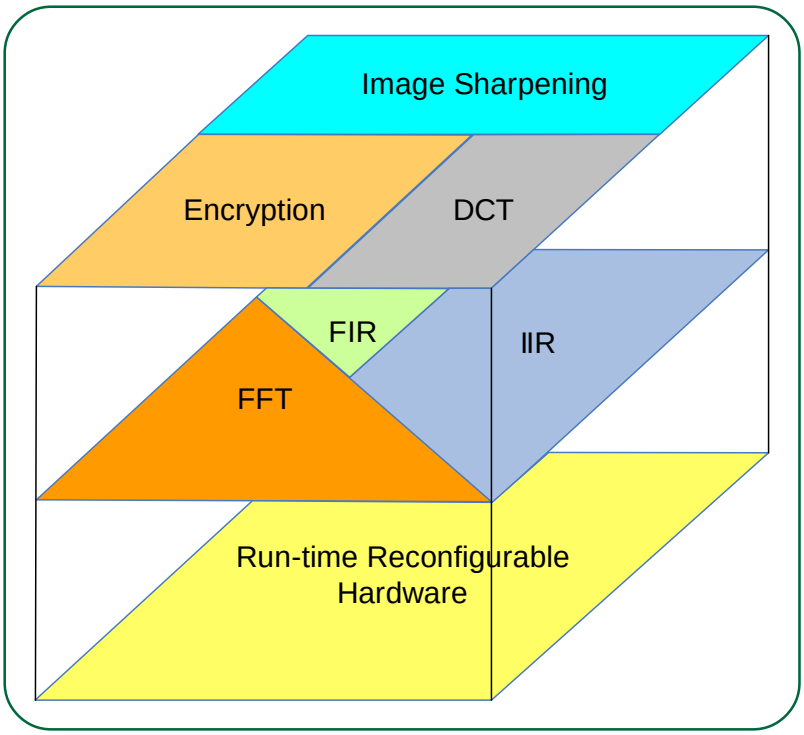
ESPOO 2007 VTT PUBLICATIONS 659



Yang Qu

System-level design and configuration management for run-time reconfigurable devices

VTT PUBLICATIONS 659

System-level design and configuration management for run-time reconfigurable devices

Yang Qu

Academic dissertation to be presented with the assent of the Department of Information Technology, Tampere University of Technology, for public discussion on November 30th, 2007 at 12 noon in room TB222 of Tietotalo building, Korkeakoulunkatu 1, Tampere.

ISBN 978-951-38-7053-9 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 978-951-38-7054-6 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 3, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 3, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 3, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax + 358 20 722 4374

VTT, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Technical Research Centre of Finland, Kaitoväylä 1, P.O. Box 1100, FI-90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Technical editing Leena Ukskoski

Edita Prima Oy, Helsinki 2007

Qu, Yang. System-level design and configuration management for run-time reconfigurable devices. Espoo 2007. VTT Publications 659. 133 p.

Keywords dynamically reconfigurable hardware, run-time reconfiguration, system-level design, task scheduling, configuration locking, configuration parallelism

Abstract

Dynamically reconfigurable hardware (DRHW) not only has high silicon reusability, but it can also deliver high performance for computation-intensive tasks. Advanced features such as run-time reconfiguration (RTR) allow multiple tasks to be mapped onto the same device either simultaneously or multiplexed in time domain. This new type of computing element also brings new challenges in the design process. Design supports at the system level are needed. In addition, the configuration latency and the configuration energy involved in each reconfiguration process can largely degrade the system performance. Approaches to efficiently manage the configuration processes are needed in order to effectively reduce its negative impacts. In this thesis, system-level supports for design of DRHW and various configuration management approaches for reducing the impact of configuration overhead are presented.

Our system-level design supports are based on the SystemC environment. An estimation technique for system partitioning and a DRHW modeling technique are developed. The main idea is to help designers in the early design phase to evaluate the benefit of moving some components from fixed hardware implementation to DRHW. The supports have been applied in a WCDMA case study. In order to efficiently apply the multi-tasking feature of DRHW, we have developed three static task scheduling techniques and a run-time scheduling technique. The static schedulers include a list-based heuristic approach, an optimal approach based on constraint programming and a guided random search approach using a genetic algorithm. They are evaluated using both random tasks and real applications. The run-time scheduling uses a novel configuration locking technique. The idea is to dynamically track the task status and lock the most frequently used tasks on DRHW in order to reduce the number of reconfigurations. In addition, we present two novel techniques to reduce the configuration overhead. The first is configuration parallelism. Its idea is to enable tasks to be loaded in parallel in order to better exploit their parallelism. The second is dynamic voltage scaling. The idea is to apply low supply voltage in reconfiguration process when possible in order to reduce the configuration energy.

Preface

The research was carried out at the Technical Research Centre of Finland (VTT) from 2004 to 2007. Most of the work presented in this thesis was conducted in the ADRIATIC (Advanced methodology for designing reconfigurable SoC and application-targeted IP-entities in wireless communications) project, funded by the European Commissions's "IST" initiative, and the MARTES (Model-based approach to real-time embedded systems development) project, funded by Tekes. I would like to thank the funding organizations and all the people who have been involved in these projects. Especially, I have to thank Mr. Kari Tiensyrjä for setting up and managing the projects so that I could have the opportunity to complete my thesis work.

I owe thanks to lots of people for their help during this work. I would like to thank Dr. Juha-Pekka Soininen for leading me into the research field and guiding me in the various research stages. His constant inspiration has led to the completion of this thesis. I wish to express my sincere gratitude to my advisor, Professor Jari Nurmi, for his continuous support and his total confidence in my work. I would also like to express my appreciation to Dr. Juan José Noguera Serra for his comments and help with my research. I wish to thank the reviewers of the thesis, Professor Johan Lilius and Professor Peeter Ellervee, for their insightful comments.

I would like to thank my VTT colleagues, Mr. Jari Kreku, Mr. Tapio Rautio, Mr. Jussi Roivainen, Mr. HuaGeng Chi, Mrs. Yan Zhang and Dr. Martti Forsell, for all the lively discussions throughout this work. I wish to thank Mr. Antti Pelkonen, Mr. Konstantinos Masselos and Mr. Marko Pettissalo for their help in the ADRIATIC project. I also extend my gratitude to my colleagues in the Tampere University of Technology, Mr. Xin Wang, Mr. Bin Hong and Dr. Tapani Ahonen, for their help. VTT has provided an excellent working environment and research facilities. I would like to thank the management team, the publishing team and all the VTT people for helping me to complete this thesis. I want to thank Nokia Foundation and the Jenny ja Antti Wihurin Rahasto for their motivation and financial support.

I owe lots of thanks to my friends ShuFeng Zheng and HongLei Miao for their constant encouragement and all the unforgettable moments that we share together. I wish to thank Mikko Alatossava, Attaphongse Taparugssanagorn and the rest of my football team for helping me to refresh myself.

More importantly, I want to thank my wife, Jing Chai, for her sincere love, encouragement and understanding. I would not be able to complete my doctoral studies without her. I wish to thank my mother and my parents-in-law for their ever-present care and faith in me. I want to thank my brother, my sister-in-law, my nephew and all my other family members for supporting me when my father passed away. I owe my father too much, and this thesis is in memory of him, with my deepest sorrow.

Yang Qu

October 6, 2007

Contents

List of symbols

1D	One-Dimension
2D	Two-Dimension
B&B	Branch and Bound
BRAM	Block RAM
AEP	Abstract Execution Platform
ALAP	As Late As Possible
ANSI-C	the standard published by the American National Standards Institute for the C programming language
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
C-string	Controller String
CAD	Computer-Aided Design
CDFG	Control/Data Flow Graph
CF	Compact Flash
CFG	Control Flow Graph
CL	Configuration Latency
CPICH	Common Pilot Channel
CPU	Central Processing Unit

CP	Constraint Programming
CS	Configuration Scheduler
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DC	Direct Current
DCS	Dynamic Circuit Switching
DFG	Data Flow Graph
DRCF	Dynamically Reconfigurable Fabric
DRHW	Dynamically Reconfigurable Hardware
DSM	Deep Sub-Micro
DSP	Digital Signal Processor
DVS	Dynamic Voltage Scaling
EDF	Earliest Deadline First
EDK	Embedded Development Kit
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
FDS	Force-Directed Scheduling
FeRAM	Ferroelectric Random Access Memory
FIFO	First In First Out

FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FPSLIC	Field Programmable System Level Integration Circuits
GA	Genetic Algorithm
GPP	General Purpose Processor
GSM	Global System for Mobile communications
HDL	Hardware Description Language
HW	Hardware
IMC	Interface Method Call
I/O	Input/Output
IP	Intellectual Property
IS	Input Splitter
ISS	Instruction-Set Simulator
ITRS	International Technology Roadmap for Semiconductors
JPEG	Joint Photographic Experts Group
LE	Logic Element
LUT	LookUp Table
LZW	Lempel-Ziv-Welch
MPEG	Moving Picture Experts Group

MRAM	Magnetic Random Access Memory
MUX	Multiplexer
NoC	Network-on-Chip
NRE	Non-Recurring Engineering
OCP	the Open Core Protocol
OS	Operating System
PAE	Processing Array Element
PLB	Processor Local Bus
PRM	Partially Reconfigurable Module
RAM	Random Access Memory
RC	Reconfigurable Cell
RISC	Reduced Instruction-Set Computer
RISP	Reconfigurable Instruction-Set Processor
RSoC	Reconfigurable System-on-Chip
RTL	Register Transfer Level
RTOS	Real-Time OS
RTR	Run-Time Reconfiguration
s-graph	Schedule Graph
SAT	Satisfiability

SoC	System-on-Chip
SRAM	Static Random Access Memory
SW	Software
T-string	Task String
TGFF	Task Graphs For Free
TLM	Transaction-Level Modeling
UML	Unified Modeling Language
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
vN	von Neumann
WCDMA	Wideband Code Division Multiple Access
WiMAX	World Interoperability for Microwave Access
WLAN	Wireless Local Area Network

1. Introduction

In April of 1965, Intel co-founder Gordon Moore published an article in Electronics Magazine. In the article, he stated that the logic density of integrated circuits has closely followed the relationship that the number of transistors per chip doubles every 12 months, which was revised to a slower period of 18 months in the late 1970s [1]. This prediction is honored as "Moore's Law" that has been verified for the past four decades. The 2005 edition of international technology roadmap for semiconductors (ITRS) [2] forecasted that advanced technologies would keep Moore's law to be valid until 2020, at which time 14nm technology will be commercially used. Currently, a single chip that contains more than 1.3 billion transistors [3] has been manufactured on 65nm technology, and it has been commercially sold on a vast scale.

On the other hand, new applications and algorithms are growing rapidly as well. Traditionally, they are implemented in general purpose processors (GPPs), which employ an instruction-stream-based von Neumann (vN) paradigm [4]. Applications are described in software and compiled into a sequence of instructions, which guide the microprocessor central processing unit (CPU) to perform operations step by step. However, as applications become more complex and algorithms become more computation-intensive, the vN-type processing cannot deliver the required performance. Additional supports are required. One solution is to couple the host processor with dedicated hardware accelerators. A straightforward example is that most personal computers (PCs) nowadays need a graphics processing unit (GPU) to drive the display.

Usually, hardware accelerators are implemented as application specific integrated circuits (ASICs). For a known and well-defined application, an optimized design can achieve high speed, high throughput as well as low power consumption and small chip area. However, the higher performance achieved over software implementation on GPPs is mainly due to the high Non-recurring engineering (NRE) cost and the sacrifice of flexibility. The challenges in deep sub-micro (DSM) technology, e.g., noise and transient errors [5, 6], and the increasing complexity of algorithms have increased the design gap and driven the design cost higher and lengthened the design time, which is unfavorable to electronics industry that suffers from shrinking time-to-market and decreasing product life cycles.

In addition, ASICs have very low silicon reusability, because they have a fixed hardware structure. When the implemented design becomes outdated or new requirement is needed, ASIC chips cannot re-shape themselves. One processing element that can provide both high performance and flexibility is a digital signal processor (DSP), which is usually implemented as a very long instruction word (VLIW) processor that has multiple function units and thus can exploit a certain degree of parallelism. However, DSPs fail to deliver the required performance for many modern applications that consist of very high computation-intensive algorithms, e.g., H.264 video codec [7]. In addition, DSPs are extremely power inefficient when compared to ASIC solutions.

Another alternative is reconfigurable systems, which usually appear in the form of field programmable gate array (FPGA). The FPGA is an array of gate-level configurable logic elements (LEs) embedded in a reconfigurable interconnect fabric. Both LEs and interconnect are programmable. They can implement any combinational logic as well as sequential circuit. In addition, customized blocks, such as hardwired multipliers and memories [8], can be embedded to support various kinds of DSP applications. Modern FPGA platforms also embed one or more than one hardwired processor. Such structure makes it possible to implement a rather complex system in such platforms. Because FPGAs are pre-fabricated chips with guaranteed performance, design cost and design time are much smaller when compared to ASIC implementation. FPGAs have been widely used in different applications, including image processing [9], SAT solvers [10], and cryptograph processors [11].

Based on the technology used in the manufacture, FPGAs can be divided into two groups: one-time configurable devices and reconfigurable devices [12]. A one-time configurable FPGA is mainly manufactured using fuse or anti-fuse technology. As its name indicates, this type of FPGA can be programmed only once, and the device will remain configured even when it powers off. On the reconfigurable side, static random access memory (SRAM), erasable programmable read-only memory (EPROM), and electrically erasable programmable read-only memory (EEPROM) are state-of-the-art technologies. Magnetic random access memory (MRAM) [13, 14] and ferroelectric random access memory (FeRAM) [15] have also emerged as interesting alternatives. In SRAM-based reconfigurable systems, configuration data is stored in external non-volatile memories, such as FLASH memories, and transferred into the SRAM during either boot-up or execution. Recently, single chip solutions for tightly integrating reconfigurable logic and FLASH memories have also been applied [16, 17, 18].

For SRAM-based technology, these devices consist of the circuit and the configuration-SRAM whose outputs are connected to the circuit and whose values continuously control the circuit. Reconfiguration is realized by altering the contents of the configuration-SRAM. This allows the circuit or a part of it to be reconfigured while the rest of the system is running [19]. Such a feature is referred to as run-time reconfiguration (RTR), and devices with such a feature are usually referred to as dynamically reconfigurable hardware (DRHW).

1.1 Run-time reconfigurable systems

Reconfigurability is becoming an important issue in the design of System-on-Chip (SoC) because of the increasing demands of silicon reuse, product upgrade after shipment and bug-fixing ability. There are many ways to realize reconfiguration, such as modifying software services [20] or changing the system structure [21] in a distributed environment. In the following context, we refer to run-time reconfigurable systems as those systems that include DRHW and achieve reconfiguration by modifying the design on DRHW at run-time.

1.1.1 Benefits of using run-time reconfigurable systems

Using DRHW has many advantages. DRHW can be used to improve the system performance. In many applications, the input data or the operating environment is varying all the time, which however cannot be decided at design time. Therefore, the ability to optimally or near-optimally adapt the system itself in order to gain higher speed and better performance according to the environment is highly favorable. For example, in a software-oriented application, a monitor can be set in the system to identify the currently most frequently used task, and then an equivalent hardware function can be dynamically generated and loaded onto DRHW to speed up the system [22, 23]. We can also use RTR to partially evaluate the system and replace generic circuits by more specialized circuits based on the run-time data [24]. It is also possible to reduce the power consumption by RTR. Some input data may require only a limited part of the circuit. Therefore, by eliminating the unnecessary circuit, the power dissipation can be significantly reduced [25].

Another advantage is that silicon reusability can be increased. As today's applications become more and more complex, the implementation needs more hardware resources. It means that either larger chips or more chips are needed, which might not be suitable for many products, such as portable devices that are required to have a limited dimension. With RTR, tasks that are non-overlapping either in time domain or in space domain can be mapped onto the same DRHW. Tasks that are required initially can be configured in the beginning. When another task is required, the configuration to load it can then be triggered. For example, in a typical software-defined radio (SDR) environment, different wireless technologies, such as GSM, WCDMA, WLAN and WiMax in the future, have to be supported. However, in most situations, these wireless technologies will be used at the same time. Therefore, it is possible to put them into a single DRHW and dynamically load the one that is needed.

In fact, SDR is a good environment for applying DRHW technology. Recent development of communication technology has brought huge challenges in the design of mobile devices. A demand for high data-rate services over wireless mobile devices has emerged, which requires such devices to have high processing capability. In addition, such devices also have to be able to handle the large amount of applications, such as audio/video streaming, teleconference and data encryption. On the other hand, mobile devices also require supporting multi-mode and multi-band. Additionally, new standards are being proposed for fast data-rate services, and mobile devices need to be flexible to accommodate them. Recently, the design cycle for mobile devices has been reduced to one year, or even less, which makes it infeasible to have a complete fixed-HW solution. DSP is also not a favorable solution, since it lacks enough processing capability and it is power-inefficient as well. Considering all these factors, DRHW is a reasonable technology alternative for SDR, since it can provide a combined benefit of flexibility and performance.

1.1.2 Challenges of using run-time reconfigurable systems

Current reconfiguration technologies have certain limitations, which result in some challenges when using RTR systems. One of the challenges is the configuration overhead related to each configuration process. It includes both configuration latency and configuration energy. A task needs to be loaded onto DRHW before it can be used. This is similar to the SW loading time in a pure SW system. The loading/configuration process takes time and energy, which can largely degrade the system performance.

Considering the commercial MPEG-4 simple profile decoder [26] that is implemented in Xilinx XC2V1500 FPGA [27], under the assumption that the configuration speed is 200 Mb/s the configuration latency is about 26 ms, which is considerably large since the decoding time per frame should be under 33 ms. It should be noted that such challenge exists only when DRHWs are used frequently. For example, if DRHW is reconfigured once a week, the configuration overhead can be ignored, because it is relatively very small compared to the task execution time.

