data FIFO of the MWMR channel;
- Update the status of the data FIFO of the MWMR channel;
- Release the lock.

When the access is required by a software task (see top of Figure 5.3), the five-stage protocol is implemented by a user library. The two software communication services *mwmr_write* and *mwmr_read* in the library are blocking functions which can be called simultaneously by several tasks mapped to different CPUs.

When there is not enough data to be read from an MWMR channel, or there is not enough space inside an MWMR channel to receive data, the thread of the

task which calls the communication service will be blocked by the access request. In such a situation, another thread may get the CPU resource to run, if there is any. An underlying operating system is responsible to do the context switch and to synchronize the simultaneously called communication services.

When the access is required by a hardware coprocessor(see bottom of Figure 5.3), the five-stage protocol is implemented by a state machine in an MWMR hardware controller. A hardware MWMR controller can have several small unidirectional FIFOs, called ways. Each way can connect an MWMR channel with an input or output FIFO in a hardware coprocessor. A hardware coprocessor and a way communicate using the FIFO protocol, while an MWMR channel and a way communicate using the MWMR five-stage protocol.

It should be noted that in practice, an access requested by a hardware coprocessor is composed from a sequence of fixed-length data transfers. The data transfer length equals to the depth of the way. Only when a way is empty or full will a data transfer take place. The state machine inside the MWMR controller is responsible to watch over the status of a way and to realize the data transfer accordingly.

5.2.2 Why the MWMR Channel is Chosen

We have chosen the MWMR channel as a starting point to solve the four communication problems identified in section 5.1. Three major reasons are stated in the following text, explaining why such a decision is made from the hypotheses, the useful features and the technique aspects.

5.2.2.1 Shared Hypotheses

First of all, the MWMR channel shares with us certain hypotheses on the platform and the application.

The MWMR channel is developed aiming at shared-memory multi-processor system on chip, which contains several I/O hardware coprocessors. It is well adapted to our GPP/FPGA hybrid context by considering our hardware components realized on FPGA cells as coprocessors, which are linked with general purpose processors by memory-mapped I/Os.

In terms of programming model, the MWMR channel tackles two kinds of models in telecommunication applications: the pipeline model, which means splitting the algorithm into functional tasks that execute sequentially; and the task farm model, which means duplicating the application into several clones. Both of programming models extract the coarse grain parallelism from an application based on data flow, which is also congruous with our hypothesis: The application is made of coarse-grained independent tasks running in parallel, the inter-task communications contain no control relative information.

5.2.2.2 Useful Features

Secondly but not less important, the MWMR channel has already met two of the essential communication requirements required by our objective dynamic reconfigurable systems: the tolerance regarding the different nature and number of tasks.

Figure 5.4 shows the situation when we use MWMR channels to replace the simple data FIFOs as the communication solution in the motivated example.

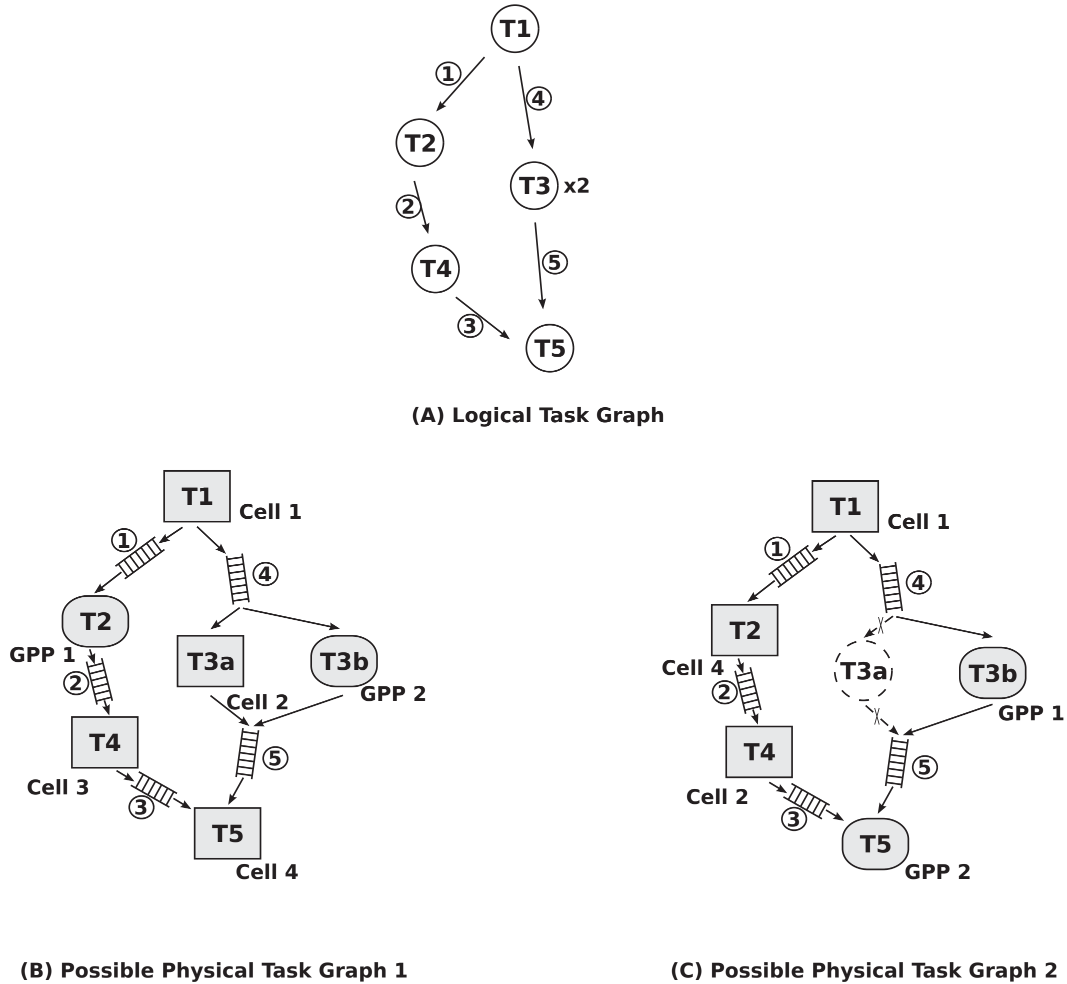


Figure 5.4: Different Physical Task Graphs Using the Same MWMR Channels

We can see that in the two physical task graphs, each link between tasks in the logical task graph is implemented by an MWMR channel. Although the task mapping in the two physical task graphs are different, the same MWMR channels can be used, forming a unique unchanged communication scheme.

Thanks to the task-nature-tolerant feature of the MWMR channel, the communication scheme accepts a data producer implemented either as a software task or as a hardware component (T2 of MWMR channel 2, for instance); the communication scheme is also agnostic towards the nature of a data consumer (e.g. T5 of MWMR channel 3 and MWMR channel 5).

Thanks to the task-number-tolerant feature of the MWMR channel, the communication scheme accepts the number of sources of a communication channel to be unique or multiple (T3 instance(s) of MWMR channel 5); so does the number of destinations of a communication channel (T3 instance(s) of MWMR channel 4).

5.2.2.3 Reasonable Technical Requirements

Finally, from the technical point of view, the MWMR channel protocol keeps its generality as it does not impose much special requirements on the implementation details. The library which allows software tasks to access the MWMR channel can be built up on a very thin hardware abstraction layer and a few basic operating system services. The hardware controller which permits hardware tasks to access the MWMR channel employs the generic virtual component interface (VCI) rather than any specific bus. The freedom from specific implementation detail makes it easy to integrate the MWMR channel in our system; also, the communication mechanism which we propose based on the MWMR channel may get a chance to receive wide acceptance.

5.2.3 The Problems Unsolved by the MWMR Channel

A MWMR channel accepts any number of writers and readers, which can be software or hardware tasks, so that various source-destination combinations of inter-task communications can be supported by a uniform communication scheme. However, the tolerance of such a communication scheme is achieved under a default condition, which is that all data sources and destinations should be known in advance by the MWMR channels. In other words, the MWMR channel itself is not responsible for detecting the existence or the location of the communicating tasks.

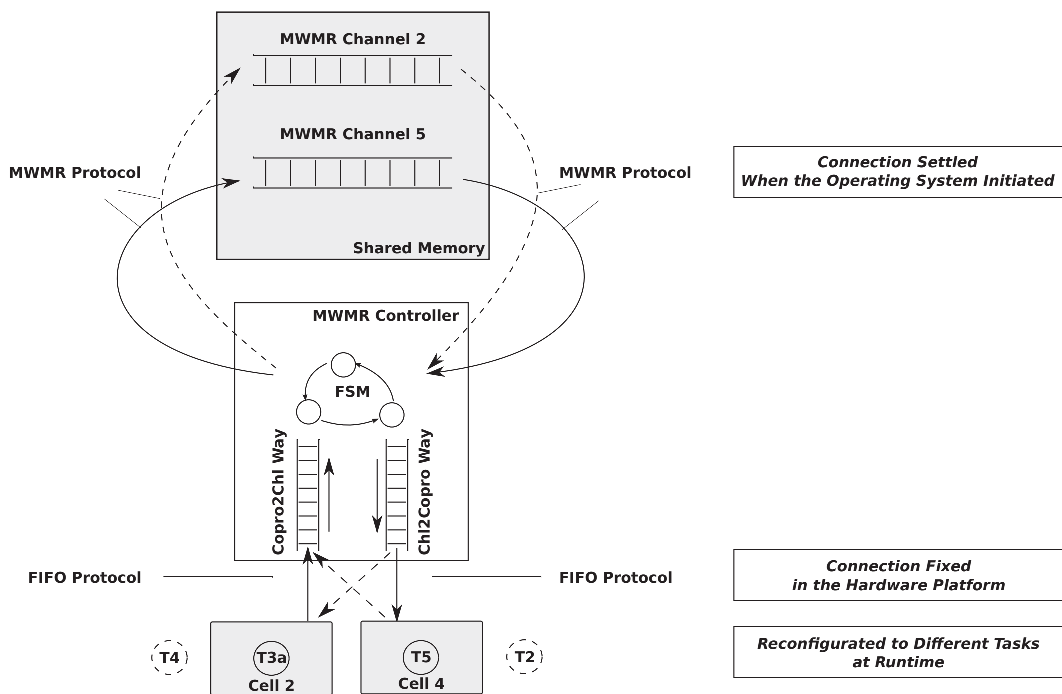
When writing the application, the programmer knows which tasks should be connected to a MWMR channel according to the logical task graph. Mapped to a dynamic reconfigurable physical platform, a software task has no problems of accessing any MWMR channel. It is the hardware tasks that suffer from connecting issues. A detailed analysis of these issues is given in the following paragraphs.

A dynamic mapped software task is able to access any MWMR channel, because, with the help of an operating system, the process (or thread) model has abstracted common characteristics of different physical CPUs. In fact, software tasks are mapped to identical processes (or threads) instead of different CPUs. On whichever physical CPU the software task is running, it is sufficient to call the *mwmr_read* and the *mwmr_write* services inside the software task. Indicating explicitly the name of the MWMR channel as a parameter, the services and the underlying operating system guarantee the exclusive access and coherence of the associated MWMR channel in the shared memory.

It is not the case for hardware tasks. From the section 5.2.1, we know that a hardware coprocessor is statically connected to a MWMR channel through ways in a MWMR controller. In the context of the original MWMR channel proposition, the hardware coprocessor has a dual-nature. It is a function which completes a specific algorithm, and at the same time, it is a computing unit which is a part of physical platform. However, in a dynamic reconfigurable platform, a function and the computing unit of this function are two different concepts. The location of a hardware task changes depending on the runtime conditions. The static connecting solution between a hardware coprocessor and a MWMR channel no longer fits for the dynamic cases.

Figure 5.5 illustrates the dynamic connection problem that the MWMR channel has not resolved. Sub-figure A gives a close view of two scenarios of inter-task communications extracted from Figure 5.4. Sub-figure B shows the underlying physical implementation of the two scenarios in the sub-figure A.

(A) Two Scenarios of Communication Between Cell 2 and Cell 4



(B) The Underlying Implementation of the Above Two Scenarios

Figure 5.5: The Problems that MWMR Channels Left Unsolved

From sub-figure A, we can see in the physical task graph 1, the two communicating tasks are T3a and T5. Data is transfered from cell 2 to cell 4 through MWMR channel 5, represented by the solid line. While in the physical task graph 2, the two communicating tasks are T2 and T4. Data is transfered from cell 4 to cell 2 through the MWMR channel 2, represented by the dotted line.

From sub-figure B, we can see that in order to realize the scenario in physical task graph 1 represented by the solid line, cell 2 is connected to a coprocessor-to-channel way in the MWMR controller, which is in turn connected to the writing end of MWMR channel 5. The reading end of MWMR channel 5 is connected to a channel-to-coprocessor way, which is in turn connected to cell 4. While in order to realize the scenario in physical task graph 2 represented by the dotted line, all con-

nections should be changed. Cell 2 should be connected to a channel-to-coprocessor way in the MWMR controller, which should be in turn connected to the reading end of MWMR channel 2. The writing end of the MWMR channel 2 should be connected to a coprocessor-to-channel way, which should be in turn connected to cell 4.