In Figure 1, we compare several computing technologies (software on GPPs, hardware on ASICs, configurable computing on FPGAs, and RTR on DRHW) in terms of flexibility and performance. It can be seen that FPGAs have the advantage of simultaneous flexibility and performance. When considering the benefits of using RTR as presented in section 1.1.1, it is likely that DRHWs can be placed in the top-right corner, as shown in Figure 1, where additional flexibility and performance can be achieved over FPGAs. However, DRHWs also suffer from the configuration overhead, which might overrun the benefit and cause DRHWs to become a less competitive and less interesting technology, the one moving to the left represented by dotted cycle. Therefore, techniques to reduce the configuration overhead or configuration management approaches to reduce the effect of the configuration overhead are needed.

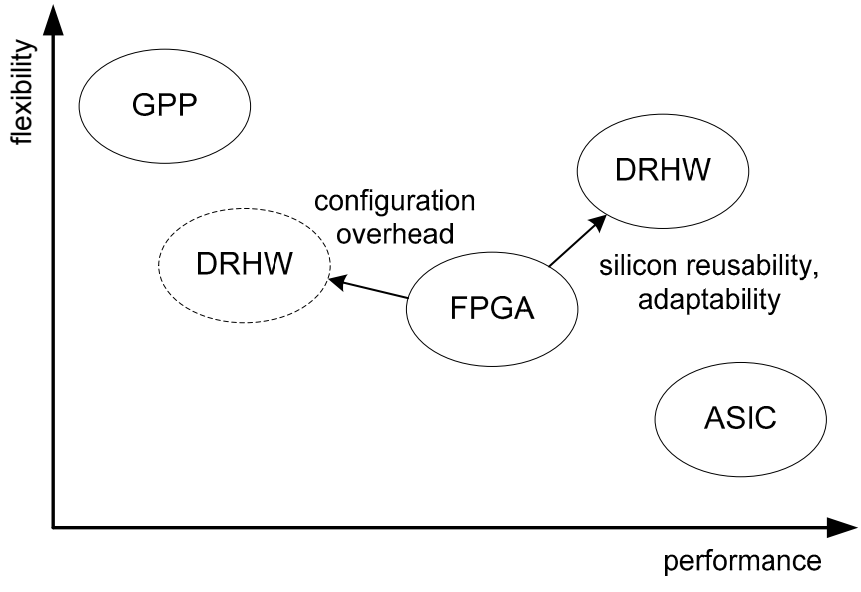


Figure 1. Comparison of different computing technologies in terms of performance and flexibility.

Another challenge of using RTR is the lack of supporting methodologies and tools at each abstraction level in the design phase. DRHW is a new type of computing element. At the system level, how to support and make system partitioning for not only software and hardware, but also for DRHW, and how to take the reconfiguration process into account during the design space exploration phase need to be studied. At the implementation level, approaches to handle the communication between DRHW and the rest of the system are needed and debugging techniques to validate the configuration process are missing.

1.2 Key contributions of the work

I have worked from different directions to meet the challenges of using RTR systems. From the design flow's point of view, I focus on system level design. Because any decision made at the system level might have significant impact on the final performance, providing useful supports at this level can help to eliminate unnecessary re-designs and therefore reduce the total design time. The ultimate goal is not to develop a fully automatic system partitioning approach, which I believe will not succeed. This is because applications and platforms nowadays are becoming so complex that it is not possible to quantitatively characterize them precisely in the early design phase so that complex mathematical formulas can be applied to fully partition the design in such a way that optimal solutions can be guaranteed. However, providing support to designers at this phase can help to prune the design space and possibly avoid re-designs. In my work, approaches to support system partitioning and DRHW modeling for fast design space exploration are provided. Considering the main bottleneck of using RTR systems, the configuration overhead, I have developed different static/run-time task scheduling techniques that can either reduce the effect of reconfiguration or reduce the amount of required reconfigurations. In addition, I also propose different techniques to physically reduce the configuration overhead.

1.2.1 System-level design flow and support tools

In the approach [28, 29], I focus on the type of RTR systems in which DRHW is frequently used as a coprocessor to accelerate computation-intensive tasks. The goal of the approach is to help designers in the early phase of the design process to easily evaluate the effect of moving some tasks, which are traditionally implemented in

fixed hardware, to DRHW. The supports that I provide are an estimation approach [30] and a modeling technique [31] for DRHW. The estimation approach starts from function blocks represented in ANSI-C language, and it produces hardware execution time and resource utilization estimates for each function block by applying a set of high-level synthesis algorithms. In the DRHW modeling, the real reconfiguration process is not modeled. Instead, its behavior is modeled, and then its effect (the configuration overhead) can be reviewed during simulation. A number of parameters are provided in DRHW models. Designers can tune them to target a particular type of reconfiguration technology. In addition, a tool to automatically generate DRHW models is created in order to reduce the coding effort.

1.2.2 Scheduling techniques to manage the configuration process

The configuration overhead is a bottleneck that might largely degrade the performance of RTR systems. In this work, I present several task scheduling techniques to minimize its effect. They are grouped into a quasi-static scheduling framework. It is divided into design-time scheduling and run-time scheduling. Tasks with known dependencies are scheduled at design time. The goal is to reduce the total schedule length while taking the reconfiguration processes into account. Three static scheduling algorithms, using different problem-solving strategies, are developed and quantitatively evaluated [32]. The first is a list-based scheduler [33], which uses a heuristic approach. The second explores the entire domain for searching optimal solutions [34]. The last uses a guided random search strategy [35] that tries to balance accuracy and efficiency. At run-time, the focus of the scheduler is to maximally reuse loaded tasks with a locking technique [36] similar to cache-locking. The idea is to monitor the tasks' execution status at run-time and always lock the most frequently used tasks on DRHW in order to reduce the total number of required reconfigurations.

1.2.3 Techniques to reduce the configuration overhead

Directly reducing the configuration overhead would be a more straightforward mechanism to increase the efficiency of RTR systems. In this work, I also present two separate novel approaches for reducing the overhead, one for configuration latency [33, 37] and another for configuration energy [38]. In fact, in the first approach configuration latency is not physically reduced, but its effect is reduced by

performing several configurations in parallel. The technique is referred to as configuration parallelism. The idea is to divide the entire configuration-SRAM into several individual segments and allow them to be accessed simultaneously. Therefore, configurations can be performed in parallel, which allows task parallelism to be better exploited. In the second approach, configuration energy is reduced by applying the dynamic voltage scaling (DVS) technique. The idea is to apply low supply voltage on the reconfiguration process when possible in order to reduce the configuration energy but without increasing the overall schedule length.

1.3 Introduction to the most important papers

This thesis is based on one international scientific journal and 10 international scientific conference papers. I am the first author of all these scientific publications and the key contributor behind all these works.

Papers [28, 29] describe the system-level design flow and an instantiation using a practical case study. I have contributed to defining the overall design flow, and I carried out the implementation work of the case study. More importantly, I have developed the HW estimators and the SystemC code transformer. They are described in detail in [30, 31].

Papers [32, 34, 35] describe the different static DRHW task scheduling algorithms, their implementation and the evaluation results. My contribution was defining, developing and evaluating all these algorithms. Paper [36] describes the configuration locking technique for improving the system performance. My contribution was inventing the idea, implementing the run-time scheduler and the locking algorithm, and evaluating the results.

Papers [33, 37] describe the parallel configuration model for reducing the impact of the configuration latency. I invented the idea and the evaluation approach, and I also implemented the algorithms and evaluated the results. Paper [38] describes the technique to reduce the configuration energy. The idea is to apply dynamic voltage scaling on the configuration processes as long as it does not harm the schedule length. My contribution was inventing the idea and implementing the evaluation approach.

1.4 Organization of the thesis

The thesis is organized as follows. Background and a review of related works are presented in Chapter 2. The system-level design supports are presented in Chapter 3. The three static scheduling techniques and the run-time scheduling technique are presented in Chapter 4, and the two novel techniques for reducing the configuration overhead are presented in Chapter 5. Finally, conclusions and discussions of future work are presented in Chapter 6.

2. Background and related work

In this chapter, we briefly introduce the fundamentals of reconfigurable computing as well as some state-of-the-art technologies and reconfigurable systems. Instead of providing a complete survey of reconfigurable computing, we concentrate only on the research related to our work. Exhaustive surveys can be found in [19, 39, 40, 41, 42].

2.1 Run-time reconfigurable computing

In this section, we first introduce how reconfigurable systems can be categorized. In fact, there are various ways to categorize them based on different criteria, such as configuration granularity, configuration style and coupling techniques. Then, some existing typical RTR systems will be described.

2.1.1 Configuration models

The principle to achieve RTR is by dynamically altering the design on DRHW. According to the granularity of the processing element, DRHW can be classified as fine-grained or coarse-grained.

- Fine-grained DRHW: This type of DRHW is very similar to most of the traditional FPGAs, in which the basic configurable LE consists of a look-up table (LUT) and a flip-flop. The LUT usually have four input bits and one output bit, which can be implemented as any combinational logic of the four input bits. LEs are connected to each other also at bit level. Therefore, this type of granularity can be very suitable for bit-level computations but might not be efficient for computing word-width data.
- Coarse-grained DRHW: In this type of DRHW, the basic computing element is optimally designed for large computations, and such DRHW is primarily intended for implementing tasks that are dominated by word-width operations, which can be more efficiently performed in terms of both area and time when compared to fine-grained DRHW that is constructed from much smaller computation elements. However, the structure of each processing element is static, which makes such architecture unable to leverage optimizations in the size of operands. Such a model has been widely adopted in various systems [43, 44, 45].

The majority of DRHW devices are using SRAM-based technology. Such devices consist of the circuit and the configuration-SRAM whose values specify how the circuit is programmed. Reconfiguration is realized by altering the contents of the configuration-SRAM. Traditionally, configurations are operated in an off-line fashion. The configuration is performed only once when the system starts up, and the circuit remains unchanged until the system is powered off. Recently, techniques to modify the configuration-SRAM at run-time have been applied in commercial devices. This enables RTR, which can significantly improve the flexibility when compared to the traditional simple configuration approach. In terms of the reconfiguration methods DRHW can be divided into three main categories.

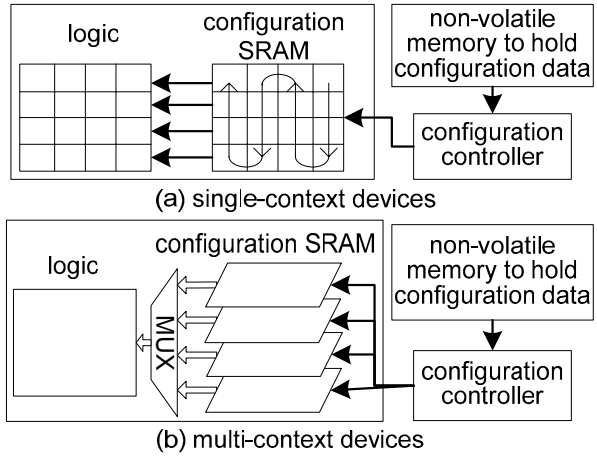


Figure 2. Different configuration models.

- Single-context DRHW: In this configuration model, as shown in Figure 2(a), the configuration-SRAM can be viewed as a large shift register, and the entire configuration-SRAM has to be written during one reconfiguration. This means that even if only a part of the chip needs to be changed, the entire chip needs to be reconfigured. Therefore, this model suffers from significant configuration overhead. Most modern FPGAs deploy this approach.
- Multi-context DRHW: Such devices [46, 47] have multiple configuration planes and a multiplexer (MUX) controls which plane is used, as shown in Figure 2(b). Switching from one to another takes only one clock cycle, thus configuration latency is significantly reduced. However, such devices require as many times the configuration-SRAM needed in single context device as the number of contexts, and caching tasks on the contexts consumes a large amount of static energy.

- Partially reconfigurable DRHW [27]: They have a similar configuration model as in single-context devices, but the configuration-SRAM is constructed as RAM. Therefore, it can selectively change the content of a portion of the configuration-SRAM. Thus, configuration of a task takes a much shorter time.

Basically, the partial reconfiguration model can be further divided into one-dimension (1D) model and two-dimension (2D) model. In both cases, the device is represented by a rectangular area of dynamically reconfigurable elements. The 1D model describes a device as a number of columns. The height of a column is fixed and equal to the height of the device area. The width of a column can either be fixed or dependent on task sizes. I/O mechanism and communications are usually implemented in the top/bottom of columns. The sides of column can also be used if necessary. A typical system that instantiates such a 1D model is the Erlangen Slot Machine [48, 49, 50]. One particular type of 1D partial reconfiguration is pipeline reconfiguration [51, 52], which is suitable for implementing pipelined applications. In this model, devices consist of a number of hardware stripes connected in a pipeline fashion, and the basic reconfiguration unit is a stripe. In the idea case, one stripe can implement one pipeline stage of the application. PipeRench [53, 54, 55, 56] is a typical striped DRHW. The 2D model allows tasks to be placed anywhere. Shapes of tasks can be either rectangular or irregular, which imply higher device utilization if fragmentation can be properly handled. Fragmentation on DRHW means that there are quantitatively enough free resources for a ready task, but the task cannot be placed because either the free resources are not continuous or they do not match the shape of the task. In the 2D model, communication is an unsolved problem, because interconnections between tasks are difficult to be re-established when tasks can be freely relocated. On-chip just-in-time routing techniques have been proposed [57, 58] which can solve the connection problem, but the overhead is in the range of seconds to minutes.

2.1.2 Coupling techniques

In most of the RTR systems developed nowadays, DRHW is used in association with a host processor in a way that computation-intensive operations are mapped onto it, and operations being relatively rarely executed are mapped onto the host processor. Depending on how closely DRHW and the microprocessor work together, there are several ways to make the coupling.

- The closest coupling is to integrate DRHW into the host processor as a special function unit to execute custom instructions in order to boost performance. DRHW behaves just as other function units residing inside the processors and is triggered when custom instructions are found during the instruction decoding phase. Such devices are referred to as reconfigurable instruction-set processors (RISPs) [59]. This coupling technique has been used in various systems [60, 61, 62, 63].
- DRHW can be used as a co-processing unit [64, 65, 66, 67], which runs independently of the host processor. Usually, the co-processing unit and the host processor are put onto the same die, and the memory caches are shared between both. Compared to the RISP, which performs only one custom instruction at a time and has to communicate with the host processor whenever a custom instruction is used, the reconfigurable logic in this type of coupling may perform relatively a larger amount of computation at a time and communicate with the host processor less frequently.
- The most loosely coupled form is to use reconfigurable units as attached processing units or standalone-processing units [68]. They behave as either an additional processor or a network-connected workstation.

2.1.3 Example systems

FPGAs are the most widely used reconfiguration technologies due to the more than 30 years of continuous improvement they have undergone. Technology advances keep increasing the densities of logic and embedded memory of FPGAs. In addition, various fixed logic modules, such as hardwired multipliers and high speed transceivers, as well as microprocessors have also been integrated. This makes it possible to implement a very complex system into a single FPGA. The most advanced FPGA platforms are Xilinx Virtex-5 [69] and Altera Stratix-III [70]. Both are manufactured as 65nm technology and able to run designs at 500 MHz. Although there are a few embedded DSP blocks for word-width operations, such FPGA platforms are still referred to as fine-grained devices, since LUTs and connecting switches are still the main configurable elements.

PACT XPP coprocessors [71] can typically represent coarse-grained technologies. The heart of the XPP is an array of configurable processing array elements (PAEs), which can perform most of the typical DSP functions, such as multiplication, addition,

shift and comparison, within one clock cycle. The communication network allows both point-to-point and point-to-multipoint connections. Only the opcodes of PAEs and the routing channels need to be configured, thus such devices do not suffer from the configuration overhead. In addition, each PAE can be individually reconfigured, thus partial reconfiguration is also supported.

The MorphoSys architecture [72, 73] is an integrated coarse-grain multi-context RTR system, in which an 8x8 Reconfigurable Cell (RC) array is closely coupled to a host processor. When a context is being used, configuration can be loaded to another context simultaneously. The communication between the host processor and the RC array is realized through a frame buffer, which consists of two sets (two banks in each set). The RC array can simultaneously access both of the two banks in one set, and at the same time the data can be moved from external memories to the other set. Data movement and reconfigurations are controlled by the host processor.

The Atmel's Field Programmable System Level Integrated Circuits (FPSLIC) family [74] is another typical reconfigurable system, which consists of a host processor and a fine-grained FPGA that are closely coupled. The FPGA shares the data memory with the host processor (protection mechanism is needed if data sharing is used), and it is treated as a normal 8-bit peripheral from the processor point of view. The FPGA can be divided into a dynamically reconfigurable region and a static region that is required to implement certain system functions.

2.2 System-level design techniques

System-level design covers various issues, such as partitioning, task scheduling and synthesis. Including DRHW in the design requires the traditional HW/SW co-design flow to be extended. In [75, 76], a co-design framework and a HW/SW partitioning approach are presented. The approach uses a list-based algorithm to create an initial partition then gradually moves a task from SW to DRHW in an iterative way. In [77], a survey of various SW/HW partitioning algorithms is presented, and a new approach to map loops into reconfigurable hardware is proposed. In [78], a co-design environment for DSPs/FPGAs-based reconfigurable platforms is presented. Both applications and architectures are modeled as graphs, and an academic system-level CAD tool is used. In [79], a macro-based compilation framework to replace logic synthesis and technology mapping is presented. In [80], a synthesis approach based on list-scheduling is presented.

The target system is a single application that can be divided into a number of dependent tasks. The approach considers HW/SW partitioning, temporal partitioning as well as context scheduling. In [81, 82], a HW/SW co-synthesis framework for a real-time embedded reconfigurable system is presented. Each application consists of a number of dependent tasks and has its own period. A task can be mapped either onto a host processor or DRHW. Applications are statically scheduled over a hyper-period, which is the least common multiple of the different application periods. Design frameworks that start from high abstraction level models have also been proposed. An approach to automatically perform the mapping of the platform independent models presented in UML onto reconfigurable system models is developed in the MOCCA design environment (Model Compiler for Configurable Architectures) [83]. In [84], researchers presented their AEP (Abstract Execution Platform) virtual machine, which executes the binary UML representation and implements the design onto a reconfigurable platform.