There is a potential problem. Although the tasks are dynamically mapped to different cells at runtime, which are unpredictable by the programmer, the connections between the cells and the MWMR channels are static: the links between the cells and the ways in the MWMR controller are fixed in the hardware platform; the connections between the ways and the MWMR channel are settled while the MWMR controller is being set up during the operating system initiation. As a result, the original MWMR channel alone cannot fulfill the two scenarios described above. Something additional must be integrated to form a communication mechanism able to handle the connections dynamically.

5.3 Proposed Communication Mechanism Based on MWMR Channels and HCM

From the above analysis, we can see that even with MWMR channels, there are still some problems left unsolved: the hardware task existence and location detections. In a dynamic reconfigurable platform, the HCM described in the last chapter is the component which is aware of the task mapping information. As a consequence, the basic idea of our proposal is to integrate an HCM with MWMR channels to form a unique communication mechanism able to handle cell sharing in dynamic reconfiguration. The communication mechanism allows the programmer to only deal with the logical task graph and to let underlying communication services connect all mapped tasks, whatever their nature and number are. The following sections explain in details of the hardware architecture and software services of the proposal.

5.3.1 Proposed Architecture

Figure 5.6 illustrates our proposed architecture. It is based on the targeted hardware architecture template shown on the figure 4.1, and modified to integrate an HCM and several MWMR controllers. Software tasks and a unique operating system are running on GPPs. MWMR channels are implemented in the shared memory. Hardware tasks are mapped to homogeneous cells. The reconfiguration control registers of cells and reconfiguration controllers are addressable and managed by the HCM. The communication interface of hardware tasks are not addressable FIFOs. They are instead handled by MWMR controllers.

We assume that each hardware task has an input FIFO and an output FIFO. The two FIFO interfaces for each cell are constructed during the platform initialization, and remain unchanged through the later reconfigurations. The input FIFO interface is statically connected to a channel-to-coprocessor way in an associated MWMR controller, while the output FIFO interface is statically connected to a coprocessor-to-channel way in the same MWMR controller. Although the single-input single-output FIFO interfaces assumption adds an extra constraint to hardware task de-

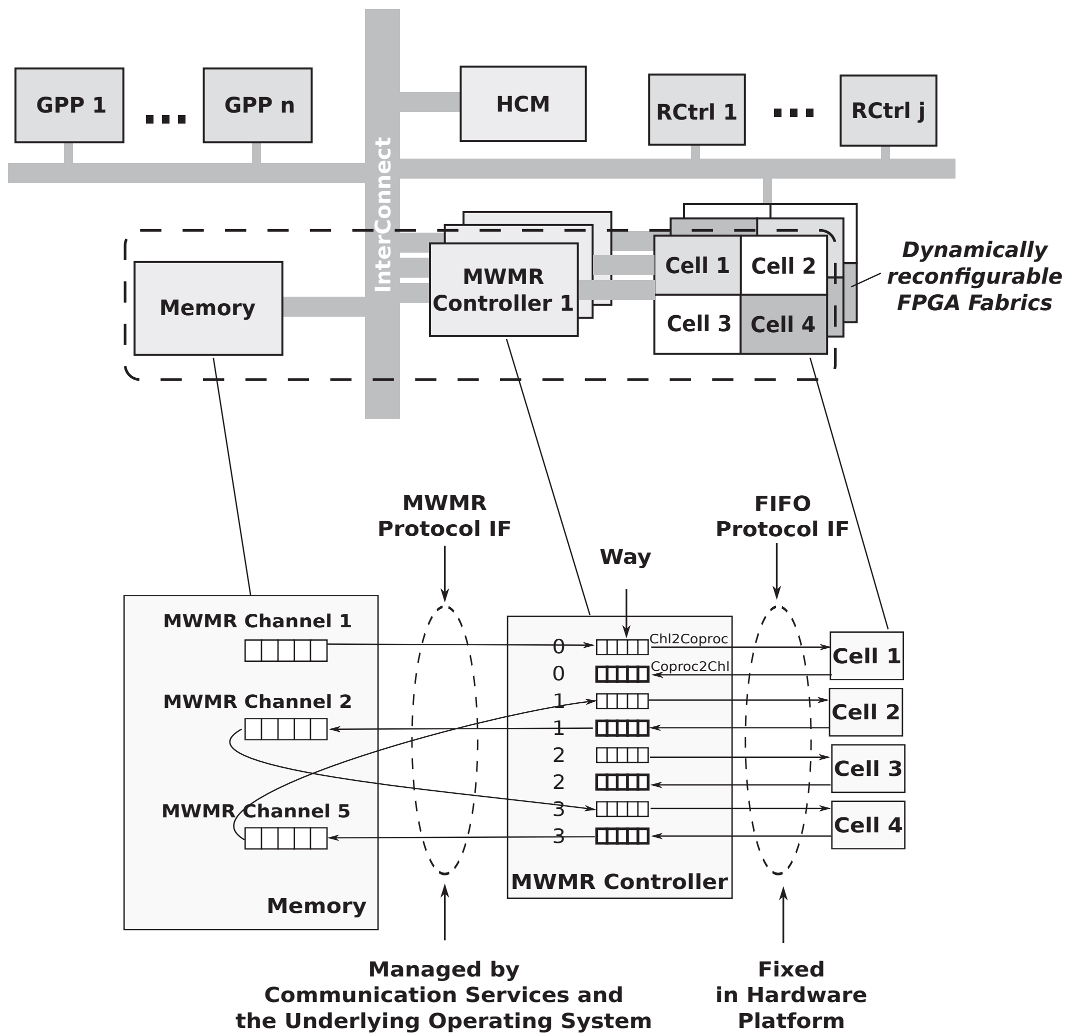


Figure 5.6: Proposed Communication Architecture

signers, it largely simplifies the integration of hardware tasks, making the communication mechanism scalable.

The communication mechanism is organized as clusters. Each cluster is an MWMR controller containing equal numbers of channel-to-coprocessor ways and coprocessor-to-channel ways. For example, in Figure 5.6, this number is four. Each pair of one channel-to-coprocessor way and one coprocessor-to-channel way is connected with the input FIFO and output FIFO of a cell.

The objective is to achieve the correct connections between the hardware tasks and the MWMR channels, in spite of the fact that hardware tasks mapped in the cells are evolving at runtime. Since the FIFO protocol interface connection between cells and ways is fixed in the hardware platform, the changeable part is the connection between the ways and the MWMR channels. Ideally, an application is independent from the underlying platform, so that none of the platform information, as for example the index of a way, should appear in the program code. The programmers indicate only the logical connection between the hardware tasks and the MWMR

channel. Some communication services are required to cooperate with an operating system for searching the hardware task mapping information from the HCM and for managing the connection of ways and MWMR channels accordingly.

5.3.2 Communication Services

In this section, we introduced five communication services, which are used to create and destroy the MWMR channels according to the need of application programmers, to connect the MWMR channels with hardware tasks at runtime and to ensure that the data will not be transfered to inactive tasks. To realize these services, some operating system support is needed.

First of all, all MWMR controllers are globally indexed, so are cells. A cluster, composed of one MWMR controller and a fixed number of associated cells, has the same global index number as the MWMR controller in the operating system. The ways connected to input and output FIFOs of the associated cell are locally indexed inside the cluster. The index information is used by the MWMR controller driver, the HCM driver and hardware component abstraction.

As mentioned in chapter 4, all hardware tasks mapped on cells are identified as hardware components in the operating system. Each time the HCM reports a successful allocation of a task on a cell, the cell index is read by the operating system and stored as a property of the hardware component.

According to the cell global index and the number of cells in each cluster, the operating system is able to locate the ways which are connected to the input and output FIFOs of the hardware task on the platform. The formulas of locating the ways are listed as follows:

1. $cl = ce/N_{cells}$
2. $way_{in} = ce \mod N_{cells}$
3. $way_{out} = ce \mod N_{cells}$

where cl is the global index of the cluster, ce is the index of the cell on which the hardware task is mapped to, N_{cells} is the number of cells in each cluster, way_{in} is the local index of the channel-to-coprocessor way which is connected to the input FIFO of the hardware task, way_{out} is the local index of the coprocessor-to-channel way which is connected to the output FIFO of the hardware task. In an MWMR controller, the channel-to-coprocessor ways and coprocessor-to-channel ways are indexed separately, so way_{in} and way_{out} of a specific cell are the same.

The location of the ways is then used by the communication service *mwmr_hw_init* to connect the ways to an existing MWMR channel created by the service *mwmr_create*. Each time a hardware task finishes its job, the communication service *mwmr_hw_cutoff* forbids the data transfer between the MWMR channel and the inactive hardware task. A disconnected MWMR channel can then be released by the service *mwmr_destroy* to free the shared memory, or can be cleaned by the service *mwmr_reset* to get ready to be reused by the connection between other tasks. We now detail the five communication services.

- ***mwmr_create***

  *void **mwmr_create**(struct mwmr_s ** channel, size_t width, size_t depth)*

  Description:

  This service is used to create an MWMR channel. In the original version of the MWMR channel proposal, all MWMR channels are created during the boot of the operating system. Thanks to this new service, an MWMR channel can be created at runtime according to the need of programmers.

  Input:

  width: the width of the MWMR channel, measured in bytes.

  depth: the depth of the data buffer of the MWMR channel. The data buffer occupies (width * depth) bytes in the memory.

  Output:

  channel: the address of the pointer which points to the structure of an MWMR channel. Before the execution of the service, the pointer has no valid value. After the execution of the service, the pointer points to the created MWMR channel structure.

- ***mwmr_hw_init***

  *void **mwmr_hw_init**(void * mwmr_controller_base, enum MwmrWay direction, size_t index, const struct mwmr_s * mwmr_channel)*

  Description:

  This service is used to connect a way to an existing MWMR channel. Once the service is done, the data can be immediately transfered between the MWMR channel and the way.

  Input:

  mwmr_controller_base: the base address of the MWMR controller, in which the way is located. This address is obtained by the combination of the cluster index (Formula 1) and the system memory map.

  direction: the direction of the way. If the way is connected with the input FIFO of the hardware task, the direction should be indicated as channel-to-coprocessor; otherwise, the direction should be indicated as coprocessor-to-channel.

  index: the local index of the way inside the cluster (Formulas 2 and 3).

  mwmr_channel: the pointer of the MWMR channel to connect.

  Output: None.

- ***mwmr_hw_cutoff***

  *void **mwmr_hw_cutoff**(void * mwmr_controller_base, enum MwmrWay direction, size_t index)*

  Description:

This service is used to disconnect a way from the MWMR channel it is linked to. Once the service is done, the data transfer between the MWMR channel and the way is immediately cut off.

Input:

mwmr_controller_base: the base address of the MWMR controller.

direction: the direction of the way. Since the coprocessor-to-channel way and the channel-to-coprocessor way connected to a hardware task have the same index number in both direction categories, the indication of way direction is important for the MWMR controller to recognize the way to be cut off.

index: the local index of the way inside the cluster.

Output: None.

- ***mwmr_reset***

  *void **mwmr_reset**(struct mwmr_s * channel)*

  Description:

  This service is used to reset an MWMR channel. Once this service is done, the status of the data buffer is forced to reset as unused and as if no read or write action had ever happened. At the same time, the lock of the data buffer is released. If there are data left in the buffer, they will be lost. So before using this service, the programmer should make sure that the required data have been retrieved.

  Input:

  channel: the MWMR channel to be reset.

  Output: None.

- ***mwmr_destroy***

  *void **mwmr_destroy**(struct mwmr_s ** channel)*

  Description:

  This service is used to destroy an MWMR channel. All the memory occupied by the channel are cleaned and released after the execution of this service.

  Input:

  channel: the address of the pointer which points to the MWMR channel to be destroyed.

  Output: None.

5.3.3 Use on the Motivating Example

With the proposed architecture and the new added communication services, the motivating example can be solved. The following algorithm is written according to the platform usage of sub-figure (C) of Figure 5.4.

1. Creating all five MWMR channels chl1 - chl5 by the service *mwmr_create*;

2. Mapping tasks T1, T2, T3a, T4 to cells and tasks T3b, T5 to GPPs.
3. Calling the service *mwmr_read* and *mwmr_write* inside T3b and T5 to accomplish the data transfer. Once the mapping is successful, linking T1 with chl1, linking T2 to chl2, linking T3a with chl4 and chl5, and linking T4 with chl3 by the service *mwmr_hw_init*;
4. When hardware tasks finish their job, disconnecting them from the corresponding MWMR channel by the service *mwmr_hw_cutoff*; Since T3a has never been mapped, only T1, T2 and T4 process this step.
5. Destroying T1, T2 and T4.
6. When all the tasks finish their job, release the MWMR channels chl1 - chl5 with the service *mwmr_destroy*.

5.4 summary of this chapter

In this chapter, we have seen four inter-task communication problems brought by the dynamic reconfigurable platform. Depending on the platform usage, the surely-mapped tasks may be switched in-and-out; some of the possibly-mapped tasks may not be present at all. Once mapped, the connection of the tasks require their precise locations; the access method of a task asks for the recognition of the nature of the task. All the above four issues are runtime issues and cannot be predicted by a programmer at the time an application is written.

Based on the existing MWMR communication channel solution which provides the required tolerance regarding the nature and number of tasks, we propose a communication cluster architecture composed of MWMR controllers and cells. In addition, five communication services are presented to support the life cycle of these new channels, so that all the runtime needs of programmers are fulfilled.

Chapter 6

Experiments

In this chapter, we prove experimentally that the HCM and the scalable communication mechanism are efficient for managing dynamic reconfiguration platforms.

The content of this chapter is organized as follows: Section 6.1 shows the interest of integrating the HCM into an OS. A hardware implementation of the HCM and its integration in DnaOS are presented as an example. The hardware overhead and integration cost are analyzed. Section 6.2 presents the validation of the communication mechanism. An image processing application confirms that the communication mechanism combined with the integrated HCM have saved much work of the programmers.

In chapter 4, we have introduced the HCM abstraction layer, which separated task allocation from the FPGA reconfiguration procedure. This separation removes the burden of reconfigurable resource management from application programmers. At the same time, the flexibility of the application has been improved.

The HCM is designed to be able to work in multi-task environments, even multi-user environments. However, the synchronization with the HCM and the processing of resource lacking situations are tedious jobs for application programmers. Sometimes, processing an unexpected allocation acknowledgement or sharing other platform resources rather than FPGAs is beyond the control scope of an individual user. Luckily, the above task management problems are well-studied subjects in the OS domain. From the next sections, we will see that the HCM can be easily integrated into an OS at a reasonable cost, which in turn completely solves the task management problems in a dynamic reconfigurable FPGA/GPP hybrid platform.

6.1 Proof-of-concept Integration in an OS

For the proof-of-concept integration, we chose DnaOS [GP09, dna10] to prototype our solution. DnaOs is a kernel-mode operating system for heterogeneous multiple-processor architectures, built on a very thin *Hardware Abstraction Layer* (HAL). Its well defined *application programming interface* (API) provides the support of the most widespread application libraries such as a fully fledged C library or *pthreads* library. Thanks to its strict separation from the hardware dependent software, DnaOs can be ported easily through different platforms and processor architectures. In addition,

the fact that DnaOS is open-source and built up using the object-component model paradigm makes it easy to tailor it to our need.

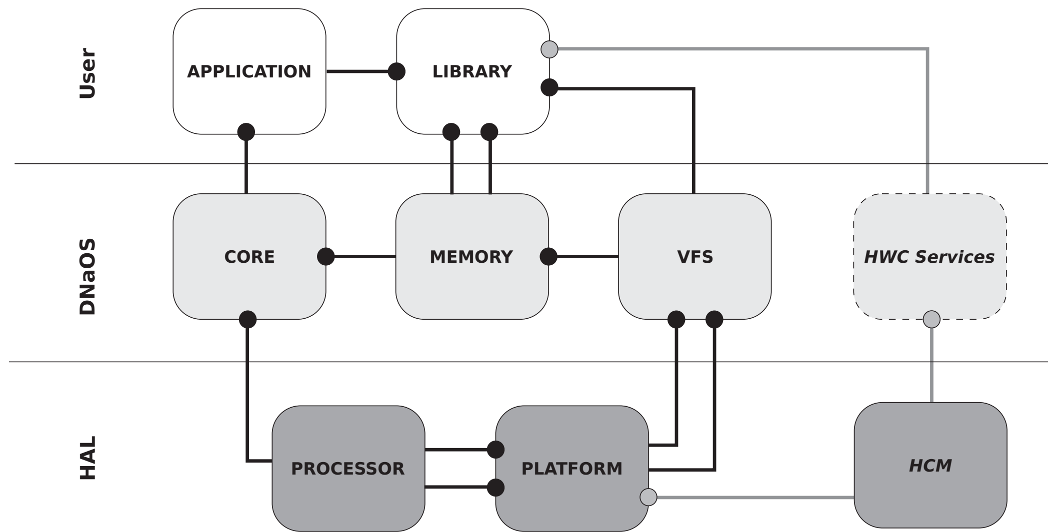


Figure 6.1: Global View of Software Organization

Figure 6.1 shows the global view of the organization of various software components running on the dynamic reconfigurable GPP/FPGA hybrid platform. The software components are organized in three layers: user application and libraries, the OS and the HAL. The HAL is the lowest layer, abstracting the functionalities of the physical platform. Upon the HAL, the OS is constructed, which in turn is providing necessary services to ease the life of the users.

In chapter 4, the HCM was presented as an abstraction layer of all reconfigurable resources on the platform, which can be considered at the same level as the HAL in the Figure 6.1. Also, some common operations were recognized during the lifetime of any hardware components. We have discussed that it would be efficient to encapsulate them in a user library. Between the HCM and the hardware component library, some additional OS supports (presented as "HWC Services" in the dashed-line block) are required to provide a uniform API to programmers, and to ease the use of the HCM. One possible implementation is described in the following text.

6.1.1 The Implementation of the HCM Integration

Figure 6.2 shows the infrastructure of the software supports and interactions amongst the different parts. The software support is achieved mainly by three elements: an *Hardware Accelerator* (HA) library at user layer, the *HardWare Component* (HWC) extension at the OS layer and an HCM driver at the HAL layer.

The HA library is the implementation of the pre-synthesised hardware accelerator library described in section 4.1.2. Each object in the library is a data structure, which contains the FUNCTION_ID of the HA, the size of the corresponding bitstream and the base address in the memory where the bitstream is stored. Thanks to the homogeneous cell choice, all hardware components realizing the same func-

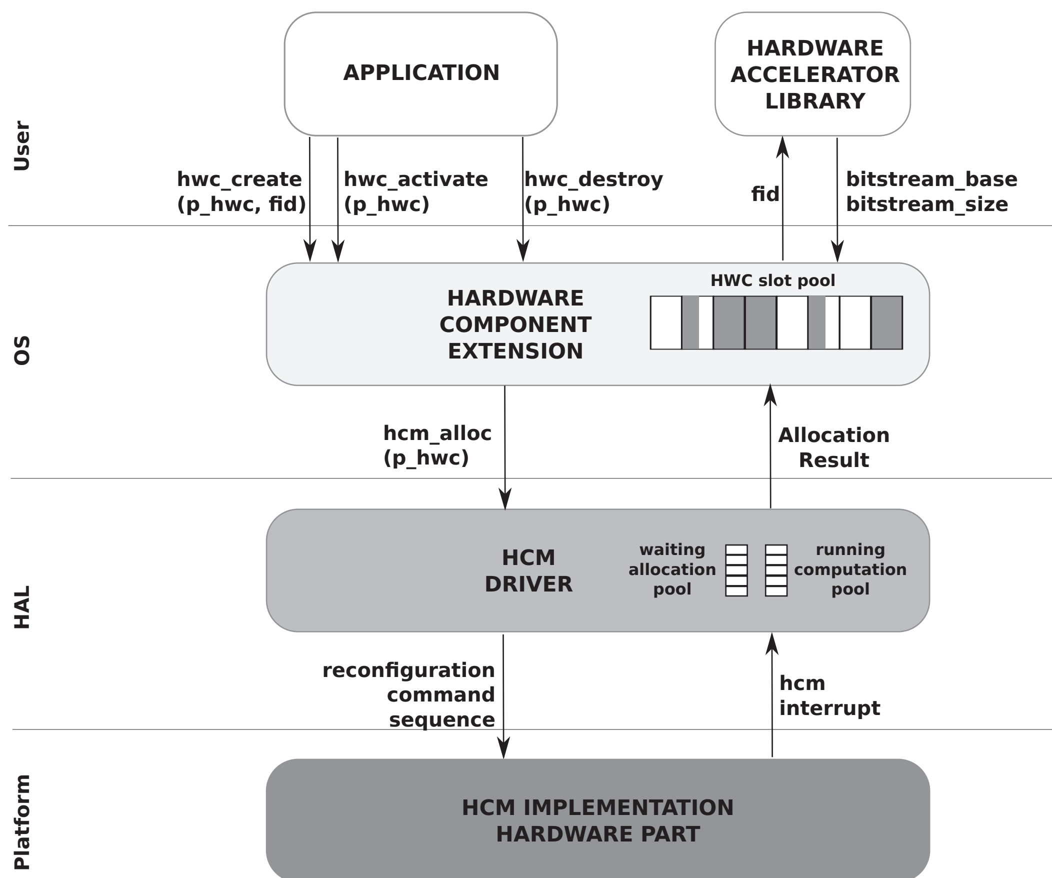


Figure 6.2: One Implementation of HCM-integrated OS

tion can use the same HA library object to search the configuration bitstream. The number and kinds of HA in the library is initiated at the OS boot phase.

The HWC extension provides a uniform API to applications. The API is formed by three HWC services, called *hwc_create*, *hwc_activate* and *hwc_destroy*. The detailed description of these three services is given in the next section. The most important part of the HWC extension is an HWC slot pool. Each slot in the pool is a data structure containing all properties of an hardware component (see Table 4.1). By maintaining the HWC slot pool, the OS is able to supervise all existing hardware components, to make them correctly synchronize with applications and to share the HCM properly.

The HCM driver chiefly communicates with the HCM implementation hardware part. It receives the `hcm_alloc` request formed by the HWC extension and sends the corresponding sequence to the HCM. It serves the interrupts coming from the HCM implementation hardware part, while maintaining two pools. The waiting allocation pool keeps the pointer of HWC slots, whose allocation sequences have been sent, but not yet on the cell. The running computation pool keeps the pointer of HWC slots, which has not yet finish the required function.

The three elements interact, recording the current status of the evolving platform and deciding upon the correct action to take. To make it clearer, a detailed description of HWC services is given below, with the explanation of service internal

processing procedure as well.

6.1.2 HWC Services Description

- ***hwc_create***

  *status_t **hwc_create**(hw_component_t * hw_component, hwc_function_id_t fid);*

  Description:

  This service is used to create a hardware component to perform the function *fid*. It works in a non-blocking way.

  The service gets a slot from the hardware component slot pool, dedicating the slot to the required hardware component; also, it uses the function identification *fid* as the index to look up the HA library, in order to collect the corresponding `bitstream_base` and `bitstream_size` as necessary information to form the reconfiguration command sequence. The information is stored in a slot as hardware component properties; finally, an allocation request ***hcm_alloc*** is sent to the HCM, using the hardware component slot as a parameter. The service checks the result of the allocation request, processing according to different situations and reporting to the user.

  Input:

  hw_component: an empty hardware component pointer;

  fid: the function identification that the hardware component will perform;

  Status:

  If the return value is `HWC_ALLOC_OK`, it means that *hw_component* is pointing to a valid hardware component slot and that the reconfiguration command sequence has been successfully sent to the dynamic resource manager. Otherwise, the creation fails.

- ***hwc_activate***

  *status_t **hwc_activate**(hw_component_t hw_component);*

  Description:

  This service is mainly responsible for the synchronization of the hardware part and the software part of the hardware component, which has already been created by a call to ***hwc_create***. It works in a blocking way.

  In this service, the hardware component status is checked up for once. At the moment of checking, if the reconfiguration of the underlying cell is complete, the service initiates the hardware component which has the ready status and the software environment related with the hardware component. After that, the hardware component status is changed to `computing` (cf. Table 4.1). The thread calling the services releases the CPU to let other threads have a chance to run, while the hardware component is doing its work on the platform.

  Otherwise, the service simply changes the hardware component status to `wait_cfig_finish` and yields the CPU, leaving the initialization work to the

hcm_isr interrupt service routine when the `CFIG_FINISH` interrupt is received, so as to avoid the constant polling by the general purpose processor.

In both cases, the service can only be rescheduled once the hardware component finishes its computation.

Input:

hw_component: the pointer pointing to the slot of hardware component which is about to be activated.

Status:

If the return value is `HWC_ACTIVATE_OK`, it means the hardware component has successfully completed its work on the platform. Otherwise, it means the hardware component suffers from synchronization problems during the initialization phase or the computation phase.

- ***hwc_destroy***

  *status_t **hwc_destroy**(hw_component_t hw_component);*

  Description:

  This service is used to destroy a hardware component. It works in a non-blocking way.

  One thing to notice is that only when the hardware component status is `calc_finish` can the hardware component be safely destroyed. The service verifies this condition and then cleans all the fields in the hardware component slot pointed by *hw_component*, and releases the slot back into the hardware component slot pool.

  Input:

  hw_component: the pointer pointing to the slot of hardware component which is about to be destroyed.

  Status:

  If the return value is `HWC_DESTROY_OK`, it means the hardware component has been safely destroyed. Otherwise, the destruction has failed.

6.1.3 Experiment 1: Feature Validation on Simulator-based Test Environment

6.1.3.1 Test Platforms

In order to validate the efficiency of the HCM integration, we designed test platforms respecting the template described in Figure 4.1 using the open platform SoCLib [soc10], which is a library of SystemC simulation models for IP cores.

The advantage of a SystemC simulation model over a FPGA prototype are: (1) the HCM implementation is not locked to any reconfigurable technology; (2) the parameter of platforms can be more easily changed and the HW/SW cosimulation is much easier and faster to debug.

In addition to the cycle accurate HCM model described in section 4.3.3.2, a cell model and a reconfiguration controller model are developed. The modeling of reconfiguration processes is implemented using the Dynamic Circuit Switching approach [LS96]. The cell model integrates all possible hardware components, but only one component can be active at a time. The selection of the active component is controlled through a register of the cell, which is writable only from a reconfiguration controller model. The reconfiguration controller model generates an interruption when a reconfiguration is finished. The implemented models can be parametrized at reconfiguration time for the reconfiguration controller model and execution time for cell model.