2.3 Configuration management techniques

The configuration overhead is the main drawback of RTR systems. Various research projects have tried to solve the problem. These efforts can be divided into three groups. The first group of these techniques is to reduce the amount of required configuration data in each reconfiguration process. The second group is to reduce the number of required configurations. This is suitable for those applications which require running some tasks repetitively. The last group is to take the configuration process into account during the task scheduling in order to reduce its effect. All of these techniques mainly focus on reducing configuration latency. However, for the second group, reducing the number of required configurations can also result in less dynamic configuration energy.

2.3.1 Reducing the configuration data

Depending on the configuration types of DRHW, configuration latency has different impacts. For single context devices, because each reconfiguration requires re-writing the entire configuration-SRAM at one time, configuration latency is simply related to the device size. For partially reconfigurable devices, configuration latency is directly proportional to the occupied area of applications. Therefore, less configuration

latency is needed. In [85], a case study of a 3-tap filter on a real platform shows that using partial reconfiguration can reduce 50% of configuration latency. However, as applications become more complex and designs become much larger, configuration latency is becoming larger as well and both types of DRHW will suffer from the long configuration latency. For multi-context devices, configuration latency can be as short as one clock cycle if the required context is already loaded. However, it is not efficient to implement a device with a large number of context memories, which require large space and consume significant amount of static power. Some existing multi-context devices hold only four contexts [86], and loading configuration data from external memories still has an unignorable impact.

A straightforward approach to reducing configuration latency is to compress the amount of configuration data to be transferred. In fact, a lot of redundant information and regularities exist in the configuration bitstream. Thus, lossless compression techniques can take advantage of these and remove the unnecessary bits. In [87], a technique that uses entropy, inspired by information theory, is proposed to evaluate configuration compression performance. The entropies of some benchmark circuits are calculated to provide estimates of the possible reduction of configuration data sizes. In [88, 89], various compression techniques, such as Huffman coding, Arithmetic coding and LZW compression, are applied on configuration compression. Other approaches, such as using a genetic algorithm [90] or using runlength coding [91], have also been proposed. All these approaches require the development of specific hardware in the device side to decompress the data. In [92], an approach that exploits the existing embedded decompression hardware in a specific type of FPGA is proposed to reduce the amount of configuration data. In addition, there are some approaches that try to exploit the similarities between successive configurations [93, 94, 95]. The idea is that if the same components are used in successive configuration, then placing them in the same location can avoid loading redundant configuration data.

Another approach to reducing configuration data is by fundamentally preventing configuration data from being large. The reason that fine-grained DRHW, such as FPGA, suffers from large configuration data is because such devices are programmed at bit level. Each LUT and each route need to be programmed, which require lots of bits for recording such detailed configuration information. In contrast, coarse-grained devices are based on word-level units, which are implemented in a fixed style but support limited programmability with fewer control bits. Therefore, using coarse-grained devices can impose less configuration latency [96].

2.3.2 Reducing the number of required configurations

In a practical scenario, it is likely that some tasks need to run repetitively. Therefore, retaining the configurations of these tasks on DRHW can reduce the amount of configuration data to be transferred. This technique is referred to as configuration caching [97], which is similar to data caching and instruction caching in a general memory. In [98], different caching algorithms targeting various device models, such as single-context DRHW and multi-context DRHW, are studied. The principle is as follows. For an RTR system, a design can be divided into a number of small blocks with a known execution sequence. Blocks are put together into groups, and each group can fit in the available hardware. Reconfiguration is then corresponding to the transition from one block to another when the two blocks do not belong to the same group. Therefore, by optimally grouping the blocks, the number of required reconfigurations can be minimized and thus the total configuration latency is reduced. In [99], the grouping approach is extended for supporting multiple tasks that are from a single application but have a non-deterministic execution order.

Configuration relocation and defragmentation [100, 101] can also help to reduce the number of required reconfigurations. This is suitable for only partially reconfigurable devices, and usually the 2D reconfiguration model is the target because fragmentation is the main concern in such a model. The main idea of this technique is as follows. When fragmentation happens, placing a task on the device will cause another task or a part of it to be evicted, which requires the evicted task to be reconfigured in the future when it is needed. By dynamically relocating valid configurations into new locations, free area can be consolidated and then used for new tasks. This defragmentation technique can result in more efficient use of DRHW and fewer reconfigurations. In [100, 101], an architecture that supports run-time relocation and defragmentation is proposed, in which shapes of tasks do not need to be rectangular. Resource compaction is realized by transforming tasks through a series of techniques, such as rotation, flipping and offsetting. In [102, 103], heuristic approaches for run-time repacking are studied. The procedure is divided into two steps. The first step identifies how to rearrange the loaded tasks in order to free sufficient space for a waiting task, and the second step focuses on how to move the tasks in order to allocate the waiting task as quickly as possible. In [104], an efficient run-time compaction technique for the 1D model is presented. The technique allows multiple columns of different tasks to be shifted in parallel, and thus the run-time overhead can be independent of the current device size.

Various other defragmentation techniques or related issues have also been proposed or discussed. Optimal solutions using a branch-and-bound technique [105] and heuristic approaches [106, 107, 108] for defragmentation are studied. In [109, 110, 111], implementation techniques for task relocation are presented. The idea is to replicate the corresponding configurable elements and routing resources. In [112], a technique to study the impact of defragmentation on system performance is presented. It proposes to use a fragmentation metric as the basis to guide the defragmentation process. In [113, 114], explicit defragmentation is not performed, but different fragmentation metrics are used in the task allocation phase to choose the more suitable locations for coming tasks.

2.3.3 Managing reconfigurations in the task scheduling process

When taking configuration into account in the task scheduling process, the effect of configuration latency can be effectively minimized if the scheduling can be done properly. One effective technique that has been widely used in RTR system scheduling is configuration prefetching [115]. The basic idea is to load tasks before they are needed. Therefore, by overlapping configurations with execution of other useful tasks, the effect of configuration latency can be reduced. Different prefetch techniques for single thread applications have been proposed. Hauck presents a prefetch scheduler [115] that can properly insert prefetch instructions into software applications to hide the configuration of single-context reconfigurable coprocessors. Targeting a partial reconfiguration model, three different configuration prefetching algorithms are studied [116]. The first is static prefetching, extended from their earlier work [115] to support loading multiple blocks. The second is dynamic scheduling, which models the system state as a Markov process and uses it to predict the next configuration to prefetch. The last one is a hybrid prefetching approach, which uses the recent execution history to make accurate predictions within a loop and uses the global history to make accurate predictions between loops.

For systems that consist of multiple independent tasks, configuration prefetching techniques have also been applied. In [117, 118, 119], Resano et al. consider that each task can be further divided into a number of dependent subtasks, and prefetch scheduling is performed dynamically at the subtask level. Although drastic reduction of the total configuration latency can be achieved, the run-time overhead of the approach prevents it from being used on a large scale. In [120, 121], Resano et al.

extend their run-time scheduler by dividing computation into design time and run-time. At design time, the objective is to calculate weights of subtasks. The higher the weight value of a subtask, the more negative impact its configuration has on the system performance. At run-time, the scheduler initially loads subtasks based on their weight values and dynamically makes other reconfiguration decisions based on the current status.

There are compiler-driven task schedulers. In [122, 123], based on the Molen programming paradigm [124, 125, 126], Panainte and co-workers present an instruction scheduler that can reduce the number of required configurations by moving the configuration instruction out of the loop. Therefore, the task inside a loop needs only one configuration, if it is mapped onto DRHW. In [127], an improved instruction scheduler is presented, which can further reduce the number of required reconfigurations by caching the most frequently reconfigured tasks on DRHW. The decision is made at design time with help of profiling the applications. In [128, 129, 130], a compilation framework for MorphoSys is described. The core is a kernel scheduler that considers task scheduling, multi-context scheduling, allocation and data transfer simultaneously with the goal to minimize the reconfiguration impact.

In [81, 82], a scheduler for a real-time embedded system is presented. Each application consists of a number of dependent tasks and has its own period. A task can be mapped either onto a host processor or DRHW. Instead of performing the scheduling at run-time, it statically schedules the applications over a hyper-period, which is the least common multiple of the different application periods. The 1D model is used for DRHW. During run-time, each tile has its own value that represents the reconfiguration frequency of the task mapped onto this tile. The frequency is equal to the number of times that task is executed in a hyper-period. When a new reconfiguration for a task of m tiles needs to flush some already loaded tasks, the m continuously connected tiles that have the minimum sum of the reconfiguration frequencies are selected. The idea is to leave the most frequently reconfigured tiles intact because they might be reused soon.

Different techniques for static task scheduling have also been discussed. In [131], an optimal placer for mapping tasks onto 2D DRHW is presented. A task is treated as a three dimensional box in space and time. The problem is then converted into a box packing problem. Configuration overhead is treated as a constant and added to the execution time. Since configurations are not separately treated, the approach does not

consider prefetching or the reconfiguration constraints. (There is a limited number of allowed simultaneous reconfigurations for different regions.) Although the approach requires a relatively short computer run-time, its simplified model significantly reduces the liability of the approach. Approaches derived from Model-checking [132] and Petri Net [133] for static task scheduling have also been proposed.

Different run-time scheduling techniques have been proposed to manage tasks whose precedence dependence is not known at design time. In [103, 134], Dissel et al. take into account the resource fragmentation problem. Different techniques for repacking and replacing loaded modules are proposed. In [113, 135], the online task scheduling for 2D DRHW is modeled as a bin-packing problem, and different algorithms working with efficient data structures are proposed for solving the NP-hard problem. Techniques to find free area in 2D DRHW for task placement have also been presented [136, 137]. In [138], operating system services for scheduling real-time tasks on DRHW are presented. The difference from other approaches is that an incoming task is either accepted with a guarantee of meeting its deadline or rejected. In [139, 140], an online scheduling algorithm, which adapts the well-known single processor earliest-deadline first (EDF) policy, is presented. However, the configuration process is not individually considered, as in [131]. Therefore, practical applications will not be able to achieve the reported scheduling performance.

3. System-level design supports for run-time reconfigurable systems

As presented in earlier chapters, reconfigurable logic is a promising alternative to deliver both flexibility and performance at the same time. New reconfigurable technologies and technology-dependent tools have been developed, but a system-level design method to support system analysis and fast design space exploration is missing. In this chapter, we present a system-level design method and supporting tools for the design of reconfigurable SoC (RSoC). An instantiation of the design flow is applied in a real case study. At the implementation level, commercial technology-dependent tools are used. At the system level, one of our supporting tools is a high-level synthesis-based HW estimator. It can generate HW estimates directly from ANSI-C code and thus help designers to make reasonable partitioning decisions. Another supporting tool is a SystemC code transformer, which can automatically generate a SystemC model of DRHW from existing SystemC code of the functions that are to be mapped onto the DRHW. Therefore, designers can generate different versions of reconfigurable systems without rewriting the code, which is slow and error-prone. The main advantage of the approach is that it can be easily embedded into an SoC design flow to allow fast design space exploration for different reconfiguration alternatives without going into implementation details.

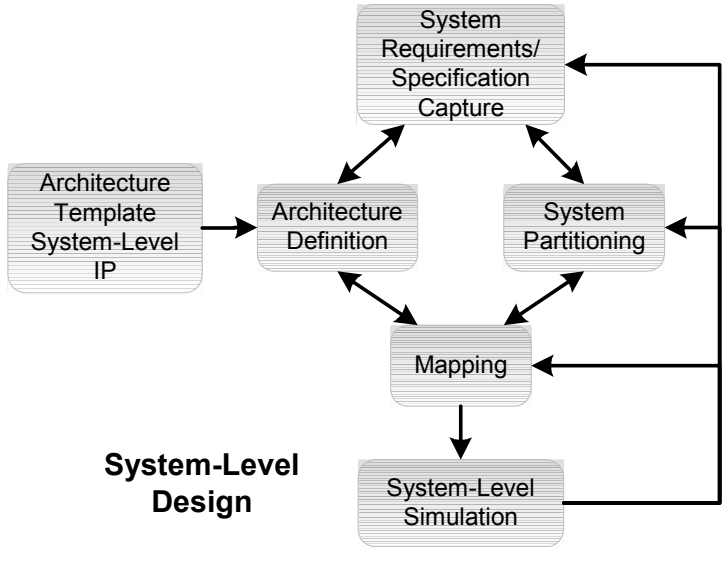


Figure 3. A generic system-level design flow.

3.1 System-level design flow and our supports

A generic view of the system-level design flow is depicted in Figure 3 [141]. The following new features are identified in each phase when reconfigurability is taken into account:

- *System Requirements and Specification Capture* needs to identify requirements and goals of reconfigurability (e.g., flexibility for specification changes and performance scalability).
- *Architecture Definition* needs to model the reconfigurable technologies of different types and vendors at an abstract level and include them in the architecture models.
- *System Partitioning* needs to analyze and estimate the functions of the application for SW, fixed HW and DRHW. The parts of the targeted system that will be realized on DRHW must be identified. There are some rules of thumb that can be applied. If an application has hardware accelerators of roughly the same size which are not used at the same time, these accelerators can be implemented onto DRHW. If an application has some parts in which specification changes are foreseeable or there are foreseeable plans for new generations of the applications, it may be beneficial to implement it onto DRHW.
- *Mapping* needs to map functions allocated to DRHW onto the respective architecture model. The special concerns at this step are the temporal allocation and the scheduling problem. Allocation and scheduling algorithms need to be made either online or offline.
- *System-Level Simulation* needs to observe the performance impacts of architecture and reconfigurable resources for a particular system function. The effect of configuration overhead should be highlighted in order to support designers to perform system analysis or design space exploration.

It should be noted that reconfigurability does not appear as an isolated phenomenon, but as a tightly connected part of the overall SoC design flow. Our approach is therefore not intended to be a universal solution to support the design of any reconfigurability. Instead, we focus on a case where the reconfigurable components are mainly used as co-processors in SoCs.

SystemC language version 2.0 [142] is selected as the backbone of the approach since it is a standard language that provides designers with basic mechanisms like channels, interfaces and events to model various kinds of communication and synchronization styles in system designs. More sophisticated mechanisms for the system-level design can be built on top of the basic constructs. More specifically, our system-level models operate on the transaction level of abstraction. The performance simulation is based on the estimates of computational complexity of each block, estimates of communication and storage capacity requirements, and characteristics of the architecture and mapped workload. In fact, our design approach is not limited to SystemC. It can also be applied to other promising design languages, e.g., SystemVerilog [143] and SpecC [144].

In the SystemC-based approach, we assume that the design does not start from scratch, but it is a more advanced version of an existing device. The new architecture is defined partly based on the existing architecture and partly using the system specification as input. The initial architecture is often dependent on many things not directly resulted from the requirements of the application. The company may have experience and tools for certain processor core or semiconductor technology, which restricts the design space and may produce an initial HW/SW partition. Therefore, the initial architecture and the HW/SW partition are often given at the beginning of system-level design. The SystemC extension is designed to work with a SystemC model of the existing device to suit the design considering RTR hardware.

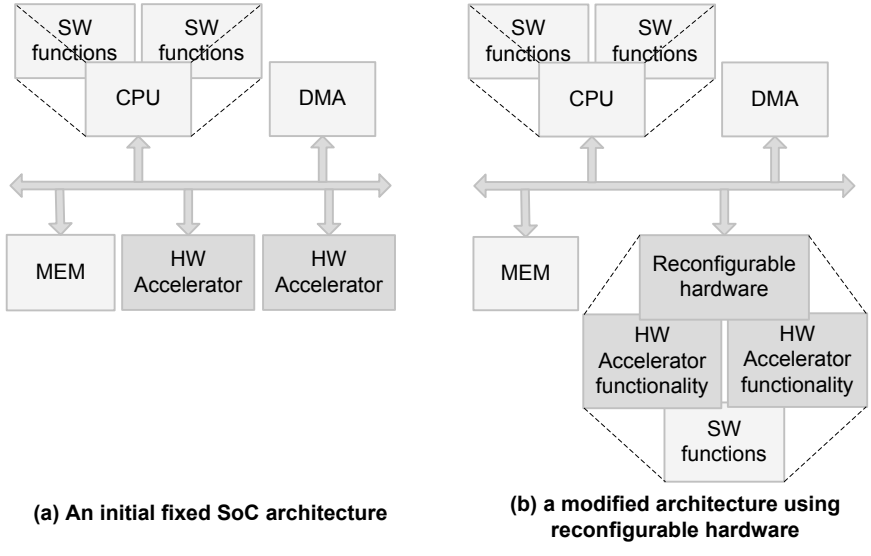


Figure 4. Creating reconfigurable SoC from fixed platform.

The way that the SystemC-based approach incorporates dynamically reconfigurable parts into architecture is to replace SystemC models of some hardware accelerators, as shown in Figure 4(a), with a single SystemC model of a reconfigurable block, as shown in Figure 4(b). In addition, functions that are mapped onto fixed HW accelerators are modeled in the same way as in DRHW. Therefore, it allows designers to easily test the effects of implementing some components in DRHW. Referring to the system-level design flow, as shown in Figure 3, we provide estimation support for system partitioning, scheduling support for mapping and modeling support for system-level simulation. These three steps are the most critical steps in the system-level design, because they produce the direct inputs to the low-level implementation. Their accuracy and efficiency in design space exploration have a strong impact on the overall design efficiency and time-to-market. The scheduling support is described in detail in the next chapter. Others, as in the list below, are described in the following sections.

- Estimation and analysis support for design space exploration and system partitioning [30].
- Reconfigurability modeling using standard mechanisms of SystemC and a transformation tool to automatically generate SystemC models of the reconfigurable hardware [31].