Test platforms also integrate a MIPS model for CPU cores and a mesh network on-chip based on the VCI standard for the interconnect.

Table 6.1: Parameters of test platforms

Platform	CPU Number	Cell Number	Reconf. Controller Number	with HCM
wo_hcm	1	3	1	no
base_hcm	1	3	1	yes
hcm_multi_rctrl	1	3	3	yes
hcm_multi_cores	3	3	1	yes
hcm_multi_rctrl_cores	3	3	3	yes
hcm_single_cell	1	1	1	yes

As described in Table 6.1, experiments are conducted on six platforms, defined by the four following parameters: the number of CPU cores, the number of cells, the number of reconfiguration controllers, and the presence or not of a HCM.

6.1.3.2 Application

In order to provide realistic test scenarios, simple image processing applications have been designed based on four possible components, represented in Table 6.2. The three generated applications are made of two components each, as represented on Figure 6.3. Communication between components is performed through software FIFOs in the applications, and through hardware FIFO in the component.

Table 6.2: Basic components for test applications

Component fid	Function	$T_r(\mu s)$	$T_c(\mu s)$
A	color_transform	147.6	844.86
B	gaussian_blur	147.6	99072.10
C	median_filter	147.6	172800.10
D	gradient_direction	147.6	20544.06

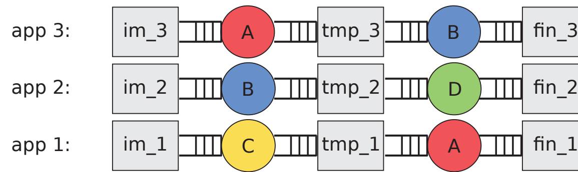


Figure 6.3: Simple image processing scenario

Reconfiguration and computation time (resp. T_r and T_c) for the proposed components are computed for a Xilinx Virtex-5 FPGA (LX110T). Targeting a cell size of 360 configuration frames [Xil11] and a 100 MHz clock, the developed components are synthesized using the UGH HLS tool [APDG05]. From the obtained hardware components, execution times are derived for 320x240 pixels images with the assumption that the communication time is overlapped. Reconfiguration time is derived from cell size assuming the reconfiguration controller is an Internal *Configuration Access Port* (ICAP) with a 32-bit wide and a 100MHz interface. The feasibility of such a controller has been presented for example in [DML11].

6.1.3.3 Results Analysis

Results are presented in Figure 6.4, Figure 6.5 and Figure 6.6. Figures 6.4 and 6.5 show the application code with and without HCM integration, while Figure 6.6 presents the resulting cell usage for each platform during execution. When looking at the **wo_hcm** platform required code in Figure 6.4, we can see that it is clearly platform dependant, not flexible, and not easy to use. The programmer needs to schedule the cell usage himself, and to explicitly call the reconfiguration procedures. He also has to manage configuration bitstreams, and the reconfiguration delay. Hardware components are implicitly used, hidden by the use of cells. It is also very error-prone, since if the programmer makes a mistake on the cell to be reconfigured, the whole application will be faulty. The three applications also need to be merged as one, which limits a parallel usage of the CPU. This could be avoided by associating one cell for each application, but it would be suboptimal.

The code designed for this platform is also non-portable. The slightest change in the number of cells or available reconfiguration controllers would mean a complete rescheduling, and thus a complete rewriting, of the application.

In HCM integrated platforms, the development of applications does not require FPGA knowledge. As can be seen in the code in Figure 6.5, all applications are wrapped in threads, which are launched in a natural order. The application is completely independent from the platform details, and only the functionality of the required hardware component is known, not its location nor implementation details. This has several advantages. First of all, the resulting applications are more flexible. They can be reused without modifications on other different platforms, moving the burden of reconfiguration management out of the scope of application developers. Cell usage results presented in Figure 6.6 show that for all platforms, the HCM performs all component allocations in a reasonable time (about 0.11% more compared to the wo_hcm platform). For the reference HCM platform, the schedule is slightly less efficient than the platform without the HCM, since the handmade scheduling of

```
1  void main() {
2          cfig(C, CELL1);
3          cfig(B, CELL2);
4          cfig(A, CELL3);
5          // Compute first component of app1
6          while(cell1_status != CFIG_FINISH);
7          compute_app1(CELL1);
8          // same for app2 and app3
9          while(cell2_status != CFIG_FINISH);
10         compute_app2(CELL2);
11         while(cell3_status != CFIG_FINISH);
12         compute_app3(CELL3);
13         // reconfigure cell3 with component B
14         while(cell3_status != CALC_FINISH);
15         cfig(B, CELL3);
16         while(cell3_status != CFIG_FINISH);
17         compute_app3(CELL3);
18         // reconfigure cell2 with component D
19         while(cell2_status != CALC_FINISH);
20         cfig(D, CELL2);
21         while(cell2_status != CFIG_FINISH);
22         compute_app2(CELL2);
23         // when component B is done, app3 is done
24         while(cell3_status != CALC_FINISH);
25         //reconfigure cell3 with component A
26         cfig(A, CELL3);
27         while(cell3_status != CFIG_FINISH);
28         // Wait for first component of app3
29         while(cell1_status != CALC_FINISH);
30         // Finishing app1 on cell3
31         compute_app1(CELL3);
32         // Wait for all apps
33         while(cell2_status != CALC_FINISH)
34         while(cell3_status != CALC_FINISH)
35 }
```

Figure 6.4: Code without HCM

allocation calls performed in Figure 6.4 is better than the unoptimized scheduling of allocation calls used for platforms with the HCM. On the other hand, the flexibility gain is clearly visible. Without changing the application, it can make use of multiple reconfiguration controllers, it can run under a multi-CPU environment, and it can still compute in a single cell platform.

```
1  void app1_proc() {
2          hw_component_t A, C;
3          hwc_function_id_t fidA, fidC;
4
5          // Allocate and compute C
6          hwc_create(&C, fidC);
7          hwc_activate(C);
8
9          // Allocate and compute A
10         hwc_create(&A, fidA);
11         hwc_activate(A);
12
13         // Release the hardware component slot
14         hwc_destroy(A);
15         hwc_destroy(C);
16 }
17 // same for app2 and app3
```

Figure 6.5: Code with HCM

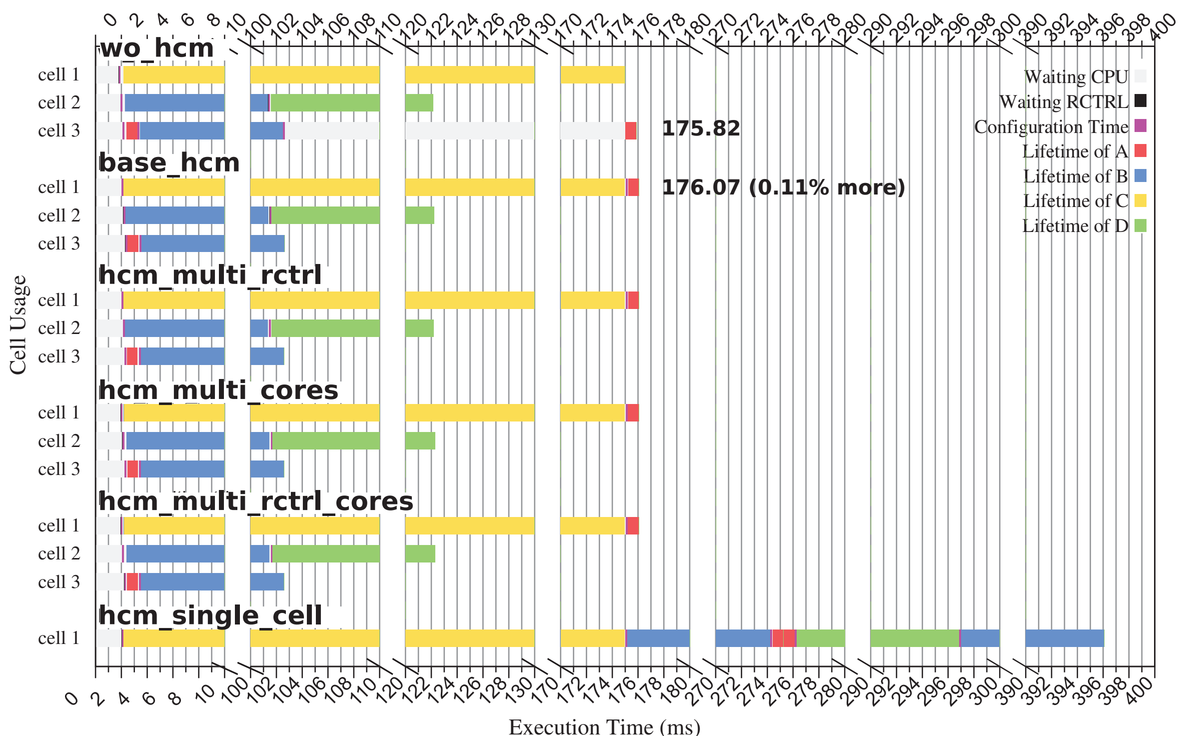


Figure 6.6: The Underneath Processing of Scenario in Different Platforms

6.1.4 Experiment 2: Integration Cost

In the last section, we have shown that the integration of a HCM enables the programmer to write flexible and elegant application code, while only adding a very small portion of execution time. In this section, we are going to measure precisely

the cost of gaining such advantages, in terms of time overhead, memory footprint and reconfigurable hardware overhead.

6.1.4.1 Test Platform and Application

In order to remove the influence of the synchronization amongst multi-GPP, and to get the worst case result, we use the **base_hcm** platform of Table 6.1 as the test platform of measuring the integration cost.

In theory, the overhead of the HCM integration is independent from the number and kind of hardware components that the HCM integration handles. In a lifetime period of one hardware component, the overhead of all elements of the extended software supports can be measured. This overhead can also represent the extended software support overhead in the lifetime of any other hardware component. Keeping this in mind, we have written an application in which component A in Table 6.2 is allocated to the reconfigurable fabric twice, in order to measure the overhead in both cases when an actual reconfiguration takes place and when the configuration in the cell is reused.

6.1.4.2 Execution Result

Figure 6.7 shows the resulting time consumption of different elements during a lifetime period of a hardware component in two cases: A) when an actual reconfiguration is needed, and B) when the hardware component reuses the configuration which is already on the cell.

In each case, there are four classes of time consumption which are measured. The bar on the top represents the time quota of involved services; the bar below stands for the time consumed by the HCM driver; the third bar illustrates the time portion of different hardware component states recognized by the OS extension; the bar at the bottom shows the time points of some important events taking place on the platform. During each lifetime period, 21 moments are sampled to measure the time overhead.

When an application needs a hardware component which does not yet exist in its scope, the service *hwc_create* is called (time sample 1 (t1) in the Figure 6.7). The FUNCTION_ID of the required hardware component is passed by the application as an input.

To begin its work, the *hwc_create* service gets an empty hardware component slot for recording information of the required hardware component. After initiation, the pointer of the slot is added into a waiting-allocation pool. Afterwards, information such as the location and size of the corresponding bitstream is collected from the HA library by *hwc_create*, in order to form an allocation request *hcm_alloc*. This request is then handled by the HCM driver (t2) and sent to the HCM implementation hardware part (t3). Since *hwc_create* works in a non-blocking manner, it terminates (t5) once the HCM driver has returned (t4) from request sending.

Then the application calls the service *hwc_activate* (t6) to ensure the presence and valid access to the HA. If the HA is not yet available, as shown in the reconfiguration case (t7), *hwc_activate* changes the hardware component state in the slot as `waiting_cfig_finish`. Otherwise, as shown in the reuse case (t11.b), *hwc_activate*