3.1.1 Definition of terms

The terms and concepts specific to the SystemC-based approach used in the rest of the paper are defined as follows:

- Candidate Components: Candidate components denote those application functions that are considered to gain benefits from their implementation on a reconfigurable hardware resource. The decision whether a task should be a candidate component is clearly application dependent. The criterion is that the task should have two features in combination: flexibility (that would exclude an ASIC implementation) and high computational complexity (that would exclude a software implementation). Flexibility may come either from the point that the task will be upgraded in the future or in view of hardware resource sharing with other tasks with non-overlapping lifetimes for global area optimization.

- Dynamically reconfigurable fabric (DRCF): The dynamically reconfigurable fabric is a system-level concept that represents a set of candidate components and the required reconfiguration support functions, which later on in the design process will be implemented on DRHW.
- DRCF component: The DRCF component is a transaction-level SystemC module of the DRCF. It consists of functions which mimic the reconfiguration process, and the instances of SystemC modules of the candidate components to present their functionality during system-level simulation. It can automatically detect reconfiguration requests and trigger the reconfiguration process when necessary.
- DRCF template: The DRCF template is an incomplete SystemC module, from which the DRCF component is created.

3.1.2 Estimation approach to support system analysis

System analysis is mainly the phase to make HW/SW partitioning and the initial architecture decision. In the design of reconfigurable SoC, system analysis also needs to focus on studying the trade-off of performance and flexibility. The estimation approach is developed to support system analysis to identify candidate components that are to be implemented on DRHW. In addition, the approach produces SW/HW performance estimates, so it can also be used for supporting SW/HW partitioning.

The estimation approach focuses on a reconfigurable architecture in which there is a reduced instruction-set computer (RISC) processor and an FPGA-type DRHW, connected by a communication channel, a system bus. The current FPGA-type DRHW is assumed to be a Virtex2-like FPGA [27] in which the main resources are LookUp-Tables (LUTs) and multipliers. The estimation approach starts from function blocks represented using C language, and it can produce the following estimates for each function block: software execution time in terms of running the function on the RISC core, mappability of the function and the RISC core [145], execution time in terms of running the function on DRHW, and resource utilization of DRHW. The framework of the estimation approach is shown in Figure 5. The starting point is the functional description given in ANSI-C language. The designer decides the granularity of partitioning by decomposing the algorithm down to function blocks. A single function block may then be assigned to SW, DRHW or a fixed accelerator.

Each of the function blocks will be individually studied and the set of estimation information will be fed into the system-level partitioning phase.

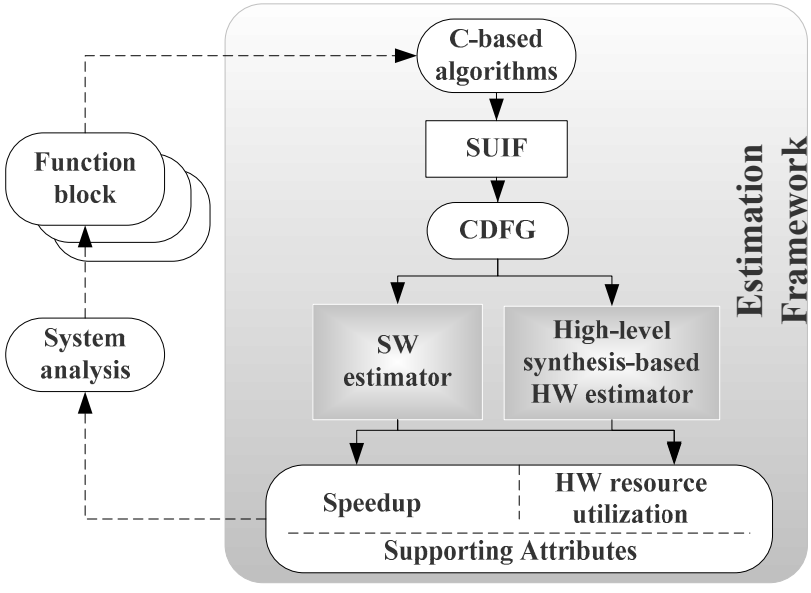


Figure 5. The estimation framework.

3.1.2.1 Creation of control-data flow graph from C code

We first transform C code into control-data flow graph (CDFG), which is a combined representation of data flow graphs (DFGs) and a control flow graph (CFG). The DFGs expose the data dependence of algorithms, and the CFG captures the control relation of a group of DFGs. In the estimator, we extend the SUIF front-end C compiler environment [146] to extract CDFG from the C code. The first step is to dismantle all high-level loops (e.g., while loop and for loop) into low-level jump statements. The produced code is restructured to minimize the number of jumps. Then, basic blocks are extracted. A basic block contains only sequential statements without any jump in between. Data dependence inside each basic block is analyzed, and a DFG is generated for each basic block. After the creation of all DFGs, the control dependence between these DFGs is extracted from the jump statements to construct the CDFG. Finally, profiling results, which are derived using gcov [147], are inserted into the CDFG as attributes. In later steps, these profiling results are used to produce the timing estimates.

3.1.2.2 High-level synthesis-based hardware estimation

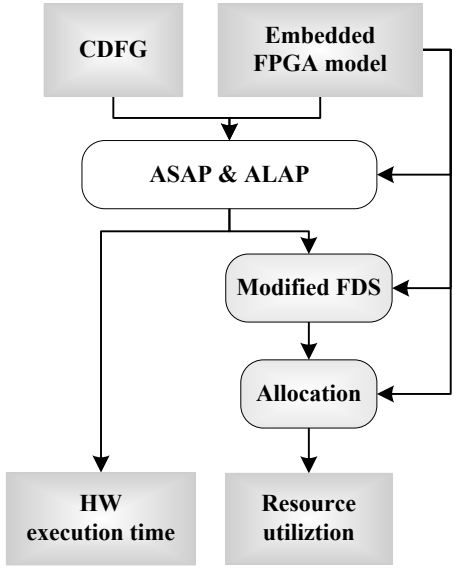


Figure 6. High-level synthesis-based hardware estimation.

A graphical view of the hardware estimation is shown in Figure 6. Taking the CDFG with the corresponding profiling information and a model of embedded FPGA as inputs, the hardware estimator carries out a high-level synthesis-based approach to produce the estimates. The main tasks performed in the hardware estimator as well as in a real high-level synthesis tool are scheduling and allocation. Scheduling is the process in which each operator is scheduled in a certain control step, which is usually a single clock cycle, or in several control steps if it is a multi-cycle operator. Allocation is the process in which each representative in the CDFG is mapped to a physical unit, such as variables to registers, and the interconnection of physical units is established.

The embedded FPGA is viewed as a co-processing unit, which can independently perform a large amount of computation without constant supervision of the RISC processor. The basic construction units of the embedded FPGA are static random access memory (SRAM)-based look-up tables (LUT) and certain types of specialized function units, e.g., custom-designed multiplier. Routing resources and their capacity are not taken into account. The embedded FPGA model is actually a mapping table. The index of the table is the type of operators, e.g., addition. The value mapped to each index is hardware resources in terms of the number of LUTs and the number of specialized units,

such as hardwired multipliers. In addition, the execution time of an operator is also an attribute. This table is generated by synthesizing each type of operation onto the target FPGA. The timing information of each function unit is used during the scheduling step, and the resource information is used when generating the resource estimates.

As-soon-as-possible (ASAP) scheduling and as-late-as-possible (ALAP) scheduling [148] determine the critical paths of the DFGs, which together with the control relation of the CFGs are used to produce the estimate of hardware execution time. For each operator, the ASAP and ALAP scheduling processes also set the range of clock cycles within which it could be legally scheduled without delaying the critical path. These results are required in the next scheduling process, a modified version of force-directed-scheduling (FDS) [149], which intends to reduce the number of function units, registers and buses required by balancing the concurrency of the operations assigned to them without lengthening the total execution time. The modified FDS is used to estimate the hardware resources required for function units.

Finally, allocation is used to estimate the hardware resources required for interconnection of function units. The work of allocation is divided into three parts: register allocation, operation assignment and interconnection binding. In register allocation, each variable is assigned to a certain register. In operation assignment, each operator is assigned to a certain function unit. Both are solved using the weighted-bipartite algorithm, and the common objective is that each assignment should introduce the least number of interconnection units that will be determined in the last phase, the interconnection binding. In this approach, multiplexer is assumed to be the only type of interconnection unit. The number and type of multiplexers can be easily determined by simply counting the number of different inputs to each register and each function unit. After allocation, the clock frequency is determined by searching for the longest path between two registers. Because routing resources are not modeled, the delay takes into account only the function units and the multiplexers.

We assume that all variables have the same size, since our goal is to quickly produce estimates with pure ANSI-C code instead of generating optimal synthesizable RTL code, which often uses some kinds of subset C code and applies special meanings to variables. Our estimation framework also supports exploring parallelism for loops. This is done at the SUIF-level, where we provide a module that allows designers to perform loop unrolling (loops must have a fixed number of iterations) and loop merging (loops must have the same number of iterations).

3.1.2.3 Estimating SW execution time

The software estimator uses a profile-directed operation-counting based static technique to estimate software execution time. The architecture of the target processor core is not taken into account in the timing analysis. The main idea of estimating the software execution time is as following. Firstly, the number of operations of each type is counted from the CDFG. Then, each type of operation node in the CDFG is mapped to one or a set of instructions of the target processor in a pre-defined manner. Then the total number of instructions is calculated by summing up the number of estimated instructions of each type. Finally, with the assumption that these instructions are performed with an ideal pipeline, the software execution time is the multiplication result of the total number of instructions and the period of the clock cycle.

3.1.2.4 Candidate component selection

The ultimate goal of the estimation approach is to make candidate component selection, which is an application-dependent procedure. In current design framework, the selection is carried out manually based on designers' experience and design constraints. A rule of thumb is to group tasks into contexts with the goal that both the number of contexts and the dependence edges crossing the contexts are minimized. For larger applications, the scheduling approach [75] and the grouping approach [97] could be applied. When global resource saving is an issue, the resource estimates are important inputs. However, to make justified decisions, other information, such as power consumption, should be included as inputs. More importantly, control/data dependence between candidate components should be analyzed. Obviously, there should be control dependence between candidate components that are mapped to different contexts. The current approach does not include automated tools to support the analysis. Other tools and manual analysis are the solutions for now.

3.1.3 Modeling of DRHW and the supporting transformation tool

The modeling of reconfiguration overhead is divided into two steps. In the first step, different technology-dependent features are mapped onto a set of parameters, which are the size of the configuration data, the clock speed of configuration process and the extra delays apart from loading of the configuration data. In the second step, a parameterized SystemC module that models the behavior of the run-time

reconfiguration process is created. It has the ability to automatically capture the reconfiguration request and present the reconfiguration overhead during performance simulation. Thus, designers can easily evaluate the trade-offs between different technologies by tuning the parameters.

3.1.3.1 Parameterized DRCF template

The performance impact of using DRHW is dependent on the underlying reconfigurable technology. Products from different companies or different product families from the same company have very different characteristics, e.g., size of reconfigurable logic and granularity of reconfigurable logic. Different features associated with the reconfigurable technology are not directly modeled in the DRCF component. Instead, the DRCF component contains the functions that describe the behavior of the reconfiguration process and relates the performance impact of the reconfiguration process to a set of parameters. Thus, by tuning the parameters, designers can easily evaluate the trade-offs between different technologies without going into implementation details.

In the SystemC extension, a parameterized DRCF template is used. At the moment, the following parameters are available for designers:

- The memory address, where the context is allocated in the external memory that holds the configuration data
- The length of the required memory space, which represents the size of configuration data
- Delays associated with the reconfiguration process in addition to delays of memory transfers.

3.1.3.2 DRCF component and Reconfigurable SoC modeling

The DRCF component is a model that can automatically capture the reconfiguration request and trigger the reconfiguration. In addition, a tool to automate the process that replaces candidate components by a DRCF component is developed, so system designers can easily perform the test-and-try and speedup the design space exploration process. In order to let the DRCF component be able to capture and understand incoming messages, the SystemC modules of the candidate components

must implement the *read()*, *write()*, g*et_low_addr()* and *get_high_addr()* interface methods showed in the code below. The DRCF component implements the same interface methods and conditionally calls the interface methods of target modules. With the forthcoming SystemC TLM 2.0 standard [150], new interface methods could be defined to comply with the TLM 2.0. Equivalently, OCP standard transaction lever interfaces [151] can be used.

```
class bus_slv_if: public virtual sc_interface{
  public:
           virtual sc_uint<ADDW> get_low_addr() =0;
           virtual sc_uint<ADDW> get_high_addr() =0;
           virtual bool read(...) =0;
           virtual bool write(...) =0;
};
```

A generic model of RSoC is shown in Figure 7. The left hand side depicts the architecture of the RSoC. The right hand side shows the internal structure of the DRCF component. The DRCF component is a single hierarchical SystemC module, which implements the same bus interfaces as other HW/SW modules do. A configuration memory is modeled, which could be an on-chip or off-chip memory that holds the configuration data. Each candidate component (*F1* to *Fn*) is an individual SystemC module that implements the top-level bus interfaces with separate system address space. The Input Splitter (IS) is an address decoder and it manages all incoming Interface-Method-Calls (IMCs). The Configuration Scheduler (CS) monitors the operation states of the candidate components and controls the reconfiguration process. Each candidate component instantiates a *DONE* signal to the CS. This signal is activated when the connected candidate component finishes its execution.

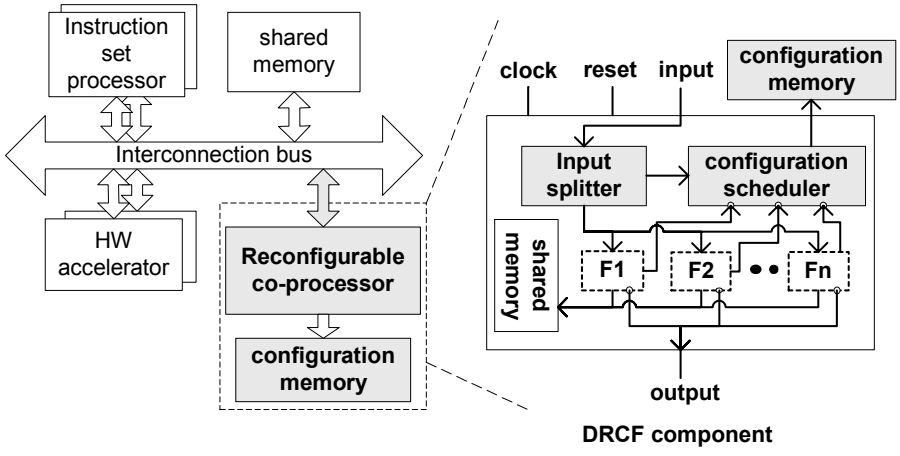


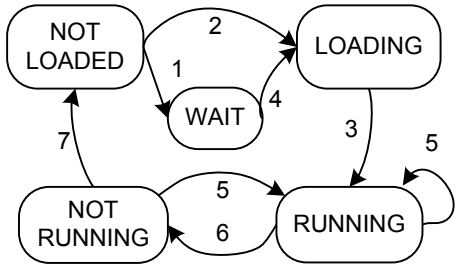
Figure 7. A generic model of RSoC.

The DRCF component works as follows. When the IS captures an IMC to a candidate component, it will hold the IMC and pass the control to the CS, which decides if reconfiguration is needed. If so, the CS will call a reconfiguration procedure that uses the parameters specified in the first step to generate the memory traffic and the associated delays to mimic the reconfiguration latency. If the CS detects the RTR hardware is loaded with another module, a request to reconfigure the target module will be put into a FIFO queue and the reconfiguration will be started after the RTR hardware has no running module. After the CS finishes the reconfiguration loading, the IS will dispatch the IMC to the target module.

The context switching with pre-emption is a common approach in operating systems, the implementation of which does not introduce too much overhead because of the regularity of the register organization in GPP. In the DRCF component, the pre-emption technique is not supported because of the very high implementation costs of context switching. In the modeling approach, designers can use different CS models when candidate components are mapped to different types of reconfigurable devices, such as partial reconfiguration and single-context devices.

There is a state diagram common to each of the candidate components. Based on the state information, the CS makes reconfiguration decisions for all incoming IMCs and *DONE* signals. A state diagram of partial reconfiguration is presented in Figure 8. For single context and multi-context reconfigurable resources, similar state diagrams can be used in the model. The main advantage of the modeling method is that the rest of

the system and the candidate components need not be changed between a static system and a run-time reconfigurable system, which makes this method very useful in making fast design space exploration.



State Definitions:

NOT LOADED:	module is only in the configuration memory
LOADING:	module is being loaded
WAIT:	module is waiting in a FIFO queue to be loaded
RUNNING:	module is running
NOT RUNNING:	module is loaded, but not running

State Transition Conditions:

1. IMC to the module occurs & not enough resources
2. IMC to the module occurs & enough resources
3. CS finishes the loading
4. Other modules finish & enough resources
5. IMC to the module occurs
6. Module finishes
7. CS flushes the module

Figure 8. State diagram of candidate components.

3.1.3.3 An automatic code transformer for DRCF component

In order to reduce the coding effort, we have developed a tool that can automatically transform SystemC modules of the candidate components, which however must follow a pre-defined coding pattern (using the predefined *bus_slv_if* methods, as shown in section 3.1.3.2), into a DRCF component. The inputs are SystemC files of a static architecture and a script file, which specifies the names of the candidate components and the associated design parameters, such as configuration latency. The tool contains a hard-coded DRCF template. It first parses the input SystemC code to locate the declarations of the candidate components (The C++ parser is based on Opencxx [152]). Then the tool creates a DRCF component by filling the DRCF

template with the declarations and making the appropriate connections. Finally, in the top-level structure, the candidate components are replaced with the generated DRCF component. During simulation, data related to reconfiguration latency will be automatically captured and saved in a text file for analysis. A VCD (Value Change Dump) file will also be produced to visualize the reconfiguration effects.

3.1.4 Link to low-level design

The low-level design is divided into detailed design and implementation design. The output of the detailed design is the intermediate representation of the system, in which SW is represented as C or assembly code and HW is represented as RTL-HDL code. The implementation is the phase where binary code for SW, bitstream for RTR HW and layout for ASICs are generated.