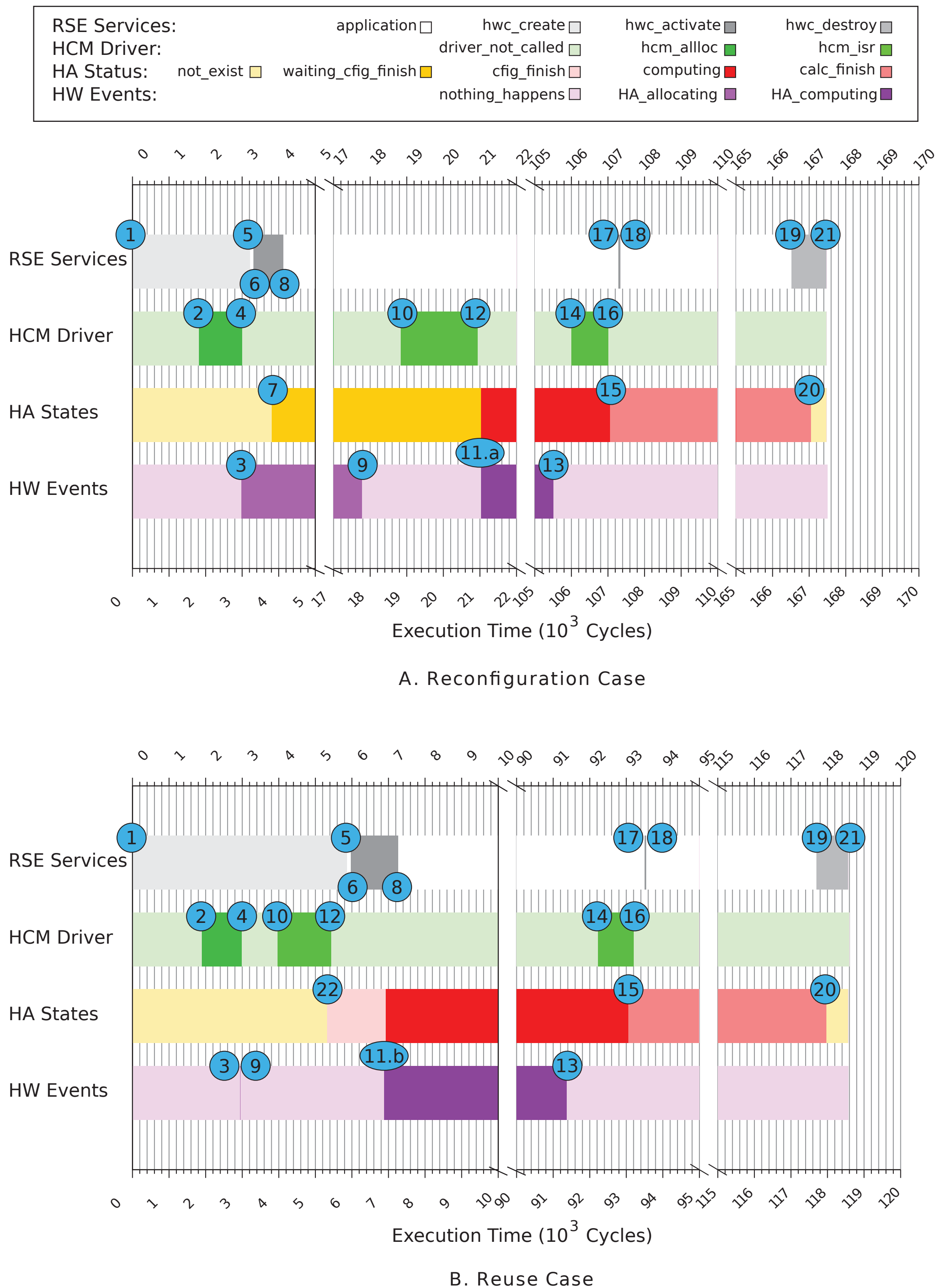


Figure 6.7: Time Consumption During the Lifetime of an HA

changes the hardware component state in the slot to `computing` and enables the hardware component function on FPGA. In both case, the *hwc_activate* service yields (t8) the CPU to other processes ready to run, if there is any.

Once the required hardware component is made available by an actual reconfiguration on FPGA (as shown in Figure 6.7.A), or by a reuse (as shown in Figure 6.7.B) according to the decision of the HCM, an interrupt is issued to report the success of the allocation (t9). This interrupt is picked up and processed by the interrupt service routine *hcm_isr* inside the HCM driver (t10 to t12). If the process had been blocked by the absence of hardware component, as shown in the reconfiguration case (t11.a), the hardware component state is changed to `computing` and the hardware component function on FPGA is enabled. Otherwise, as shown in the reuse case (t22), the hardware component state is marked as `cfig_finish`. In both case, the slot pointer of the required hardware component is extracted from the waiting-allocation queue, according to the `FUNCTION_ID` read back from the HCM implementation hardware part. The identification of the occupied cell (`BOUNDED_CELL_ID`) is also fetched from the HCM implementation hardware part. Based on the `BOUNDED_CELL_ID`, the access method of the hardware component is set up and registered into the slot. Then the hardware component slot pointer is added into the running-computation pool.

An interrupt issued by the HCM implementation hardware part reports the computation end of a hardware component (t13). This interrupt is also picked up and processed by *hcm_isr* (t14 to t16). The identification of the interrupting cell is read back from the HCM implementation hardware part. The hardware component slot with the same `BOUNDED_CELL_ID` is removed from the running-computation pool. The hardware component access method is unvalidated and its status is changed to `calc_finish` (t15). Then the yielded *hwc_activate* gets back (t17) to run until its end (t18).

At last, application calls the *hwc_destroy* service (t19) to deallocate the hardware component. The hardware component status is reset to `not_exist` (t20). *hwc_destroy* returns (t21) after the hardware component slot is cleaned and released.

6.1.4.3 Results Analysis

1. Time overhead:

   The time consumption of a reconfigurable computation (T_{rc}) is composed of computation time (T_c) and the allocation time (T_a) which is spent on getting the computing resource. In our case, the T_a can be further divided into two parts: the time spent on the HCM assisted reconfiguration on FPGA (T_r), and the time spent on managing the hardware component life cycle, the synchronization between application and the hardware component, and parallelism among hardware components. The latter is what we are going to measure: the overhead introduced by our extended OS support (T_{ov}). Using the samples from Figure 6.7 these timing are:

   T_{rc} = T[1,18] + T[19,21];

   T_c = T[11,13];

   T_r = T[3,9];

   $T_{ov} = T_a - T_r = T_{rc} - T_c - T_r$

   = T[1,3] + T[9,11] + T[13,18] + T[19,21];

where T[i,j] is the time between the sample ti and tj in figure 6.7.

The meaning of each time range is listed as follows. T[1,3] is from the moment when the *hwc_create* service is called by application to the moment when the allocation request is received by the HCM implementation hardware part. T[9,11] is DnaOS interrupt management (IRQ selection, context switch) plus *hcm_isr* for `cfig_finish`. For the reuse case, it also includes part of *hwc_activate* service. T[13,18] is DnaOS interrupt management, *hcm_isr* for `calc_finish` and reschedule of the end of the *hwc_activate* service. T[19,21] is the execution time of *hwc_destroy* service.

Table 6.3: Time Overhead of OS Extension Services

Time Range	Services	Reconfigure (cycles)	Reuse (cycles)
T[1,3]	hwc_create	2985	2945
T[9,11]		3330	3790
	DnaOS IRQ management	1230	1440
	hcm_isr	2100	1470
	hwc_activate	-	880
T[13,18]		2125	2265
	DnaOS IRQ management	1015	1230
	hcm_isr	1020	990
	hwc_activate	55	45
T[19,21]	hwc_destroy	960	870
Total Time Overhead (T_{ov})		9440	9870

The time overhead in the reconfiguration case and the reuse case are listed in Table 6.3, measured in cycles. The T_{ov} in the reuse case is slightly more than that in the reconfigure case, because of parts of *hwc_activate* service, which is concurrent with reconfiguration procedure on FPGA in the other case. In both cases, the T_{ov} stays stably less than 10k cycles. In our experiment, the cell size is relatively small (360 frames, 1600 LUTs, 1600 FFs), the time overhead is slightly less than the reconfiguration time ($T_{ov}/T_r \sim 0.65$). When the cell size increases, T_r will increase accordingly, the rate of T_{ov} in T_a will become neglected.

The developers of reconfigurable computing always wish to have a high rate of T_c/T_{rc}. Our hardware assisted OS support permits the quantitative analysis at early stage, since the T_{ov} stays almost fixed. Given the cell size and computation time of an hardware component, the developers can quickly decide whether it is worth using the hardware component at the rate of $T_c/(T_c + T_r + T_{ov})$. For example, in our experiment T_r is 14760 cycles, T_c of the worst case is about 85 thousand cycles, T_c of the best case is about 17.3 million cycles. The T_c/T_{rc} rate ranges from 77.4% to 99.8%. Once decided, then the hardware assisted OS support allows the developers to easily integrate their hardware components.

2. Footprint

Table 6.4: Memory Footprints of Systems

	Memory Footprint (KBytes)
OS extension	7.1
HWC services	2.7
HCM driver	4.4
DnaOS (Kernel, File System, drivers)	[10,40]
Bitstream (360 frames)	58
medium size application (MJPEG)	140

The footprint of the proposed extension is around 7.1KB on a MIPS processor. The Table 6.4 provides details on these components and comparison points. The obtained footprint is smaller than the original DNAOS one, which may vary between 10 and 40KB depending on OS tailoring. Note that the footprint of the OS with the extension (less than 47 KB) is always smaller than a single partial bitstream for a small cell (360 configuration frames). By considering that an application requires several bitstreams and also the application sofware footprint (e.g. an MJPEG application requires around 140KB), we can conclude that the proposed extension has a negligible impact on the overall system footprint. To the best of our knowledge, any memory footprint of existing OS extension supporting the DPR has been found in the literature.

3. Hardware overhead of the HCM

Besides the SystemC simulation model, we also implemented the HCM on a Xilinx FPGA, in the expectation of giving out a realistic reference of resource usage of the HCM. The targetted FPGA is a Virtex-5 (LX110T), which contains 17,280 slices with four Flip-flops (FF) and four look-up tables (LUT) in each slice. The following two figures are obtained through logic synthesis with a 100MHz timing constraint.

Figure 6.8 shows the curve of maximum frequency that the HCM is able to reach as a function of the number of cells. We can see when the number of cells inferior to five, the maximum frequency remains stable at about 230 MHz. Starting from five cells, the maximum frequency drops when the number of cells increases. In spite of this downward trend, the maximum frequency stays greater than 130 MHz, when the number of cells is less than 16.

Figure 6.9 is the curve diagram of the HCM resource utilization in different cell number conditions. The horizontal axis represents the number of cells that the HCM manages. The vertical axis represents the number of resources used by the HCM. There are two curves of resource utilization. The red one represents the number of FFs used by the HCM, while the green one represents the number of LUTs dedicated to the HCM.

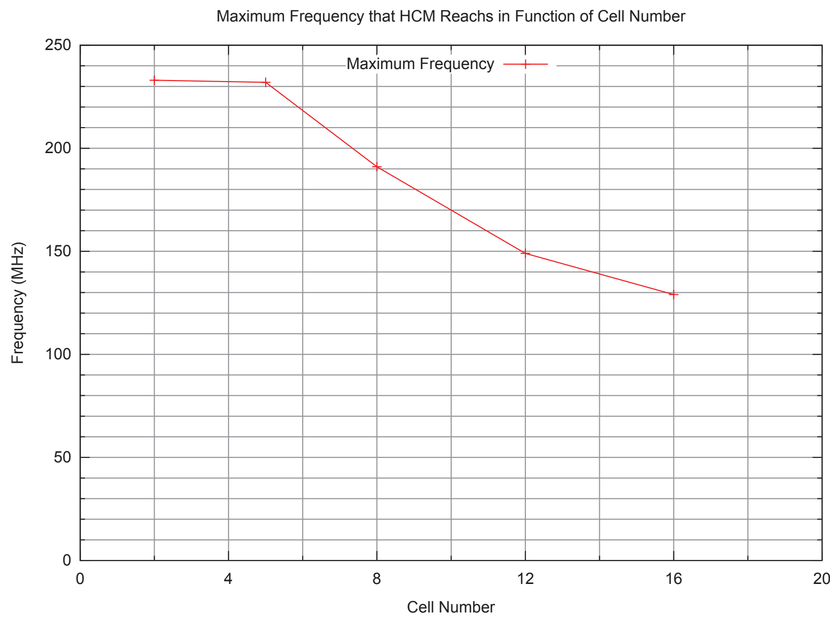


Figure 6.8: Maximum Frequency that the HCM can Reach Depending on the Cell Number

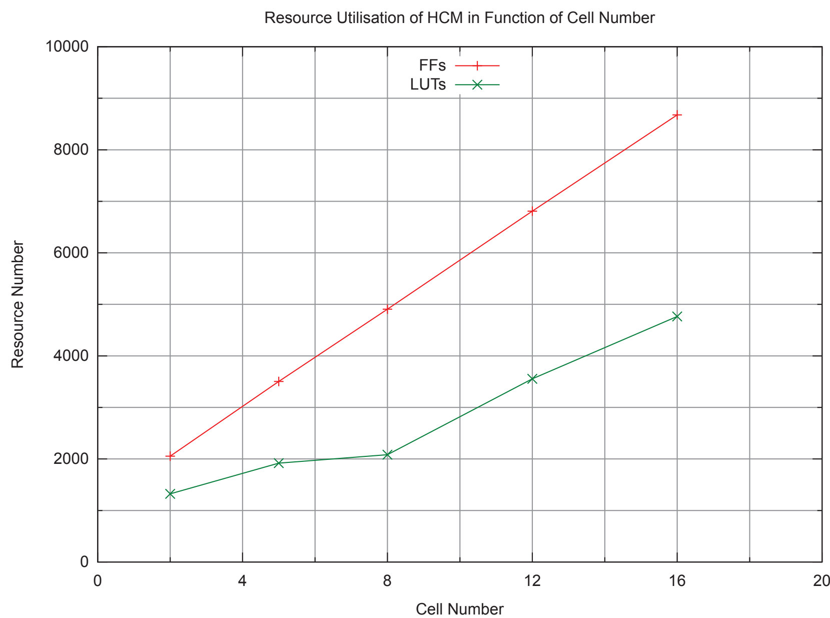


Figure 6.9: HCM Resource Utilization Depending on the Cell Number

We can see both curves raising when the number of cells increases. The number of FFs is almost proportional to the number of cells. At any number of cells, the HCM uses always more FFs than LUTs. The curves show such upward trends, because the *Cell Track Maintainer* (CTM) inside the HCM has been implemented as a *Content Accessible Memory* (CAM). That is to say, the comparison between the index of the cell which issued an interrupt and all the

cell indexes in the CTM table is done concurrently at one clock cycle. This choice guaranteed that the HCM is able to run at high speed, however, it costs much of the FPGA resource. Especially when there is no dedicated CAM in the FPGA architecture, the circuit designer has to build up CAM using FFs.