In our approach, automatic code generation for low-level design is not provided and designers should manually or using other tools transform the SystemC representation of the reconfigurable system to low-level code, such as C code for SW implementation and VHDL code for HW implementation. The implementation of the reconfiguration is technology-dependent and is outside the scope of the design method.

In our work, we used the Dynamic Circuit Switching (DCS)-based technique [153] to carry out the cycle-accurate co-simulation between the functions mapped onto the RTR hardware and the functions mapped onto the static part of the system. A VHDL module for each of the functions mapped onto the RTR hardware is manually created. Multiplexers and selectors are inserted after the outputs of the modules and before the inputs of the modules. They are automatically switched on or off according to the configuration status. In the cycle-accurate simulation model, the reconfiguration is modeled as pure delay.

3.2 A WCDMA detector case study

We selected a WCDMA detector [154] design case to validate the SystemC-based approach. We targeted on an RTR-type of implementation and the implementation platform was the VP20FF1152 development board from Memec Design group [155], which contains one Virtex2P XC2VP20 FPGA [156].

3.2.1 System description

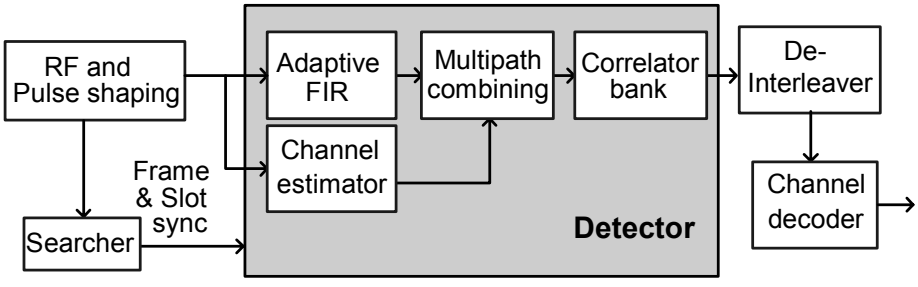


Figure 9. The WCDMA base-band receiver system.

The whole WCDMA base-band receiver system is depicted in Figure 9. The case study focuses on the detector portion (the shaded area in Figure 9) of the receiver and a limited set of the full features were taken into account. The detector case used a 384 kbits/s user data rate without handover. The detector contains an adaptive filter, a channel estimator, a multi-path combiner and a correlator bank. The adaptive filter is performing the signal whitening and part of the matched filtering traditionally implemented with the RAKE receiver. The channel estimator module calculates the phase references. In the combiner part, the different multi-path chip samples are phase rotated according to the reference samples and combined. Finally, the received signal is de-spread in the correlator bank.

When compared to traditional RAKE-based receiver concepts, this WCDMA detector achieves 1–4 dB better performance in vehicular and pedestrian channels. The detector thus provides performance benefits in more challenging channel conditions. As the traditional RAKE concepts contain several correlators for tracking the multi-path components, this detector contains a single channel equalizer for performing multi-path correction. This results in improved scalability, since increasing multipaths or data rates would mean increasing the amount of early/on-time/late correlators in the traditional RAKE-based concepts.

3.2.1.1 Adaptive filter

Regardless of the data rates or channel configurations required by the specification, the adaptive filter block is unchanged as it simply processes chip samples before the de-spreading takes place. Extendibility aspects are also not a problem as no changes

are required to support other demands. The adaptive filter is implemented by using basic FIR filtering structures with a delay line and taps for extracting complex sample values to be multiplied with the tap coefficients. The implementation is fully parallel, so the number of multiplier units for coefficient multiplication in both I and Q branches and the units needed for calculating new coefficients equal the number of taps in the filter.

3.2.1.2 Channel estimator

The function of the estimator is to de-spread the CPICH (Common Pilot Channel) chips on different multi-paths with correctly timed spreading and channelization codes. Then the received and de-spread CPICH symbols are multiplied with the complex conjugates of the CPICH pilot pattern. The output is channel estimates for different multi-paths, which are used in the combiner to rotate received chips in different multi-paths before combining, in order to match their phases and amplitudes. The channel estimator receives timing information from the searcher block. This includes the delay information about multi-paths at a specified delay spread. The channel can therefore be thought of as an FIR filter with a number of taps and with most taps zero-valued. The task of the channel estimator is to find the tap values for those taps that the searcher determines to be non-zero. The CPICH channel estimate over one slot is formed by integrating over the number of symbols and then it is scaled. It is used for actual phase correction of the received chips. The CPICH estimates are used as channel references for every data channel.

3.2.1.3 Combiner

As the base station transmits the pilot symbols through the channel, the terminal receives the directly propagated symbols and the delayed multi-paths. As the pilot symbols are known beforehand, the channel tap coefficients for each multi-path can be calculated. The different multi-path chip samples are first phase compensated according to the channel tap estimates. This is done by multiplying the chip sample with the complex conjugate of the corresponding multi-path channel tap coefficient. Finally all the phase compensated chip samples are added together to form an equalized chip sample.

3.2.1.4 Correlator

The function of the correlator bank is to create de-spread symbols from the output of the multi-path combiner. Combined chips are de-spread by spreading code, which is formed from scrambling and channelization codes. After de-spreading, chips are integrated over the symbol period in an integrator and the result is scaled.

3.2.2 System-level design

The design started from the C-representation of the system. It contained a main control function and the four computational tasks, which lead to a simple system partition that the control function was mapped onto SW and the rest onto RTR hardware. The estimation tool was used first to produce the resource estimates. The results are listed in Table 1, where LUT stands for look-up table and register refers to word-wide storages. The multiplexer refers to the hardwired 18 x 18 bits multipliers embedded in the target FPGA.

Table 1. Estimates of FPGA resources required by the function blocks.

Functions	LUT	Multiplier	Register
Adaptive filter	1078	8	91
Channel estimator	1387	0	84
Combiner	463	4	32
Correlator	287	0	17
Total	3215	12	224

Based on the resource estimates, the dynamic context partitioning was done as following. The channel estimator was assigned to one context (1387 LUTs), and the other three processing blocks were assigned to a second context (1078 + 463 + 287 = 1828 LUTs). This partition resulted in both balanced resource utilization and less interface complexity compared to other alternatives.

A SystemC model of a fixed system was then created, which had two purposes in the design. The first was to use its simulation results as reference data, so the data collected from the reconfigurable system could be evaluated. The second purpose was to automatically generate the reconfigurable system model from it via the transformation tool.

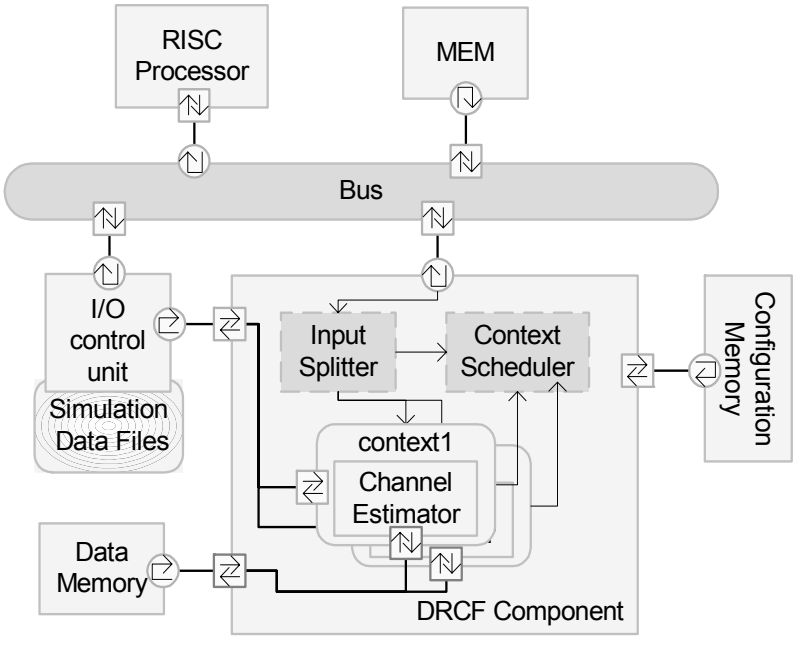


Figure 10. Reconfigurable system model of the WCDMA detector.

In the fixed system, each of the four detector functions was mapped to an individual hardware accelerator, and pipelined processing was used to increase the performance. A small system bus was modeled to connect all of the processing units and storage elements. The channel data used in the simulation was recorded in text files, and the processor drove a slave I/O module to read the data from the file. The SystemC models were described at the transaction level, in which the workload was derived based on the estimation results but with manual adjustment. The results showed that 1.12 ms was required for decoding all 2560 chips of a slot when the system was running at 100 MHz.

The transformation tool was used to automatically generate the reconfigurable system model, which is depicted in Figure 10, from the fixed model. The reconfiguration latency of the two dynamic contexts was derived based on the assumption that the size of the configuration data was proportional to the resource utilization, the number of LUTs required. The total available LUTs and the size of the full bitstream were taken from the Xilinx XC2VP20 datasheet. Some accurate approaches can be used to derive the reconfiguration latency. For example, the latency is related only to the region allocated to the dynamic contexts. In the current work, these have not been studied.

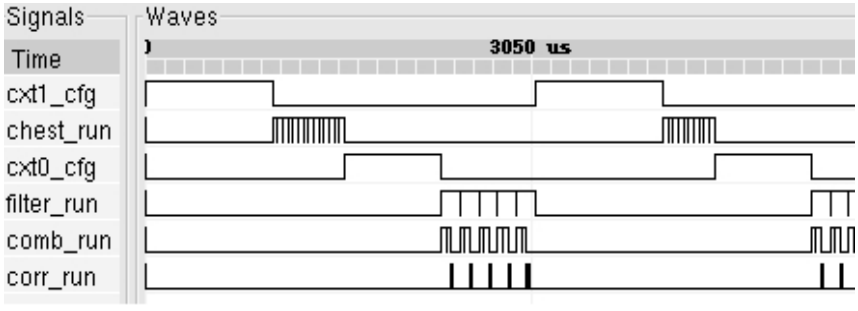


Figure 11. Simulation waveform shows the reconfiguration latency.

The performance simulation showed that the system required two reconfigurations per processing each slot of data. This is presented by the *cxt0_cfg* and *cxt1_cfg* in Figure 11. When the configuration clock was running at 33 MHz and the configuration bit-width was 16, the reconfiguration latency was 2.73 ms and the solution was capable of processing 3 slots of data in a frame.

3.2.3 Detailed design and implementation

In the low-level design phase, the RISC processor model was mapped onto the hardwired PowerPC core, and the data memories were mapped onto the embedded block memories. Other components were mapped onto Xilinx IP cores, if corresponding matches could be found, e.g., the bus model to the Xilinx Processor Local Bus (PLB) IP core. In addition to the basic functionality, we added a few peripherals for debugging and visualization. The implementation architecture is shown in Figure 12. Vendor-specific tools were used in the system refinement and implementation phases. Other than the traditional design steps for HW/SW implementation, additional steps for interface refinement, configuration design and partially reconfigurable module (PRM) design were needed. The PRM is referred to as the partial area of the target FPGA onto which the two contexts are mapped.

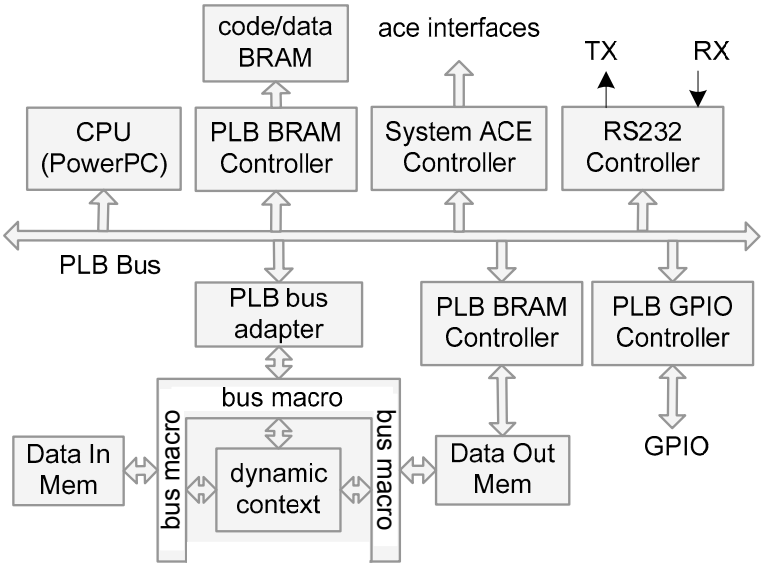


Figure 12. Implementation architecture of the WCDMA detector on the target Virtex II Pro FPGA.

3.2.3.1 Interface refinement

The number of signals crossing the dynamic region and the static region must be fixed, since the dynamic region cannot adapt itself for changing the number of wires. In this work, the step to define the common set of boundary signals shared between the PRMs is referred to as interface refinement. In Xilinx FPGAs, the boundary signals are implemented as bus macros [157], which are pre-routed hard macros used to specify the exact routing channels and will not change when modules are reconfigured. Because each bus macro is defined to hold 4 signals and there are only a limited number of bus macros, the boundary signals cannot be over-sized. Therefore, it is more beneficial to minimize the number of signals crossing the dynamic region and the static region, which can also relax the constraint during placement and routing. In this case study, the number of boundary signals was reduced to 82, which corresponded to the signals connected to the two 16-bit dual-port data memories and the PLB bus adapter. The implementation then required 21 bus macros.

3.2.3.2 Configuration design

This step is to define when and how to trigger the reconfiguration. The behavior of the DRCF component at the system-level modeling is to automatically capture the reconfiguration request and generate reconfiguration overhead when needed. In [158], we describe a technique that can realize this configuration transparence in low-level implementation. The technique requires a customized bus adapter and a customized OS kernel. The basic procedure is as follows. All reconfigurable modules are controlled by SW tasks via memory accesses. When the bus adapter notifies a memory access to a module, which is not loaded, it will trigger an interrupt and cause the OS to switch on another SW task. The current SW task that triggers the memory access (calling the unloaded module) is then blocked, and a reconfiguration request to load the called module is scheduled. When the reconfiguration is finished, another interrupt will be triggered, which causes the OS to release the blocked SW task. Therefore, reconfiguration becomes transparent to SW tasks, and all reconfiguration requests are automatically handled.

In this WCDMA case study, there was only a single SW task, which was used to control the four accelerators. It was much easier and cost-efficient to embed the reconfiguration requests into the SW task instead of using the customized bus adapter and the customized OS kernel, which would generate unnecessary overhead. The reconfiguration was implemented using the SystemACE Compact Flash (CF) solution and the configuration data was stored in a CF card. A simple device driver to control the SystemACE module was developed and the reconfiguration request was implemented as function calls to the SystemACE device driver.

3.2.3.3 Partial reconfigurable module design

RTL-VHDL code of the functions mapped onto PRMs was manually generated from the top-level C code. Synthesis results of the four functions are listed in Table 2. When considering the estimation, the results are over-estimated at about 55% on average. The main reasons for this derivation are that: 1) the estimator assumes fixed-length computation for all variables but in implementation some variables require only bit-level operations, 2) the estimator maps all multiplexers directly to LUTs but real synthesis tools usually utilize the internal multiplexers in individual logic elements [30]. Although there is a certain amount of deviation of the estimates, we can already use the results to make reasonable partitioning decisions. For the PRM,

the Xilinx module-based partial reconfiguration design flow [157] was used. First, each of the four detector functions was implemented as a single block. Then a context wrapper that matched the boundary signals was used to wrap the channel estimator as one module and the other three as another module. The static part was assigned to the right side of the FPGA (SLICE_X44Y111:SLICE_X91Y0), because 33 out of the 36 IO pads used are on the right side of the FPGA. The dynamic region was on the left side of the FPGA (SLICE_X0Y111:SLICE_X43Y0). The three IO pads that were on the left side were routed to the right side via a bus macro. The resource utilization of the placed and routed modules is presented in Table 3. The size of the configuration data is 279 KB for context 1 and 280 KB for context 2. The reconfiguration latency is about 4.3 ms. There are two contexts, and thus the total configuration latency for processing one slot of data is 8.6 ms. Including the data processing time, the total time spent on one slot of data is 9.66 ms.

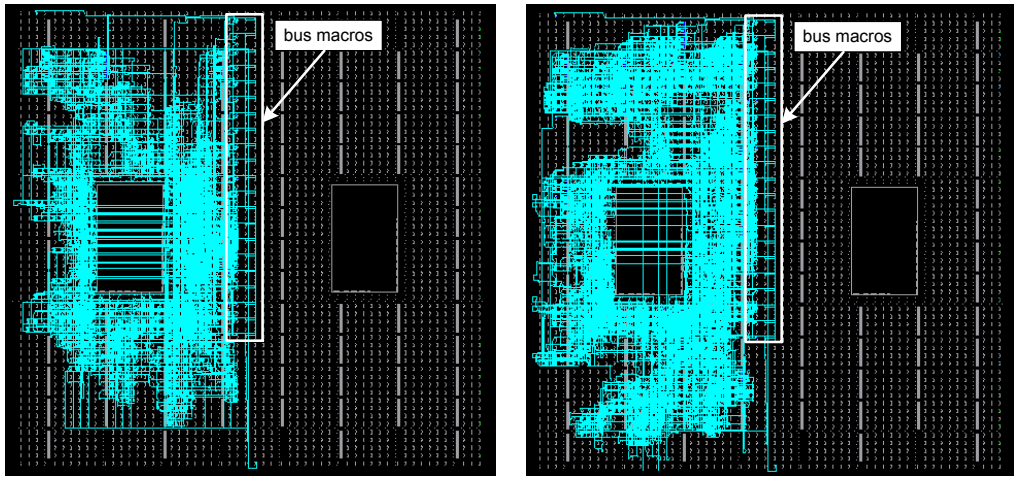
Table 2. HW synthesis results.