For a 16-cell system, the HCM requires 8,677 (12.55%) FFs and 4,765 (6.89%) LUTs of the whole FPGA. Thus, the HCM sizes five times bigger than the cell size defined in our architecture (1,600 FFs and 1,600 LUTs). To give a rough idea, a Microblaze softcore requires between 600 LUTs for the smallest configuration and 4000 LUTs for the largest one (including cache, FPU, MMU). Thus, the size of this implementation is clearly not negligible.

The reason of such high HCM resource consumption, as we have analyzed previously, comes from the CAM implementation choice. Noting that the reconfiguration of a cell through a 32-bit and 100 MHz ICAP requires 14760 clock cycles, and that the time for an allocation request of crossing the current HCM implementation only takes 4 cycles, we have clearly over-optimized the HCM circuit which is not quite critical in the whole allocation procedure. To better compromise the resource utilization and the running speed, we may have a different hardware HCM implementation which uses Block RAMs resource in the FPGA to build up the CTM, or have a software HCM implementation which is realized by a microblaze processor embedded in the FPGA, since we still have some margin of time.

6.2 Communication Mechanism Validation

In the following section, we are going to describe the experiments of validating the communication mechanism proposed in the chapter 5. The validation is achieved in two steps. Firstly, we verify that the original MWMR channel works properly in our implementation environment. Then, we show that in cooperation with the HCM, the dynamicity can be added to the original MWMR channel, so that the whole communication mechanism is able to handle the intertask communications in various runtime task mapping cases.

6.2.1 Experiment 3: Original MWMR Channel Migration

The object of this validation step is to guarantee that the original MWMR channel works well when we migrate it in our implementation environment. To do so, we have to prove that the MWMR channel can be accessed successfully by any number of tasks of any nature, under the condition that the task mapping is known in advance and kept unchanged during the system runtime. In other words, we should verify (1) that the software tasks can correctly access an MWMR channel through the *mwmr_read* and/or *mwmr_write* services, (2) that the hardware tasks can correctly access an MWMR channel through a MWMR controller, and (3) that when there are multiple accessing tasks, they can get data from and/or send data to a MWMR channel without error in a cooperative manner.

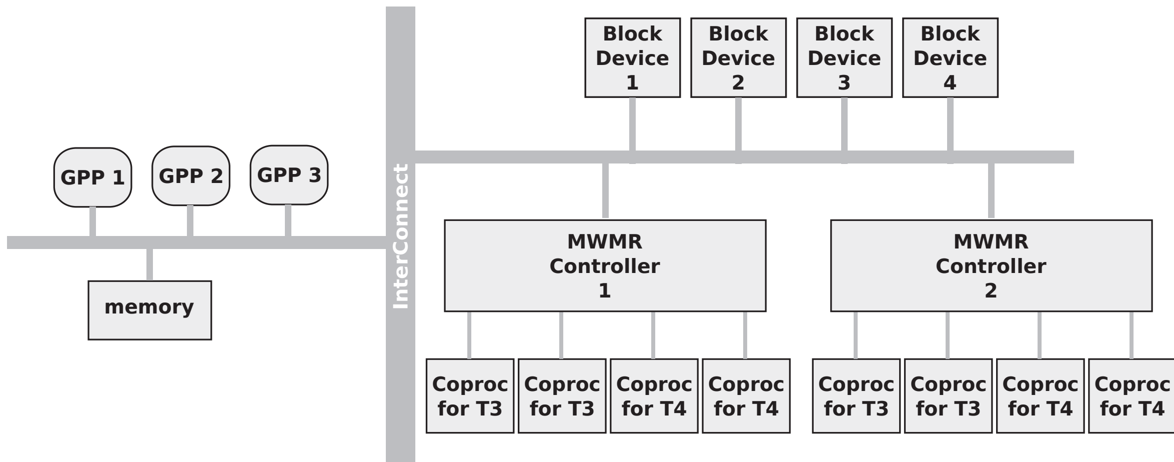


Figure 6.10: Platform for Testing the Original MWMR channel

In order to keep the consistency with the platform template proposed in chapter 4 for the GPP/FPGA dynamic reconfigurable architecture, we tested the original MWMR channel in the platform shown Figure 6.10. The platform is a simulated one, built using SoCLib [soc10] simulation models written in SystemC, containing three GPPs, the shared memory, four block devices (a component emulating a disk controller in order to transfer data from (resp. to) files in the host system to (resp. from) buffers in the memory of the virtual platform) and two MWMR hardware controllers connected to four cells each. The eight cells are configured as two kinds of specific coprocessors.

The software environment of our test uses the DnaOS operating system. The MWMR channel access services are implemented as a user library. There are four simple tasks (T1, T2, T3, T4) programmed as the basic computations of the applications. Each task implements the function that makes a grey scale image brighter for certain degrees (Ti increases the grey scale of each pixel in an image by 16i). Amongst the four tasks, T1 and T2 are pure software tasks. T3 and T4 are implemented as hardware coprocessors, in addition to their software alternatives.

Figure 6.11 shows the five applications that we use as benchmarks built using the four tasks. They are executed to validate the nine cases of possible combination of data producers and consumers at the two ends of a MWMR channel. The names of the nine test cases are the concatenation of two parts. The first part of the name indicates the nature of the MWMR channel accessors, while the second part of the name indicates the number of the MWMR channel accessors.

In sub-figure (A), black images (grey scale value of each pixel is 0x0) are stored in block device 1. They are read by software task T1 through the FIFO channel 1. T1 turns every pixel value to 0x10, and writes the output data into the MWMR channel 1. The software task T2 reads its input data from the MWMR channel 1, increases every pixel value by 0X20 and writes them into block device 3 through FIFO channel 2. A final dark grey image (each pixel value equals to 0x30) in block device 3 would prove the success of the experiment. Indeed, this experiment verified that the MWMR channel is able to handle the communication between a single software

task and another single software.

The application in sub-figure (B) is almost the same case as in sub-figure (A), the only difference being that other software instances of T1 and T2, called T1′ and T2′, are added to run in parallel. This experiment also verified that the MWMR channel can handle the communication amongst multiple software task writers and multiple software task readers.

In sub-figure (C), (D) and (E), black images are processed by T1, T3, T4 and T2 in sequence. The MWMR channel 1, 2 and 3 are used to connect two adjacent tasks. In sub-figure (C), T3 and T4 are hardware tasks connected to different MWMR hardware controllers. The experiment again proved that the MWMR channel is able to handle the communication between a single software task writer and a single hardware task reader (the MWMR channel 1), the communication between a single hardware task writer and a single hardware task reader (the MWMR channel 2) and the communication between a single hardware task writer and a single software task reader (the MWMR channel 3).

By adding to each task an instance of the same nature, the application in sub-figure (D) proved that the three cases verified in sub-figure (C) can be handled by the MWMR channel even when there are multiple writers and readers.

In sub-figure (E), T1 and T2 each have two software instances. T3 has a software instance and a hardware instance, and so does T4. The T3 and T4 hardware instances are connected to the same MWMR hardware controller. This experiment finally proved that the MWMR channel can be accessed by software tasks and hardware tasks at the same time.

The hardware tasks which are connected to the same MWMR hardware controller can communicate correctly (as T3 and T4 in sub-figure (E)), so do the hardware tasks which are connected to different MWMR hardware controllers (As T3 and T4 in sub-figure (C) and (D)). This validation is the foundation of the communication mechanism based on the MWMR-hardware-controller-centered clusters, which is proposed in the chapter 5.

We may have noted that the MWMR channel usage scenario in the sub-figure (A) and (B) is the almost the same. So does the MWMR channel usage scenario in the last three sub-figures. The reason why we have to write five applications instead of two is that the original MWMR channel has to be connected to its accessors at the system boot time and has to stay unchanged. That is also the motivation for which we would like to integrate the HCM with the MWMR channel, in order to bring the dynamicity to the communication mechanism.

6.2.2 Experiment 4: Dynamicity Management with the HCM and Modified MWMR Channels

The objective of this experiment is to verify the dynamicity of our communication mechanism. As explained in chapter 5, the original MWMR channel already allows any number of software tasks to access the data at any time, so it is sufficient for us to prove that our communication mechanism allows hardware tasks to do the same thing even when the hardware tasks are dynamically-mapped.

The platform of the experiment is shown Figure 6.12. It is composed from three

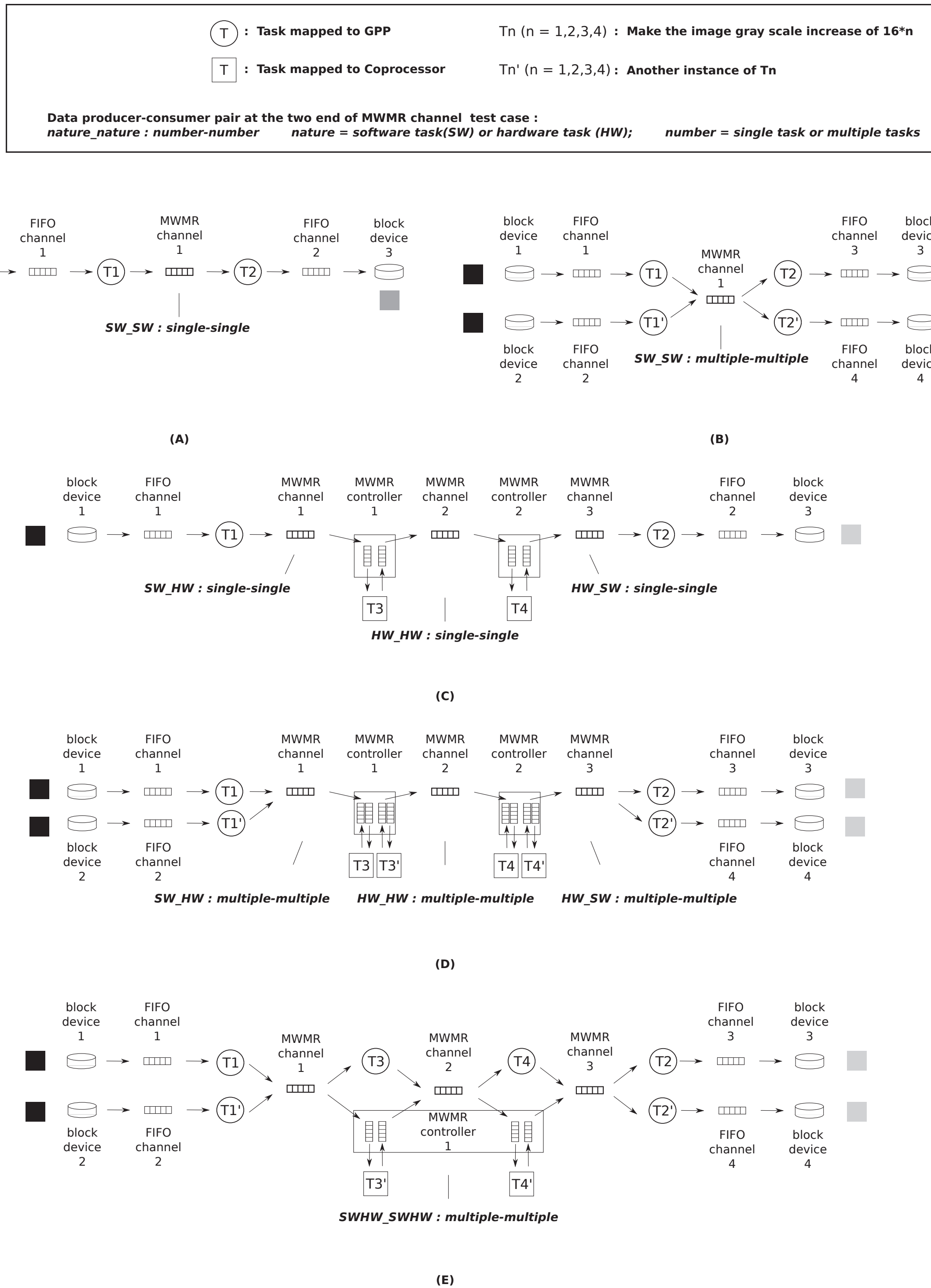


Figure 6.11: MWMR Channel Validation in Various Data Producer-Consumer Pair Cases

GPPs which share the common memory, an HCM which controls three reconfiguration controllers and four cells. Two MWMR controllers each connect to the FIFO I/O interfaces of two cells. Six block devices are used to store the data before and

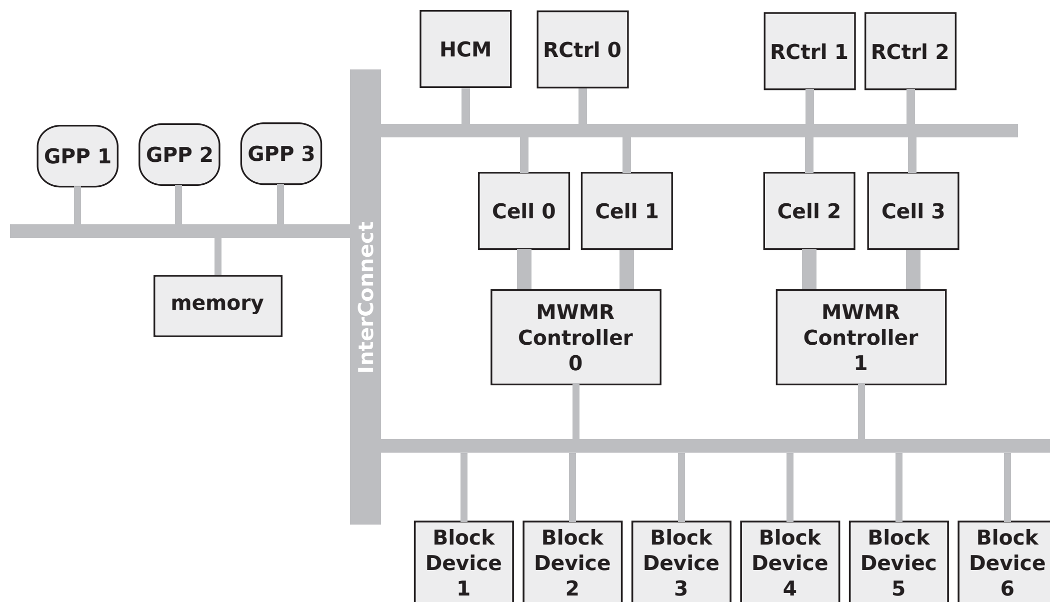


Figure 6.12: The Platform for Testing the Dynamic Communication Mechanism

after processing.