Functions	LUT	Multiplier	Register (bits)
Adaptive filter	553	8	1457
Channel estimator	920	0	2078
Combiner	364	4	346
Correlator	239	0	92

Routed PRMs on the dynamic region are shown in Figure 13. Context 1, which contains the channel estimator, is shown in Figure 13(a), and context 2, which contains the other three modules, is shown in Figure 13(b). In addition, a routed design after module assembly is shown in Figure 14. The assembled design is the integration of context 2 and the static part. The bus macros that are used for providing reliable connections for the boundary signals are marked by the block in the middle.

Table 3. Resource utilization in Xilinx XC2VP20.

	LUT	BRAM	MUL	Reg/bit	PPC
Static	1199	41	0	1422	1
Context 1	941	4	0	2083	0
Context 2	1534	6	12	1855	0



(a) context: channel estimator

(b) context: adaptive filter, combiner, correlerator

Figure 13. Routed design of PRM on the dynamic region.

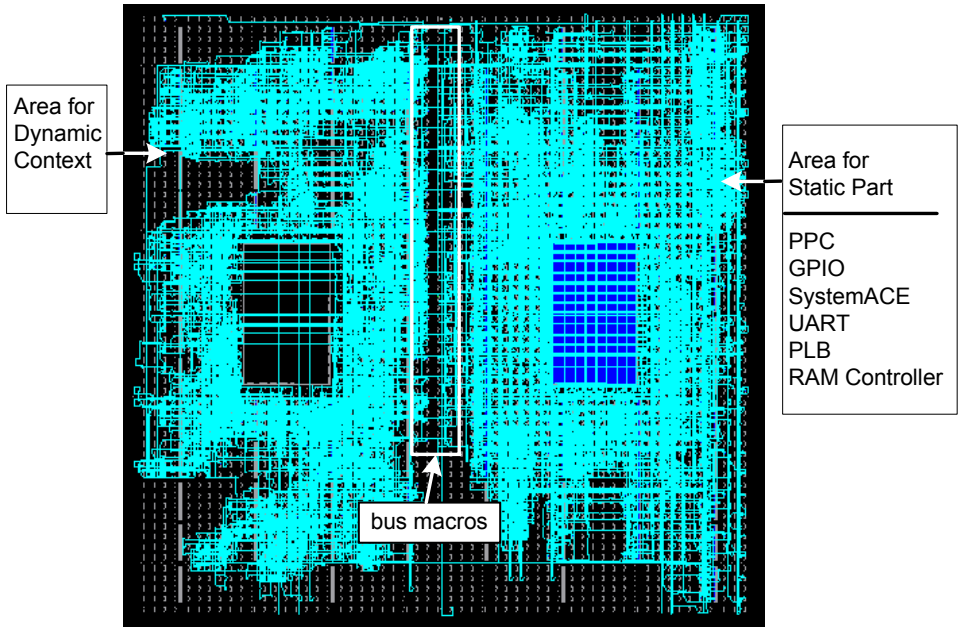


Figure 14. The assembly of context 2 and the static part.

3.2.3.4 Co-verification and execution

The EDK design suite [159] was used to create the simulation files for the complete system. However, the tool set did not provide the support to integrate the two dynamic contexts and the static context into a single simulation environment. A DCS-based VHDL wrapper [153] was manually created to enable the integration of the two dynamic contexts for simulation. Reconfiguration latency was estimated according to the SystemACE datasheet. Modelsim [160] was used as the simulation environment. In the SW side, SW code was compiled and stored as data into Block RAM. A PowerPC instruction-set simulator (ISS) was linked to Modelsim using the SWIFT interface [160] to perform HW/SW co-simulation.

The iMPACT [161] tool was used to transform the configuration files into SystemACE file format, and these configuration files were stored in a 128 MB CF card. During execution, a complete system (integration of the static part and context 1) was initially downloaded to the FPGA using the iMPACT, and then the partial bitstreams were automatically loaded by the SystemACE module when necessary.

3.2.4 Comparison with other implementation alternatives

In addition to the implementation of the dynamic reconfiguration approach, a fixed hardware implementation and a pure software implementation were made as reference designs. In the fixed-hardware implementation, the processing blocks were statically mapped onto the FPGA as accelerators and the scheduling task was mapped onto SW that ran on the PowerPC core. The resource requirements were 4632 LUTs (24% of available resources), 55 Block RAMs (62%) and 12 Block Multipliers (13%). The system was running at 100 MHz. The execution time for processing one slot of data was 1.06 ms. For our dynamically reconfigurable system, the required resources are calculated by summing up the required resources of the static parts and the largest ones of the two contexts. Considering LUTs, which are the main elements in FPGA, the dynamically reconfigurable system requires 2733 LUTs (1199 LUTs for the static part and 1534 LUTs for context 2), as shown in Table 3. Compared to the fixed reference system, the dynamic approach achieved a resource reduction of more than 40% in terms of the number of LUTs, but at the cost of eight times longer processing time.

For the full software implementation, the design was done as a standalone approach and no operating system was involved. Everything was running in a single PowerPC core and data were entirely stored in internal BRAMs. For the same clock frequency, the processing time of one slot of data was 294.6 ms, which was over 30 times of the processing time in run-time reconfiguration case, 9.66 ms as shown in section 3.2.3.3. This did not fulfill the real-time requirements.

3.3 Analysis and discussion

The main advantage of the SystemC-based approach is that it can be easily embedded into an SoC design flow to allow fast design space exploration for different reconfiguration alternatives without going into implementation. This is achieved by our system-level supporting tools and the modeling method of DRHW. We developed a high-level synthesis-based HW estimator, which can produce HW resource estimates for algorithms that are represented in ANSI-C code. This helps designers in the early phase of the design to make reasonable partitioning decisions without going into the implementation details, and therefore reduce the design time. In our DRHW modeling technique, the components that are mapped onto fixed HW accelerators and the DRHW are modeled using the same interface. Therefore, a component can be easily moved in to and out from DRHW when exploring the design space. This also helps to reduce the coding effort and thus makes the design space exploration process fast. In addition, to further reduce the coding effort, we have developed a SystemC code transformer that can automatically generate SystemC code of DRHW from existing SystemC code. Considering the design at a detailed level and implementation level is time consuming and can take from weeks to months, the SystemC-based approach can help to avoid re-design in the early design phase.

Comparing our design method and others is not feasible, because adopting a new design method and flow requires a large amount of economic as well as human effort. In addition, quantitatively evaluating the design flow is not practical, since most design-related attributes, such as design time and number of lines of target code, are in fact very dependent on designers and their experiences. In this work, the feasibility of the design method is studied by applying it to a real design case. Through the design case, the estimation approach and the DRHW modeling approach have shown their usefulness by providing reasonably accurate results without going into low-level implementation. The HW resource estimates were used to guide the context partitioning

process, which resulted in a balanced partitioning decision. The performance impact of reconfiguration overhead was quickly revealed through performance simulation, where the DHRW modeling technique was applied.

The potential benefit of using the run-time reconfiguration approach is obviously the significant reduction of reconfigurable resources. Compared to a completely fixed implementation, the reduction of LUTs is more than 40%. Compared to a full software implementation, the run-time reconfiguration approach is over 30 times faster. The commercial off-the-shelf FPGA platform caused limitations on the implementation of run-time reconfiguration. Although the selected approach used partial reconfiguration, the required configuration time significantly affected the performance in the data-flow type WCDMA detector design case. The ratio of computing to configuration time is about 1/8 in this design case. This value shows that configuration overhead must be effectively managed in order to gain benefit of using DRHW. In the following chapters, we will present several techniques to reduce the negative impact of the reconfiguration overhead.

4. Task scheduling approaches for run-time reconfigurable devices

4.1 Introduction

As presented earlier, the flexibility of DRHW allows such devices to be shared by tasks in a time-multiplexing manner. With the multitasking feature, DRHW tasks can be managed similarly to those in multiprocessor systems. However, one challenging issue in using DRHW is how to manage configuration. The RTR that enables the multitasking feature results in the configuration overhead. There are different approaches [97] to hiding or reducing configuration latency, such as configuration prefetching and configuration caching. At the design level, the concern is how to apply these techniques in the task scheduling process in order to effectively and efficiently hide configuration latency.

Another difference from multiprocessor scheduling is that task allocation needs to be carefully managed in order to efficiently utilize the DRHW. In multiprocessor scheduling, the target system usually consists of a number of homogeneous processors, and a task can be mapped onto any processor without much difference from a scheduling point of view. However, in DRHW scheduling, allocating a task to an inappropriate area can cause fragmentation of the reconfigurable logic and therefore prevent new tasks from being loaded. All these RTR-specific features make DRHW scheduling more complicated than the scheduling in a multiprocessor environment. In this chapter, we present techniques that optimally or near optimally schedule tasks onto DRHW.

As presented in section 1.2.2, various task scheduling techniques for DRHW have been proposed. Similar to multiprocessor scheduling, there are in principle three different scheduling approaches. The first is static scheduling, in which execution orders are decided at design time. It is suitable when tasks have known dependencies. The second is dynamic scheduling, in which all decisions are made at run-time. However, there is a stringent time-limit to make decisions, and thus only simple scheduling algorithms can be applied. The third is quasi-static scheduling, in which the execution orders are determined partially at design time and partially at run-time. This approach is interesting as it can make decisions based on the current run-time status but does not need to carry out all the calculations at run-time.

Because quasi-static scheduling has the advantage of balanced efficiency and performance, we use it as the basic framework in our approach. Our quasi-static task scheduling is divided into two parts: design-time scheduling and run-time scheduling. At design time, scheduling is preformed for tasks of each individual application, and the main focus is to use configuration prefetching to hide configuration latency. For each individual application the design-time scheduler produces a number of possible options. Each option corresponds to an optimal or near-optimal scheduling under the setting of the number of tiles on the device. We mark each option as a pareto point, and the scheduling result contained in it is called a pareto profile. During execution, when an application is ready the run-time scheduler selects a suitable profile based on the current device status. In fact, our quasi-static scheduling approach shares an idea similar to the one in [121]. However, the main difference is how the configurations are managed. Our contributions are highlighted as follows.

- The focus in design-time scheduling is to provide optimal scheduling solutions for a group of tasks whose dependence is known at design time. Three scheduling techniques with different problem-solving strategies are developed and quantitatively evaluated in our work.
- For the run-time scheduling, we propose a novel technique called configuration locking. The main idea is to dynamically monitor the running tasks and always lock the configurations of a number of the most frequently executed tasks on the device in a way that the resources occupied by these locked tasks cannot be evicted by any ready task. (A ready task means that the task is ready to be executed.) This is similar to the cache locking technique [162] in a general memory system.

4.2 Target models

4.2.1 Device model

We use a generic configuration model as the device model. It consists of a number of continuously connected homogeneous tiles and each tile consists of the circuit and its own configuration-SRAM that controls the circuit, as described in Figure 15. A task that requires *m* tiles of resources can use any set of *m* continuously connected tiles. A crossbar connection is used to connect the configuration SRAMs of the tiles to a number of parallel configuration controllers. The crossbar ensures that any configuration SRAM can be accessed by any configuration controller, but only one at a time. Data

transfers among tiles and between tiles and the rest of the SoC are all through the communication network. A thorough discussion of the parallel configuration model is presented in Chapter 5.

The model can be described using the following parameters: 1) N_{tile}, the number of tiles; 2) N_{ctrl}, the number of configuration controllers; 3) *SIZE*, the size of a single tile; and 4) *CL*, the configuration latency of a single tile. To present the generality of the model, we provide two simple examples for setting these parameters. If we set parameter N_{ctrl} to 1, the model then describes a partial reconfigurable device, such as Virtex devices [27]. If we set the *SIZE* to a very high value such that a single tile is large enough to hold a task, the model can then be used to describe a number of individual tiles on a network-on-chip (NoC) platform.

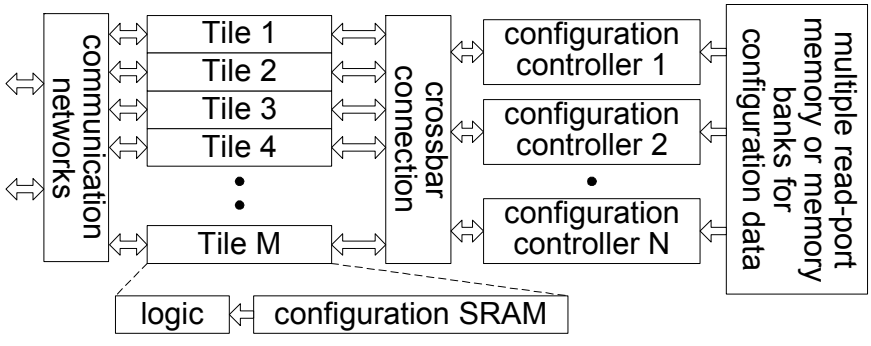


Figure 15. A generic parallel reconfiguration model.

4.2.2 Task model

Most applications of an embedded system are independent from each other, although occasionally they are linked together by users to achieve a particular purpose. For example, when a user wants to use a mobile device to send an edited picture from a live music concert, first he uses the device to capture the wanted view and uses an editing application for inserting text, and then he sends it through as a multimedia message. In this scenario, applications for image encoding, image processing and communication are called in sequential order. However, this dependence is forced by users, and it is not explicit during the application development stage.

In our approach, such user-level dependence is not considered, and each application is treated as an independent unit. However, each application might consist of a number of closely dependent tasks, which can be modeled as a directed-acyclic graph (DAG). In the DAGs, nodes represent tasks and edges represent dependencies of the tasks. Dependencies crossing different applications are not modeled and task reuse for different applications is not applied, as we consider that such situations rarely happen in real cases. Applications can be non-periodic or periodic with soft deadlines. A soft deadline means that a process should but need not necessarily finish its execution by the deadline. For a periodic application, tasks from the current period cannot start before all tasks from the previous period have finished. At run-time, applications are triggered either periodically or sparsely without pre-defined orders.

Dependent tasks of an individual application are modeled as a DAG, $G(V,E)$, where $V = \{j_1,j_2,\ldots, j_n\}$ is a set of nodes that represent the tasks and E is a set of edges that represent the dependence of the tasks. A task is ready to run when all of its predecessors have finished. There are two attributes for a task i, execution time, RT_i, and the number of required tiles, R_i. However, normal DAG representation is not enough as configurations are explicit processes but do not explicitly appear in the DAG. For our purpose, we use an extended DAG $G^+(V^+,E^+)$. Extra configuration nodes V' are added with a single node representing the configuration of one tile. Extra edges E' are constructed from V' to V, because configurations have to precede execution. Mathematically, the relationship between the normal DAG and the extended DAG is $V^+ = V \bigcup V'$ and $N^+ = N \bigcup N'$. As an example, Figure 16(a) shows a normal DAG of 4 tasks and Figure 16(b) shows the extended DAG where $C_{<i,j>}$ represents the configuration of the j^{th} section of task i.

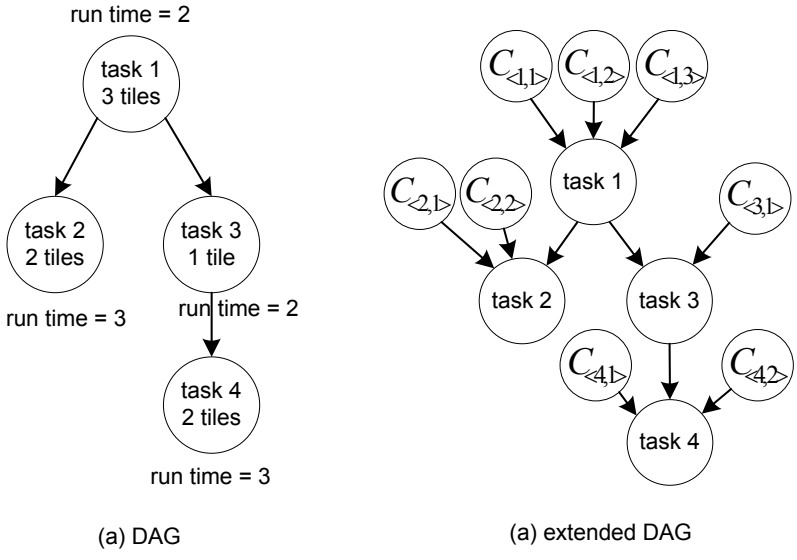


Figure 16. DAG and extended DAG.

4.3 Static scheduling approaches

The scheduling problem at the design-time phase is defined as searching for a valid task schedule that minimizes the overall execution time of a set of dependent tasks over a given DRHW model. This scheduling problem is very similar to multiprocessor scheduling, in which the objective is to map a set of dependent tasks onto a number of processors with the goal of minimizing the overall execution time. However, our scheduling problem for run-time reconfiguration is more complicated, because the task allocation, configuration prefetching, task dependence and configuration-task dependence all need to be managed together under the constraints that both the number of computation resources and the number of configuration resources are limited.

In this section, we present three task schedulers that are from different domains of problem solving. The first is a heuristic approach developed from traditional list-based schedulers [33]. The second is based on a full-domain search using constraint programming (CP) [34]. The last is a guided random search implemented using a genetic algorithm (GA) [35].

4.3.1 The list-based scheduler

List-based scheduling has been extensively used in task scheduling for single processor or multiprocessor environments, high-level synthesis, or similar situations, where the problem can be represented as a DAG. The basic idea is to sort the DAG nodes based on their priorities and always schedule the highest priority node first. The priority usually refers to the urgency of a node, and the nodes in the critical path have higher priorities. In DRHW scheduling, issues such as task allocation, configuration scheduling and configuration prefetching can also affect the scheduling results. Therefore, a new technique to calculate the task priority is needed.

The principle of configuration prefetching is to load tasks whenever there are tiles and configuration controllers available, instead of after the tasks become ready. In the list-based scheduler, each task has a priority, which represents the urgency of the configuration of the task. The task with the highest priority is scheduled first when free resources are available. The priority function consists of three elements: the mobility, the gap and the delay. The mobility shows the urgency of execution, and a low mobility value means a high priority. The gap shows how much benefit a task can get if its configuration immediately starts. A low gap value means a high priority. The delay shows how many additional configurations have to be delayed if the configuration of the task cannot start immediately. It is obvious that prefetching successors prior to predecessors does not bring any benefits. Therefore, a task with more successors has a high delay value, which means a high priority.

```
1.   Insert_Tasks (V,PList);
2.   s_time = 1;
3.   while(PList ≠ ϕ ) do
4.     Calculate_RTR_Priority (PList,s_time);
5.     while(Search_for_Free_Res(s_time) ≠ 0) do
6.       task = First(PList);
7.       m = Required_Tiles(task);
8.       Search_for_Candidate_Tiles(tiles,m);
9.       Schedule_Configuration(tiles,task);
10.      run_time = Calculate_Ready_Time(task);
11.      Schedule_Task(tiles,task,run_time);
12.      Delete(PList, task);
13.      if(PList = ϕ) then
14.        break;
15.      end if;
16.    end while;
17.    s_time = s_time +1;
18.  end while;
```

Figure 17. List-based scheduler.

The pseudo code of the algorithm is shown in Figure 17. The algorithm iterates starting from scheduling time (s-time) 1 and stops when all tasks have been scheduled, as in (1)–(3). In each iteration, the priorities of all the unscheduled tasks are calculated, as in (4). The algorithm searches the DRHW device for any pair of a free tile and a free configuration controller. The scheduling of configurations and executions is continuously performed as long as such a pair exists, as in (5)–(16). Upon scheduling, candidate tiles are selected for the task and configuration of the task is then scheduled, as in (7)–(9). Due to prefetching, a task might not be ready when its configuration has finished. The ready time is then calculated, and execution of the task is scheduled upon that time, as in (10)–(11). When a pair of a free tile and a free controller cannot be found, the s-time is increased, as in (17). Then, free resources might be available at the new s-time.

A brief explanation of the important functions is as follows. The function *Calculate_RTR_Priority* calculates the priorities of the tasks and sorts them according to the priority values. The value is calculated as *a/mobility+b/gap+c*delay*, where *a*, *b* and *c* are weight values decided by the designers. The mobility is calculated as (*ALAP s-time*) – (*ASAP s-time*) +1. The gap is calculated with the assumption that the configuration of the task starts at *s-time* and the task can start to run at *ASAP s-time*. Its value is equal to (*ASAP s-time*) – (*configuration_end s-time*). Offset values are added to the gap values to make all of them positive. The delay value is equal to the normalized value of the total number of successors of the task. The function *Search_for_Free_Res* returns *TRUE* if a pair of a free tile and a free configuration controller are found at *s-time*. The function *Search_for_Candidate_Tiles(tiles, m)* returns the *m* continuously connected tiles on which the configuration can finish within the shortest time. The function *Schedule_Configuration* uses a resource-constraint ASAP scheduling approach to schedule the configuration of the task onto the selected tiles. Its return value represents the configuration finish time. The function *Schedule_Task* sets the task to run at *run_time*. The function *Insert_Tasks(V,L)* puts the tasks in *V* into List *L*. The function *First(L)* returns the first task in *L*. The function *Delete(L, T)* deletes element *T* from *L*. The function *Required_Tiles(T)* returns the number of required tiles of task *T*. The function *Calculate_Ready_Time* returns the earliest *s-time* at which both configuration of the task and executions of all its predecessors can be finished.

4.3.2 The constraint programming approach

4.3.2.1 Introduction to constraint programming

Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it [163]. CP consists of describing constraints and solving such constraints. Programmers state the problem requirements, but do not need to specify how to meet these requirements. Constraint solvers, constructed based on well-known algorithms, such as branch & bound (B&B) [164], will find a solution that satisfies all of the constraints.

Specifically, the problem that we are dealing with belongs to a finite domain. The satisfaction of constraint problems over a finite domain is usually referred to as the constraint satisfaction problem (CSP). In this paper, our goal is to find an optimal schedule, and the B&B technique is used to solve it. Its basic idea is to keep tracking the best solution and to try to improve it until the entire search tree has been explored. Because B&B covers the complete search space, this method is proven to find a global optimal solution.

4.3.2.2 Constraint models

In this section, the constraint models that characterize the DRHW task scheduling problem are presented using implementation-independent mathematical formulas. There are five constraints and one optimization goal.

A. Definition of terms

- T_i : The start execution time of task i
- RT_i : The duration of the execution time of task i
- R_i : The number of required tiles by task i
- $< i, n >$: The n^{th} segment of task i
- $Tile_{<i,n>}$: The tile that is allocated to the n^{th} segment of task i
- $C_{<i,n>}$: The start configuration time of the n^{th} segment of task i, on the $Tile_{<i,n>}$.

B. Task dependence modeling

This constraint states the task dependence. Basically, the start execution time of any task has to be larger or equal to the end execution time of any of its predecessors. Therefore, task dependence can be described using the following constraint:

$$\forall_{(i,j)\in E} T_i + RT_i \le T_j \tag{1}$$

C. Configuration dependence modeling

This constraint states the dependence between the execution of a task and the configurations of the task. A task has to be loaded before its execution, thus the start execution time of any task has to be larger or equal to the end of the configuration of all assigned tiles. Mathematically, the constraint is:

$$\forall_{(n,i)\in E'} C_{<i,n>} + CL \le T_i \tag{2}$$

D. Tile allocation modeling

The allocation constraint states that a task, which requires more than one tile, must be assigned to continuously connected tiles. This is equivalent to setting the tile allocated to the $(n+1)^{\text{th}}$ segment of task i incrementally to the next tile allocated to the n^{th} segment of the task i. Mathematically, the allocation constraint is represented as:

$$\forall_{i\in N,(n,n+1)\in[1,R_i]} Tile_{<i,n+1>} = Tile_{<i,n>} + 1 \tag{3}$$

E. Resource constraint modeling – tile

Tiles are shared resources. A task starts to occupy a tile when the configuration of the tile starts, and it releases the tile when it finishes executing. Therefore, if a tile is allocated to two different tasks, the time frames during which they occupy the tile cannot overlap. For example, Figure 18 shows a situation that two independent tasks i and j, which require three and two tiles respectively, share tile number 3. Legal schedules are that either $C_{<j,1>}$ is scheduled after task i finishes execution, shown in Figure 18(a), or $C_{<i,3>}$ is scheduled after task j finishes execution, shown in Figure 18(b). When the two tasks are assigned to share different tiles, the constraints will change correspondingly.

We construct two kinds of tile constraint models. The first is for tasks that do not have predecessor-successor dependence. It is formulized as:

$$\begin{aligned}&\forall_{(i,j)\in N, m\in[1,R_i], n\in[1,R_j]}\\&\quad(((Tile_{<i,m>}=Tile_{<j,n>})\wedge(T_i+RT_i\le C_{<j,n>}))\vee\\&\quad((Tile_{<i,m>}=Tile_{<j,n>})\wedge(T_j+RT_j\le C_{<i,m>}))\vee\\&\quad(Tile_{<i,m>}\ne Tile_{<j,n>}))\end{aligned} \tag{4}$$

The basic idea is to force the configurations of one task to be scheduled on shared tiles after the end execution time of the other task. These are shown in the first two lines of the expression. Certainly, if two tasks do not share tiles, such constraints do not apply. This is shown in the last line.

The second constraint is for dependent tasks. If task *i* precedes task *j*, then the constraints $C_{<i,m>}+CL\le T_i\big|_{m\in[1,R_i]}$ and $T_j\ge T_i+RT_i$ can be propagated to $T_j\ge C_{<i,m>}\big|_{m\in[1,R_i]}$, which is contradictory to the constraint $T_j+RT_j\le C_{<i,m>}$. In fact this means that on shared tiles, configurations of successor tasks cannot start before those of predecessor tasks. Therefore, we need to model only the case represented in Figure 18(a). The constraint can then be simplified as:

$$\begin{aligned}&\forall_{(i,j)\in N, m\in[1,R_i], n\in[1,R_j]}\\&\quad(((Tile_{<i,m>}=Tile_{<j,n>})\wedge(T_i+RT_i\le C_{<j,n>}))\vee\\&\quad(Tile_{<i,m>}\ne Tile_{<j,n>}))\end{aligned} \tag{5}$$

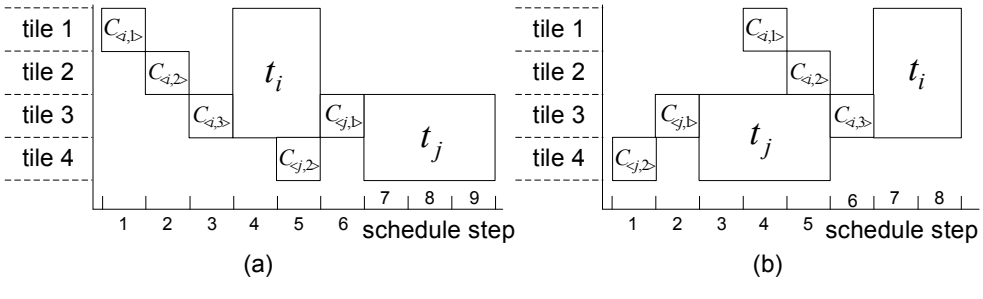


Figure 18. Tile constraint modeling.

F. Resource constraint modeling – controller

Because there are a limited number of configuration controllers, they are also shared resources. The number of tiles being configured at any given time cannot exceed this limit. This can be mathematically modeled as:

$$\forall_{t\in[1,\max(T_i+RT_i)|_{i\in N}]}(\Sigma_{i\in N,m\in[1,R_i]}B(t,C_{<i,m>})\le NC)\text{, where} \quad (6)$$

$$B(t,C_{<i,m>})=\begin{cases}1, & if \quad C_{<i,m>}\le t<C_{<i,m>}+CL\\ 0, & else\end{cases}$$

G. Optimization goal

The objective of the approach is to find a schedule that minimizes the overall execution time. To achieve this goal, we minimize the maximum of end execution times of all tasks in the labeling step:

$$\text{minimize}(max\,(T_i+RT_i)\big|_{i\in N}) \quad (7)$$

4.3.2.3 Constraint implementation

Both the constraints, described from B to E, and the optimization goal are implemented as they are in the Prolog language [164]. The constraint F is implemented using a complex constraint *cumulative*, which can be efficiently solved. In fact, the constraints D and E can be jointly implemented using the complex constraint *diff* with additional dependence constraints. However, because the configurations are crossly stated both in E and F, this results in less efficient solving in terms of the computer run-time in our experimentation.

4.3.3 The genetic algorithm

4.3.3.1 Introduction

The GA is a guided random search technique inspired by evolutionary biology and natural genetics [165]. The basic idea is to iteratively improve the results (individuals) through randomly combining (crossover) and modifying (mutation) the previous

results until some termination criteria are satisfied. In each generation, preferred individuals survive, thus each generation tends to be better than the previous one. Its implementation is usually based on a loop structure, as follows.

step 1: Create an initial population (a group of solutions).

step 2: Evaluate the fitness of all individuals in the current population. (Fitness is a measurement of the quality of an individual.)

step 3: Select individuals to reproduce, and breed new offspring through crossover with high probability and mutation with low probability.

step 4: Stop if termination criteria are satisfied, otherwise go back to step 2.

The chromosome (strings that represent solutions) and the evaluation process are problem-specific. The genetic operators (crossover, mutation, evaluation and selection that operate the chromosome to evolve into new offspring) control the evolution process. We use the implementation in a multiprocessor scheduler [166] to illustrate these basic ideas. In [166], a solution is represented with two-dimension strings $\{S_1, S_2, \ldots, S_n\}$. Each string S_i represents the tasks scheduled on the processor P_i and the order of appearance is the execution order of the tasks.

The crossover allows two parents, par_1 and par_2, to mate and generate two new individuals, $child_1$ and $child_2$. The crossover can be seen as a way of achieving the guided search, because new solutions are directly derived from the old ones. In [166], a random task is first selected, and then the crossover site (a place to cut a string into half) for each string S_i is generated based on the height value [166] of the selected task. The height values implicitly determine task precedence. Then, string S_i of $child_1$, is generated by appending the right string (the partial string after the crossover site) of the S_i of par_1 to the left string (the partial string before the crossover site) of the S_i of par_2. The offspring, $child_2$, is built in the same way after swapping the parents.

The mutation generates a new individual by randomly modifying the chromosome of another individual. It is a technique to increase the randomness of the search to avoid solutions being trapped in local optimal points, which is the ultimate result if only the crossover is used. In [166], the mutation is performed by randomly changing the positions of two tasks with the same height.

The evaluation measures the fitness of all individuals, so the worst results can be eliminated in a later phase. In [166], the fitness of an individual i is calculated as $max\_length - current\_length_i + 1$, where max_length is the longest schedule length in the current generation and $current\_length_i$ is the schedule length of individual i.

The selection picks up some individuals to reproduce offspring. The natural rule is that better ancestors tend to generate better offspring, because the "good" genes are passed on. In [166] and other GA approaches, the roulette wheel is a common style of implementing this GA operator. The basic procedure is to assign each individual with a slot size in the roulette wheel that is proportional to its fitness value. Then a random number is generated as an index to the roulette wheel, and the individual that covers the index is selected to reproduce. Because an individual with a larger fitness value covers a larger slot, it then has higher chance of being selected to reproduce. Towards our problem, these problem-specific GA issues are described in the following subsections.

4.3.3.2 Coding of solutions

The coding style in multiprocessor scheduling cannot be applied to our problem, because 1) a task might require multiple tiles, and 2) both tasks and their configurations need to be scheduled. In our approach, we use a pair of two-dimension strings to represent an individual. The first two-dimension strings $\{Tile_1, Tile_2, \ldots, Tile_n\}$ are the task strings (T-strings), similar to those in [166]. They represent the scheduling results of tasks on the tiles. Each tile has a corresponding string, and a string $Tile_i$ represents the tasks scheduled on the i^{th} tile. The order of the tasks on string $Tile_i$ is then the execution order of these tasks on the i^{th} tile. This is similar to the strings in multiprocessor scheduling. However, for a task that requires multiple tiles, its instance appears on all of the tiles assigned to it.

The second two-dimension strings $\{Ctrl_1, Ctrl_2, \ldots, Ctrl_n\}$ are the controller strings (C-strings). They represent the configuration scheduling results. Each controller has a corresponding string, and a string $Ctrl_i$ represents the configurations scheduled on the i^{th} controller. The same as in the task strings, the order of appearance on $Ctrl_i$ is the configuration order using the i^{th} controller. An example of two-dimension strings and the corresponding scheduling results of the extended DAG in Figure 16 are shown in Figure 19.

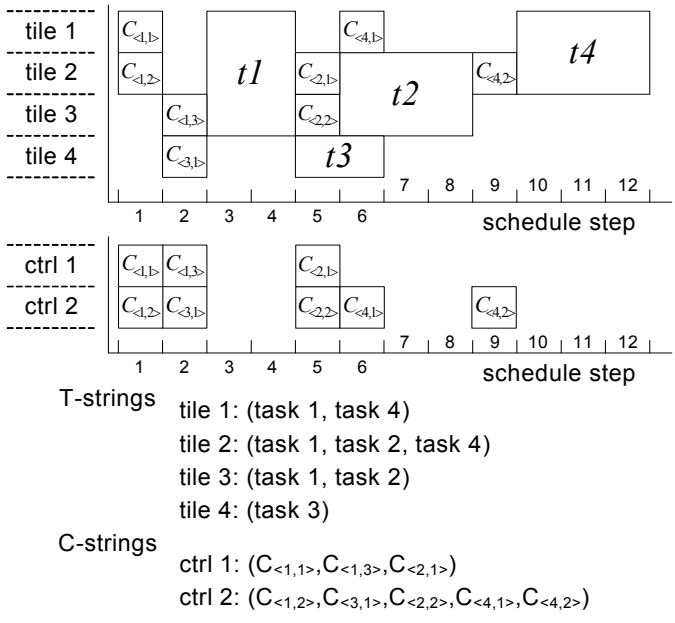


Figure 19. Gene representation.

Based on the strings of an individual we can derive a new graph by inserting extra edges for the scheduling dependence into the extended graph G^+. We refer to this graph as the schedule graph (s-graph), and each individual has its own s-graph. The s-graph is constructed as follows. Firstly, for each two adjacent positions, j and $j+1$, on the T-string $Tile_i$, an edge from the task node at the j^{th} position to the task node at the $(j+1)^{th}$ position is inserted into G^+. For example, on the strings as shown in Figure 19, an edge from task 2 to task 4 is needed, because task 2 is scheduled before task 4 on tile 2. Secondly, on each T-string $Tile_i$, an edge from the task node at the j^{th} position to the configuration node, which configures the task at the $(j+1)^{th}$ position onto the i^{th} tile, is inserted into G^+. For example, an edge from task 2 to the configuration node $C_{<4,2>}$ is needed, because the configuration cannot start before task 2 has finished. In fact, this edge can replace the first edge by propagating the dependence between the nodes, but we keep the first one because it can result in an efficient implementation. Finally, on each C-string $Ctrl_i$, an edge from the configuration node at the j^{th} position to the configuration node at the $(j+1)^{th}$ position is inserted into G^+. For example, a link from $C_{<1,3>}$ to $C_{<2,1>}$ is needed, because they are not allowed to run in parallel and configuration $C_{<1,3>}$ should precede $C_{<2,1>}$. So, with the additional links that show the dependence due to allocation, a schedule graph explicitly presents the execution order of tasks. The schedule graph for the chromosome given in Figure 19 is depicted in

Figure 20, in which the dotted lines with smaller arrows are the additional links that show the allocation dependence.

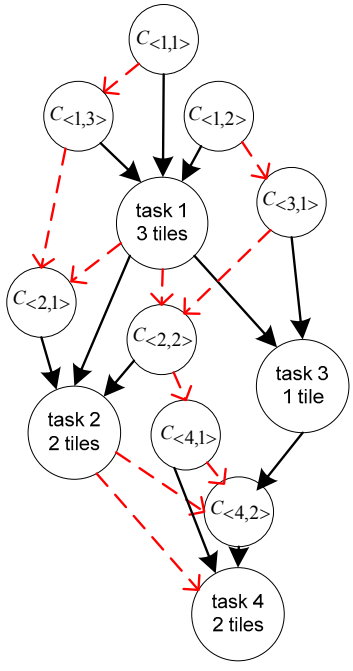


Figure 20. Schedule graph with additional links showing allocation dependence.