This platform is a multi-GPP, multi-reconfiguration controller and multi-cell use scenario of the HCM. As a specific example of proposed cluster-based communication mechanism, the platform is organized as two clusters, and each cluster contains one MWMR hardware controller and two cells. The platform is implemented in SystemC as described before.

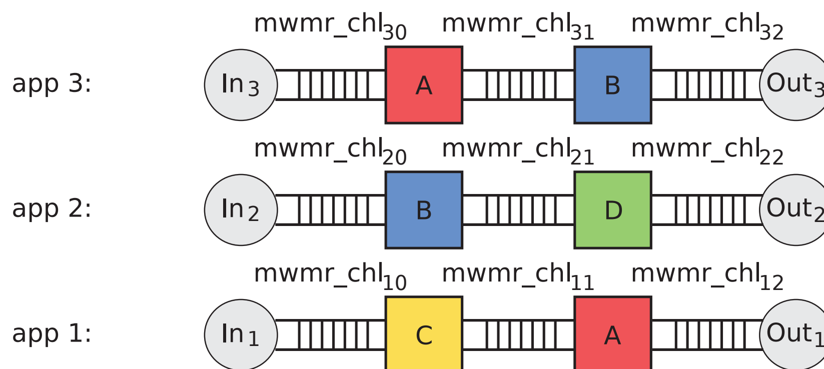


Figure 6.13: The Platform for Testing the Dynamic Communication Mechanism

The schematic diagrams of the applications in this experiment are shown in Figure 6.13. We can see that they are quite similar to the applications in the experiment 1 (Figure 6.3). The computation performed by hardware tasks (square box) are the same as the ones presented in Table 6.2.

In this experiment, the three applications are running in parallel. Each application consists of two software tasks and two hardware tasks. Connected by MWMR channels, the four tasks in an application are launched in parallel. The software task

In_i (i = 1, 2, 3) is mainly responsible for reading images from a block device and rearranging the data for the following image processing task. The software task Out_i (i = 1, 2, 3) is responsible for obtaining data from the last image processing task and organizing them as images to write to another block device. The two hardware tasks perform image processing functions, constantly driven by the data from from last step and sending the generated data to the next step, until a whole image has been processed. After that, the hardware task considers the computation completed. As a consequence, the current occupied cell is released. Before processing the next image, the hardware task has to be allocated to a cell according to the resource usage at that moment.

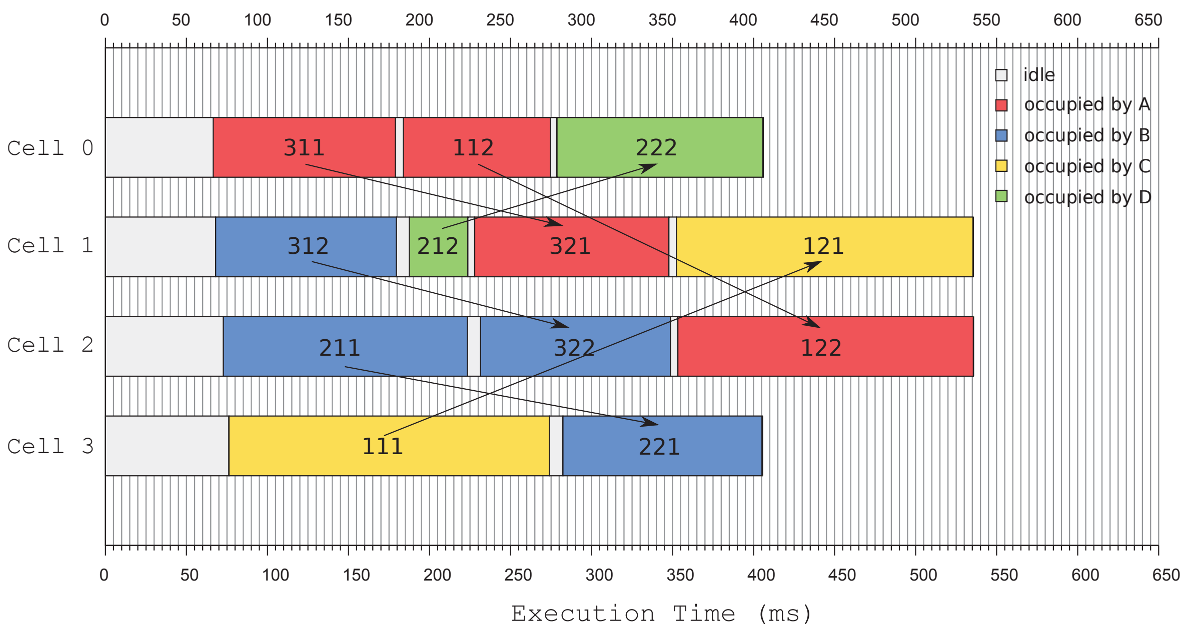


Figure 6.14: Underlying Cell Usage for Each Application Processing the Two First Images

We have launched these three applications in parallel, Figure 6.14 shows the underlying cell usage for each application to process the two first images. In the figure, on each period of time when cells are occupied, there is a code to identify the owner of the hardware component. The code is composed from three numbers. The first one indicates the number of the application; the second one indicates the number of the processed image; and the third one indicates the identifier of the image processing task. For example, the number 312 represents that the component executes the second task of the application 3 while processing the first image.

When writing the application, the programmer only knows the logic connection amongst tasks. This logic connection remains unchanged during the executing time. However, we can see clearly from Figure 6.14 that the locations of these tasks changed dynamically during the run time. These physical locations are determined depending on the cell usage at a specific moment. The programmers have no way to predict the physical location of a component. It is the communication mechanism based on the HCM and the extended MWMR channel services which handles the

redirection of data flow to the new location.

Our experiments produced the expected results, which validate the principles of our solutions. We thus have shown that:

1. This kind of redirection can be managed inside a cluster.
   For example, component A function of application 3 is firstly mapped to cell 0 (component number 311), before being remapped to cell 1 (component number 321). Similar examples include also component B function in application 2 (from cell 2 to cell 3), and component D function in application 2 (from cell 1 to cell 0).

2. This kind of redirection can be managed amongst different clusters.
   For example, component B in application 3 is firstly mapped to cell 1 (component number 312), then remapped to cell 2 (component number 322). Similar examples include component C in application 1 (from cell 3 to cell 1), and component A in application 1 (from cell 0 to cell 2).

6.3 Conclusion

In this chapter, we answer the question posed in chapter 2 "how to ease the life of application programmers" by experimentally approving our approaches presented in chapter 4 and chapter 5: to have a centralized abstraction layer to separate the notion of tasks from its implementation, and to have a scalable communication mechanism to gain the independence of task access from the nature and number of implementations.

Moreover, through the measurement of the experiments, we showed that with a reasonable price, the HCM and dynamic MWMR channels – the implementations of the two above approaches can be easily integrated into an OS. The OS integration eases further the application programmers work.

Chapter 7

Conclusion

7.1 Contribution

The work introduced in this thesis aims at providing a flexible hardware/software environment comprising CPUs and a pool of reconfigurable elements, which is a promising way to take advantage of both the hardware speed and the software versatility. We showed that the major issues with such architectures were the task allocation mechanism and its associated hardware resource management, the communication mechanism between tasks, and the issues related to code generality and hardware specificities. In the problem statement chapter, we asked several questions related to these issues, for which we will try to answer now using the work presented in past chapters.

How to allocate tasks to different resources in a CPU/FPGA hybrid system where dynamic partitioning is allowed?

We saw in chapter 4 that task allocation should be performed using at best the available parts of the FPGA. We also showed that hardware tasks should be reusable whenever possible, so as to avoid costly useless reconfigurations. In order to achieve this, we presented the key notion of Hardware Component to abstract a task running on the Reconfigurable Fabric. This notion allows the programmer not to care about the real FPGA below and instead to reason in terms of operations (e.g. activate the hardware component) and status (e.g. computation is complete). We further presented the Hardware Component Manager (HCM), whose role is to centralize the knowledge of all the hardware components. Using these two concepts, we are able to provide a task allocation mechanism which guarantees a efficient usage of hardware resources.

How should the reconfiguration related resources (RFs, bitstreams, reconfiguration ports) be managed, so that DR processing can be well maintained even in multi-threaded, multi-FPGA environment?

Apart from being efficient, the utilization of hardware resources should guarantee exclusivity properties so as to ensure that other applications do not interfere

with currently running applications. This is why we showed in chapter 4 and 5 that the reconfiguration related resources should be managed in a way that applications do not have access to low-level information, and that this kind of information should be kept in a centralized way and protected module. We decided to store these information in the HCM as it seemed well suited given our design. In case of a multi-FPGA environment, the centralization in the HCM of all reconfiguration related resources permits a cooperative resource management.

How to design a communication mechanism which can recognize the existence of dynamically appearing communicating entities, no matter what the nature (hardware or software) and number of communicators at the both sides of a producer-consumer pair?

In chapter 5, we presented the difficulty to write correct code when the number of instances of some tasks cannot be known in advance. In particular, tasks left unmapped can lead to a loss of data if the connection between tasks is not made properly. To solve this, we proposed to use an existing communication channel model called MWMR to perform the communications between tasks. Using the task mapping information, we integrated the ability for the HCM to dynamically link and unlink hardware tasks instances to the corresponding MWMR channels, by the means of five communication services. Two levels of configurations are presented: one linking tasks with MWMR controllers, since one task can be mapped to different cells, which are fixed to different ways of a MWMR controller or even different MWMR controllers; and one linking controllers to channels themselves, since the latter are located in memory, and are constructed and destroyed as the need of applications.

How to ease the life of application programmers by separating the management of task, reconfiguration resource and communication, so that they can write more flexible applications?

In chapter 4, we presented a 3-level layered architecture, comprising a) the application: b) the Hardware Component Manager, and c) the Reconfigurable Hardware Resources, which is in charge of knowing the functionality and occupation of cells. In a way, the HCM is acting as a "hardware server" for application, taking and responding to their requests. This dividing makes application more flexible since the interface provided hides the nature, status and occupation of the FPGA. Besides, the ability to use dynamically linked communication channels is also a step towards application flexibility. Of course, such hardware modules have to be integrated in the operating system to be truly usable. This is what we did by integrating this layered architecture into an existing operating system called DnaOS. We then showed in chapter 6 that the provided integrated services had a limited impact in terms of performance compared to a manual utilization of the FPGA. Overall, our proposal respects the separation between applications and hardware management through the use of the HCM, for the hardware mapping of both tasks and communication channels, and at a limited cost.

Perspectives and Future Work

For the work done in this thesis, we have the following perspectives.

Development of a real prototype on a FPGA platform

Currently, we use an open simulation platform named SocLib for validating the correctness and efficiency of the HCM and the communication mechanism. Based on this, it will be good to develop a prototype using a real FPGA device. In such a realistic environment, we may better observe the actual behavior and performance, and validate the robustness of our proposal. An HCM implementation has been developed for FPGA. The development of other parts of the prototype is an ongoing work.

Mixed hardware/software task implementations

At the current stage of development, whenever a hardware task faces the problem of no more resource available, it is blocked by the HCM until some cells are released, even if some processors are idle.

An idea to avoid this is to have mixed hardware/software descriptions, so that the operating system can decide online with the knowledge of resource usage whether to run a task in software or in hardware.

To achieve this, it is necessary to integrate a mechanism which allows the HCM to yield the control back to the operating system. This can be done using system calls inside the HCM driver. Thus, when a task containing a hardware description is launched, it can be placed on the FPGA if enough cells are available; otherwise it can be placed on a processor.

Taking advantage on the fact that actual hardware and software tasks use the same interface, this evolution only requires the management of mixed descriptions.

Dynamic management of mixed implementations

Based on the previously presented evolution, we can also imagine the infrastructure to be able to dynamically switch between several implementations of one task, in order to automatically balance the hardware load of the platform.

A task running in software can be migrated to hardware when a cell becomes available. This requires the need to migrate the intermediate state of the task, since we can't retrieve the consumed inputs and we don't want to rewrite the outputs which have already been written.

Reciprocally, a task running in hardware must be able to migrate to software if the system needs to evict a running task to make room for hardware only tasks.

HCM with support for heterogeneous granularity

The homogeneous choice that we have made in this work has the advantage of simplicity at different levels; however, it can lead to a waste of area when the tasks consume a wide range of surfaces. In this case, it is necessary to consider the biggest task in terms of surface to define the cell size.

To solve this problem, we might want the HCM to handle various size of cells. The imagined approach consists in combining the cells in a hierarchical way, such that four adjacent cells of the same level can be merged to create a cell of the higher level, following the principle of the buddy memory allocator [Kno65], but in two dimensions.

Improvement of on-demand placement strategy

Currently, there is no management of priority of hardware tasks – in case of contended resources –, but there is even no guarantee that when no resource is available, the upcoming tasks will be granted allocation in order of arrival.

This perspective aims at adding a priority mechanism for hardware task allocation. Each task can be specified a priority level. Based on this level:

- if two tasks of different levels are blocked until some cells are available, the one with higher priority will get the resource first
- if two task with the same level are blocked, the one which accessed the HCM first will get the resource first.

This evolution also requires the HCM to give back the control to the operating system, for the management of associated priority queues.

Optimization of the number of I/O channels per cell

In the current infrastructure, each cell is connected to two ways of an MWMR controller. This connection is fixed in hardware and thus the number of connections canot be changed at runtime. However, many tasks do not use only two channels, what requires multiplexing and/or demultiplexing at channels ends. This multiplexing may be a bottleneck for some tasks, and we suggest to study the impact on performance of different number of I/O channels per tasks.

Of course, the more I/O channels are connected, the more hardware controller will be required, so a tradeoff between both must be analyzed.

Optimization of communications between hardware tasks

Currently, when two hardware tasks communicate via a MWMR channel, the data has first to be copied from the first controller to memory where the data part of the channel lies. Then, it has to be copied back from memory to the second MWMR controller. It is possible that these two controllers are actually two ways of the same controller, thus generating a lot of data transfers.

We think that these transfers could be avoided with an appropriate bypass mechanism, which could route directly data from a way of a controller to another, or even to the way of a distinct controller.

If combined with dynamic task migration, when a task is migrated from hardware to software, the mechanism needs to remove the bypasses of the migrated task, by informing the corresponding controllers.

Bibliography

[Ahm01] Elias Ahmed. The effect of logic block granularity on deep-submicron fpga performance and density. Master's thesis, University of Toronto, 2001. 3.1.2.1

[APDG05] I. Auge, F. Pétrot, F. Donnet, and P. Gomez. Platform based design from parallel C specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24:1811–1826, Dec. 2005. 3.1.1.2, 6.1.3.2

[ASA+08] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. Achieving programming model abstractions for reconfigurable computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):34–44, 2008. 3.2.3.1

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM. 3.1.1.1

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, 1995. 3.1.1.3

[BLM96] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. 3.1.1.3

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007. 1

[BSH08] L. Bauer, M. Shafique, and J. Henkel. Efficient resource utilization for an extensible processor through dynamic instruction set adaptation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(10):1295–1308, 2008. 3.2.4.4

[Bui13] Dai Bui. *Scheduling and Optimizing Stream Programs on Multicore Machines by Exploiting High-Level Abstractions*. PhD thesis, EECS Department, University of California, Berkeley, Nov 2013. 3.1.1.3

[CC01] Jorge E. Carrillo and Paul Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, pages 141–150, New York, NY, USA, 2001. ACM. 3.1.2.3

[CDHL12] Youenn Corre, Jean-Philippe Diguet, Dominique Heller, and Loïc Lagadec. A framework for high-level synthesis of heterogeneous mp-soc. In *Proceedings of the Great Lakes Symposium on VLSI*, GLSVLSI '12, pages 283–286, 2012. 3.1.1.2

[CFF+99] D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, pages 23–40, 1999. 3.1.2.3

[CHW00] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *Computer*, 33(4):62–69, 2000. 3.1.2.3

[CM08] P. Coussy and A. Morawiec, editors. *High-Level Synthesis from Algorithm to Digital Circuit*. Springer Netherlands, 2008. 4.1.2.3

[CMN+09] Simone Corbetta, Massimo Morandi, Marco Novati, Marco Domenico Santambrogio, Donatella Sciuto, and Paola Spoletini. Internal and external bitstream relocation for partial dynamic reconfiguration. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(11):1650–1654, nov. 2009. 4.1.1.1

[DCL13] PapaIssa Diallo, Joël Champeau, and Loïc Lagadec. A model-driven approach to enhance tool interoperability using the theory of models of computation. In Martin Erwig, RichardF. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2013. 3.1.1

[DCPS02] Raphaël David, Daniel Chillet, Sebastien Pillement, and Olivier Sentieys. DART: a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints. *Parallel and Distributed Processing Symposium, International*, 2:0156, 2002. 3.1.2.1

[DML11] François Duhem, Fabrice Muller, and Philippe Lorenzini. Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on fpga. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek El-Ghazawi, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 6578 of *Lecture Notes in Computer Science*, pages 253–260. Springer, 2011. 6.1.3.2

[dna10] DnaOS website. `http://tima.imag.fr/sls/research-projects/application-elements-for-socs`, 2010. 6.1

[Est60] Gerald Estrin. Organization of computer systems: The fixed plus variable structure computer. In *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '60 (Western), pages 33–40, New York, NY, USA, 1960. ACM. 1

[Fau07] Etienne Faure. *Communications matérielles/logicielles dans les systèmes sur puces multi processeurs orientés télécommunications*. Thèse de doctorat, Spécialité Informatique, Université Pierre et Marie Curie, 2007. 5.2

[FMG13] Clément Foucher, Fabrice Muller, and Alain Giulieri. Online codesign on reconfigurable platform for parallel computing. *Microprocessors and Microsystems*, 37(4–5):482 – 493, 2013. 3.2.1.3

[For12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications. 3.2.1.3

[GG09] Samuel Garcia and Bertrand Granado. Ollaf: A fine grained dynamically reconfigurable architecture for os support. *EURASIP Journal on Embedded Systems*, 2009(1), 2009. 4.1.2.2

[GKM+12] Laurent Gantel, Amel Khiar, Benoit Miramond, Mohamed El Amine Benkhelifa, Lounis Kessal, Fabrice Lemonnier, and Jimmy Le Rhun. Enhancing reconfigurable platforms programmability for synchronous data-flow applications. *ACM Trans. Reconfigurable Technol. Syst.*, 5(3):14:1–14:16, October 2012. 3.2.1.1

[GP09] X. Guerin and F. Petrot. A system framework for the design of embedded software targeting heterogeneous multi-core socs. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 153–160, 2009. 6.1

[HD07] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. (document), 3.1.2, 3.1.2.3, 3.1

[HKHT05] R. Hecht, S. Kubisch, A. Herrholtz, and D. Timmermann. Dynamic reconfiguration with hardwired networks-on-chip on future FPGAs. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 527–530, aug. 2005. 4.1.1.1

[hSAS+07] Hayden Kwok hay So, Borph An, Operating System, Fpga based Reconfigurable, and Hayden Kwok hay So. Borph: An operating system for fpgabased reconfigurable computers. Technical report, 2007. 3.2.2.1

[idc11] Idc digital universe study. http://idcdocserve.com/1414, 2011. 1

[IS11] Aws Ismail and Lesley Shannon. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '11, pages 170–177, 2011. 3.2.3.3

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974. 3.1.1.2

[KLPR05] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. REPLICA: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005. 4.1.1.1

[Kno65] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965. 7.1

[KTR08] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, February 2008. 3.1.2.1

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 3.1.1.3

[LMA+12] F. Lemonnier, P. Millet, G.M. Almeida, M. Hubner, J. Becker, S. Pillement, O. Sentieys, M. Koedam, S. Sinha, K. Goossens, C. Piguet, M.-N. Morgan, and R. Lemaire. Towards future adaptive multiprocessor systems-on-chip: An innovative approach for flexible architectures. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 228–235, July 2012. 3.2.1.4

[LP07] Enno Lübbers and Marco Platzner. Reconos: An rtos supporting hard- and software threads. In Koen Bertels, Walid A. Najjar, Arjan J. van Genderen, and Stamatis Vassiliadis, editors, *FPL*, pages 441–446. IEEE, 2007. 3.2.3.2

[LP09] Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, October 2009. 3.2.3.2

[LS96] P. Lysaght and J. Stockwood. A simulation tool for dynamically reconfigurable field programmable gate arrays. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(3):381–390, 1996. 6.1.3.1

[LTC+03] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri. A vliw processor with reconfigurable instruction set for embedded applications. *Solid-State Circuits, IEEE Journal of*, 38(11):1876–1886, 2003. 3.2.4.2

[Mut03] S. Muthukrishnan. Data streams: Algorithms and applications. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’03, pages 413–413, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. 3.1.1.1

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, September 2005. 1

[ope11] OpenCL official website. http://www.khronos.org/opencl/, 2011. 3.2.1.3

[Par95] Thomas Martyn Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Berkeley, CA, USA, 1995. UMI Order No. GAX96-21312. 3.1.1.2

[QSN06] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. A parallel configuration model for reducing the run-time reconfiguration overhead. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, pages 965–969, 2006. 4.1.1.2

[RS94] R. Razdan and M.D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, pages 172–180, 1994. 3.1.2.3

[SB09] M. Sabeghi and K. Bertels. Toward a runtime system for reconfigurable computers: A virtualization approach. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1576–1579, 2009. 3.2.4.3

[soc10] Soclib website. http://www.soclib.fr/trac/dev, 2010. 6.1.3.1, 6.2.1

[SSB09] M. Sabeghi, V.-M. Sima, and K. Bertels. Compiler assisted runtime task scheduling on a reconfigurable computer. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 44–50, 2009. 3.2.4.3

[TGB+06] B.D. Theelen, M. C W Geilen, T. Basten, J. P M Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194, 2006. 3.1.1.3

[VPI05] M. Vuletid, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *Design Test of Computers, IEEE*, 22(2):102–113, 2005. 3.2.3.5

[WK01] Grant B. Wigley and David A. Kearney. The development of an operating system for reconfigurable computing. In *FCCM*, pages 249–250. IEEE, 2001. 3.2.1.2

[WKJ06] Grant B. Wigley, David A. Kearney, and Mark Jasiunas. Reconfigme: a detailed implementation of an operating system for reconfigurable computing. In *IPDPS*. IEEE, 2006. 3.2.1.2

[WZW+13] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong. Spread: A streaming-based partially reconfigurable architecture and programming model. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(12):2179–2192, 2013. 3.2.3.4

[Xil11] XilinxInc. Partial reconfiguration user guide. Xilinx Inc. on-line documentation, October 2011. 6.1.3.2

[YMHB00] Z.A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 225–235, 2000. 3.1.2.3, 3.2.4.1

Résumé Cette thèse s'intéresse aux architectures contenant des FPGAs reconfigurables dynamiquement et partiellement. Dans ces architectures, la complexité et la difficulté de portage des applications sont principalement dues aux connections étroites entre la gestion de la reconfiguration et le calcul lui-même. Nous proposons 1) un nouveau niveau d'abstraction, appelé gestionnaire de composants matériels (HCM) et 2) un mécanisme de communication scalable (SCM), qui permettent une séparation claire entre l'allocation d'une fonction matérielle et la procédure de reconfiguration. Cela réduit l'impact de la gestion de la reconfiguration dynamique sur le code de l'application, ce qui simplifie grandement l'utilisation des plateformes FPGA. Les application utilisant le HCM et le SCM peuvent aussi être portées de manière transparentes à des systèmes multi-FPGA et/ou multi-utilisateurs. L'implémentation de cette couche HCM et du mécanisme SCM sur des plateformes réalistes de prototypage virtuel démontre leur capacité à faciliter la gestion du FPGA tout en préservant les performances d'une gestion manuelle, et en garantissant la protection des fonctions matérielles. L'implémentation du HCM et du mécanisme SCM ainsi que leur environnement de simulation sont open-source dans l'espoir d'une réutilisation par la communauté.

Mots-Clés Calcul reconfigurable, virtualisation matérielle, gestion des tâches, mécanismes de communication, décision en ligne, gestion des zones reconfigurables

---

Abstract This thesis shows that in FPGA-based dynamic reconfigurable architectures, the complexity and low portability of application developments are mainly due to the tight connections between reconfiguration management and computation. By proposing 1) a new abstraction layer, called Hardware Component Manager (HCM) and 2) a Scalable Communication Mechanism (SCM), we clearly separate the allocation of a hardware function from the control of a reconfiguration procedure. This reduces the dynamic reconfiguration management impact on the application code, which greatly simplifies the use of FPGA platforms. Applications using the HCM and the SCM can also be transparently ported to multi-user and/or multi-FPGA systems. The implementation of this HCM layer and the SCM mechanism on realistic simulation platforms demonstrates their ability to ease the management of FPGA flexibility while preserving performance and ensuring hardware function protection. The HCM and SCM implementations and their simulation environment are open-source in the hope of reuse by the community.

Keywords Reconfiguration computing, hardware virtualization, task management, communication mechanism, online decision, reconfigurable area management

ISBN: 978-2-11-129189-8
Laboratoire TIMA – 46 Avenue Félix Viallet, 38000 Grenoble