In our approach, each individual, including all offspring after crossover and mutation, represents a feasible solution. An individual represents a feasible solution if and only if it satisfies the precedence constraints and its s-graph is acyclic. This theorem has been proven in [167] for multiprocessor scheduling (in multiprocessor scheduling, the s-graph does not contain the configuration nodes and the associated edges). Since we are dealing with the same type of task graphs (although configuration nodes are added, they do not change the acyclic properties of the graph), this theorem also applies to our problem. In our case, the first condition is met as we always maintain task precedence in each string. The second condition is guaranteed in our genetic operators, which are described in later subsections.

4.3.3.3 Initial population

The initial population is a group of initial solutions, from which the GA starts to evolve. In our approach, the initial population is generated through a resource-constraint list

scheduling approach, but resources are randomly selected upon scheduling. The basic procedure of creating an initial individual is as follows.

step 1: Select a ready task node. A task node is ready if all of its predecessor task nodes are scheduled or if it has no predecessor task node.

step 2: Randomly select controllers for its configuration nodes, and randomly select tiles for the task node. If it requires multiple tiles, randomly select continuously connected tiles. Append the task node and its configuration nodes to the end of the strings of the selected resources.

step 3: If there are unscheduled task nodes, go to step 1. Otherwise an initial individual is created, and exit.

As nodes are placed based on their execution order, no cycle exists in the s-graph in all initial solutions.

4.3.3.4 Crossover

The crossover mechanism for multiprocessor scheduling [166] can guarantee that feasible solutions will be generated. However, a severe drawback is that the optimal solutions might never be generated. Correa et al. [167] fixes the problem with an improved crossover. In our work, we use this improved crossover and extend it for the task scheduling problem of DRHW.

The basic idea to guarantee feasibility in [166, 167] is as follows. Task nodes in the strings must be ordered based on their height values in order to satisfy precedence constraints. During crossover, a graph is divided into two acyclic sub-graphs, G_L and G_R, in such a way that there exist edges only from G_L to G_R, but not vice versa. The basic graph G is used in [166], and the s-graph is used in [167]. Then, the crossover sites are selected in such a way that all nodes in the left-strings belong to G_L and all nodes in the right-strings belong to G_R. Therefore, no cycle will be generated when swapping the right-strings between parents, and thus the offspring are feasible solutions.

In our approach, we use the s-graph of par_1 and par_2, to generate the two sub-graphs, similar to [167], but configuration nodes are considered as well. The procedure is as follows.

step 1: Start with a randomly selected task node. Move this node and its configuration nodes into G_L.

step 2: In the s-graph of par_1, search for the task nodes that precede the selected task node, and move these task nodes and their configuration nodes into G_L.

step 3: In the s-graph of par_2, search for the task nodes that precede the nodes already in G_L, and move these task nodes and their configuration nodes into G_L. Put the rest of the task nodes and their configuration nodes into G_R.

The basic idea of crossover is to generate new solutions by combining the parents' solutions, which in our approach means that part of the strings from par_1 and part of the strings from par_2, are transformed into new individuals, $child_1$ and $child_2$. The crossover is performed as follows, and an example is shown in Figure 21.

step 1: Randomly select a task node and generate the sub-graphs G_L and G_R from the s-graphs of both parents.

step 2: Mark the crossover sites in the parents' strings. In each string, all nodes (task nodes in T-strings, and configuration nodes in C-strings) that occur before the crossover site must belong to G_L.

step 3: Copy the left-strings of par_1 to $child_1$. Use par_2's allocation results to perform ASAP scheduling for the nodes in G_R, and convert the results into the right-strings of $child_1$. A similar process is done for $child_2$.

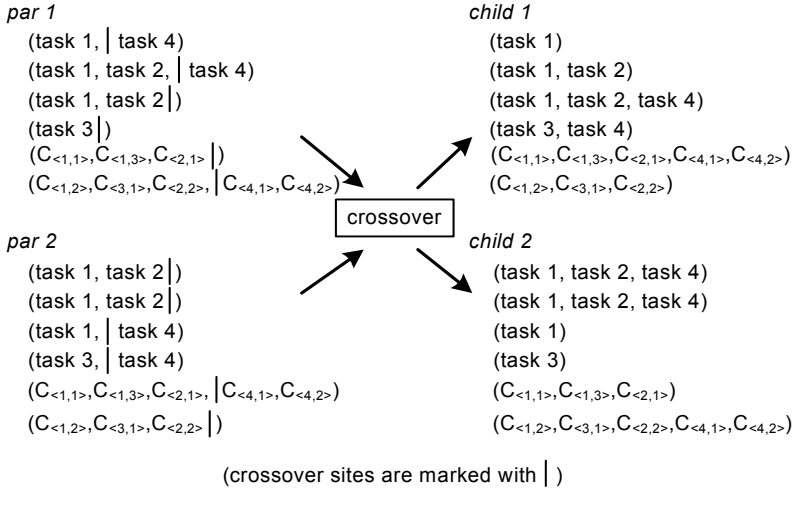


Figure 21. Crossover.

4.3.3.5 Mutation

In our case, we can separately mutate task nodes and configuration nodes, because they are two different kinds of nodes. We have totally created three different mutations and they are used together in the mutation phase.

In the first mutation, only the T-strings are mutated. A task node is randomly selected and moved to a new location. If the task requires multiple tiles, it will then be inserted into each of the T-strings that correspond to the selected tiles. Let's use *height*(V_i) to represent the height value of node V_i. Then the place in the new T-string to insert the task node must satisfy the condition that *height*(the node before V_i) < *height*(V_i) <= *height*(the node after V_i). The height value of a task node is calculated based on the s-graph as follows.

$$height(V_i)=\begin{cases}1, \text{ if } V_i \text{ is a root} \\ 1 + max(height(\text{predecessor})), \text{ else}\end{cases} \quad (8)$$

In the second mutation, only the C-strings are mutated. We randomly select a configuration node and inserted it into a new controller's equivalent C-string. The insertion place is selected in a similar way to the previous task mutation technique, but the height values of configuration nodes are calculated differently. We define the height value of a configuration node to be equal to the height value of the task node that it configures.

The last mutation is to rotate the controller assignment for the configuration nodes of a task. This is done as follows. A task node T_i is randomly selected. If it has N configuration nodes (N tiles are needed for the task). Then in the C-strings, node $C_{<i,1>}$ is replaced by $C_{<i,2>}$, $C_{<i,2>}$ is replaced by $C_{<i,3>}$, and finally $C_{<i,N>}$ is replaced by $C_{<i,1>}$. This mutation is applied only for the task that requires multiple tiles.

Each of the three mutations has its own probability to run in the mutation phase. This is arranged as shown in pseudo code in Figure 22. In the chromosome, we always guarantee that the task nodes in T-strings are ordered based on their height values, but this is not true for the C-strings and there might exist height-inverse in the C-strings (a node with a higher height is placed before a node with a lower height). When the scheduling dependences in the s-graphs are randomly modified in mutation, such height-inverse might cause cycles in the new s-graphs. Therefore, we must sort the

C-strings based on their new height values at the end of the mutation phase to ensure the feasibility of new individuals.

```
if(random_number() > mutation probability)
    if (random_number() > T-strings mutation probability)
        mutate T-strings;
    else
        rotated = false;
        if (the selected task node needs more than 1 tile)
            if (random_number() > C-strings rotation probability)
               rotate C-strings; rotated = true;
            end if
        end if
        if (not rotated)
            mutate C-strings;
        end if
    end if
    update heights and sort C-strings;
end if
```

Figure 22. Arrangement of the three mutation operators.

4.3.3.6 Evaluation and selection

The GA selection is implemented using the roulette wheel style, and fitness is measured in the same way as in [166]. In our case, because an s-graph deterministically defines the scheduling order and allocation results, the length of the critical path of the s-graph is then the schedule length. For example, in the chromosome shown in Figure 19, the schedule length can be derived from the critical path in Figure 20, $C_{<1,1>}$ => $C_{<1,3>}$ => *t1* => $C_{<2,2>}$ => *t2* => $C_{<4,2>}$ => *t4*.

4.3.3.7 Evolution strategy

We use a fixed population size during evolution. In each generation, new offspring (80% of the original size in our case) are generated and inserted into the old population, and then the worst individuals are removed in order to return the population to its original size.

To decrease the chance that solutions are trapped at a local optimal point, we dynamically modify the mutation probability. In each generation, if the average fitness of the current population is equal to the best fitness of the current population,

the mutation probability of the next generation is increased by 10% unless it reaches the upper boundary, 1.0. Otherwise, the mutation probability is decreased by 10% unless it reaches the lower boundary, the initial mutation probability. The basic idea is to increase the mutation probability when all of the solutions have converged into a single point. Therefore, more offspring will be mutated and there will be higher chances of some offspring being in the region near to globally optimal solutions.

4.4 The run-time scheduling technique

The run-time scheduling is divided into two levels: application level and task level. When a process is ready, the application-level module selects a suitable pareto profile for the process based on the current number of free tiles. Therefore, when interferences with other processes are not considered, the selected profile can guarantee that the free tiles can be optimally or near-optimally utilized. When there is no free tile, the ready process is put into a pending queue. Processes in the pending queue are sorted according to their ready time. The task with the earliest ready time is put in the front of the pending queue. When free tiles become available, the first task in the pending queue is selected, and a suitable pareto profile is then used. The following tasks in the pending queue will be selected until there is no free tile left.

In the task-level scheduling, tasks of all the running processes are managed. For each individual running process, because the execution order of its tasks is specified in the selected profile, there is no run-time overhead to schedule the tasks from the same process. In addition, because each process owns different tiles from others and all the tasks of the same process are assigned to the tiles the process owns, tasks from different processes do not overlap on the same tile. So, at any given time, tasks assigned to the same tile are all from one process and their execution order is determined at design time. Therefore, there is no need to perform cross-process task scheduling at run-time as well.

However, current DRHW has only limited configuration resources. In fact, current commercially available devices have only one configuration port and one configuration controller. This means that multiple configurations cannot be performed in parallel. It implies that although tasks from different processes do not interfere with each other, their configurations might, and their orders affect the scheduling results. For example, Figure 23 shows two running scenarios of two concurrent processes on

a device with four tiles and one configuration controller. In the first scenario, as shown in Figure 23(b), configurations of application 2 are all scheduled after that of application 1. In the second scenario, as shown in Figure 23(c), configurations are performed interleaved. The difference is obvious that in the first scenario the device is not fully utilized and application 2 suffers from long and unnecessary waiting time due to an inappropriate scheduling decision for the configurations.

In our approach, the configuration order is dynamically determined for concurrent processes in order to avoid inefficient utilization of resources as shown in Figure 23(b). The configuration scheduling is done as follows. For tasks from the same process, their configuration order is defined at design time and the run-time scheduler simply follows it. For tasks from different processes, if their configurations overlap, the task that has the earliest start time will be loaded first. The current run-time scheduler does not deal with the case that a task might occupy more than one tile, although at design time our schedulers take it into account. This problem can be solved with run-time task allocation algorithms, which will be studied in the future.

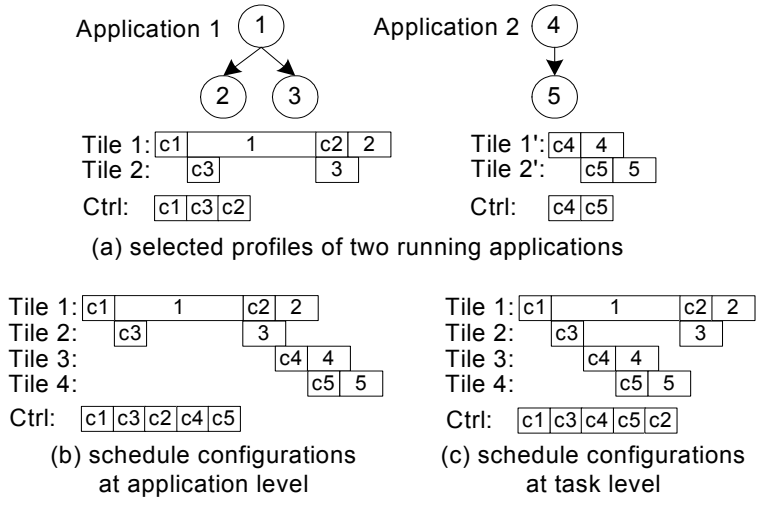


Figure 23. Two scheduling scenarios.

4.4.1 Configuration locking technique

The main goal of our configuration locking technique is to improve the efficiency of configuration caching. Intuitively, caching the most frequently used tasks has better

opportunities to reduce the total number of reconfigurations and thus reduce the configuration overhead. This is shown in a motivation example in Figure 24. We assume that application 1 and application 2 are two independent periodic applications, as shown in Figure 24(a). When they are mapped onto a device containing four tiles, no reconfiguration is needed except the initial ones, as shown in Figure 24(b). However, when they are mapped onto a resource-limited device that contains only three tiles, how to reuse loaded tasks can result in significant difference. Figure 24(c) shows that an inappropriate scheduling causes application 1 to miss its deadline. At step 12, task 4 is required. Because both tile 1 and tile 3 are being used, tile 2 is selected and task 2 is evicted for loading task 4, which causes an additional configuration of task 2 and thus the deadline of application 1 is missed. If we allow the scheduler to always keep the two most frequently used tasks (in this case task 1 and task 2) on the device, task 2 will not be evicted, and task 4 will be allocated to tile 3 and later loaded at step 13. The result is that both applications can meet their deadlines, as shown in Figure 24(d).

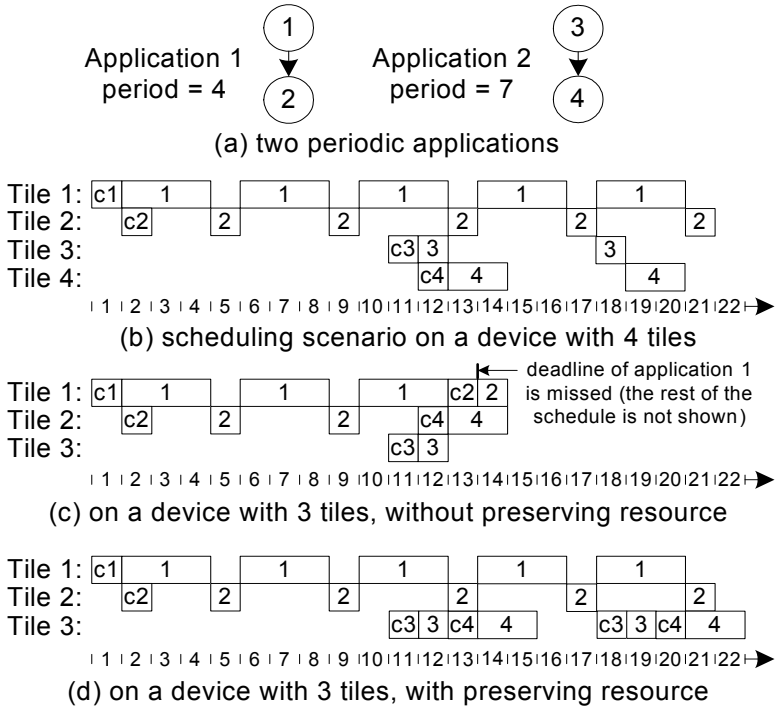


Figure 24. A motivation example of preserving resources in configuration caching.

The motivation example shows the benefit of locking the most frequently used tasks. However, it is usually infeasible to decide at design time which tasks are the most frequently used. For example, when using a smartphone, a user might make a phone call for 10 minutes and then watch movies for the next 30 minutes. Therefore, some wireless communication tasks will be intensively used in the beginning, but video decoding tasks for the following 30 minutes. The situation will change if a video call is started later on. To solve this problem, in our locking technique the run-time scheduler dynamically counts the times that tasks are executed (when an application is finished, the counts of its tasks are reset to zero). A number of frequently used tasks are always locked to avoid being evicted by any ready task. The number of tasks to preserve (N_{tp}) is decided by designers at design time. Because one task is locked on one tile, N_{tp} also means the number of tiles that are used to preserve the most frequently used tasks. The number must be less than the total number of tiles, $N_{tp} < N_{tile}$. Otherwise no tile can be assigned to any ready task.

In the run-time scheduler, a reuse module dynamically checks if any previously loaded task can be reused for any task of running or pending processes. Only if a task is cached and ready to run (all its predecessors have finished) is it dispatched. This is done at task-level scheduling before configurations are scheduled. It should be noted that when the scheduler checks which task can be reused, it goes through all the tiles, not only the preserved tiles. The intention of using preserved tiles is to increase the cache-hit rate by locking the most frequently used tasks on the device, and this does not prevent us from reusing other tiles.

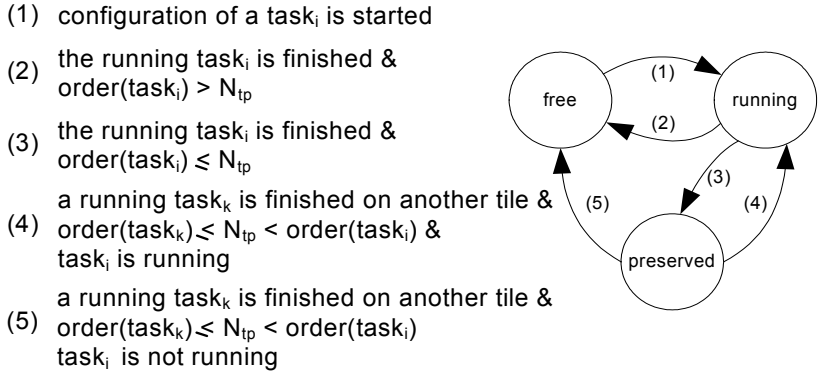


Figure 25. Tile state transition diagram.

Each tile has its own state, which is maintained by the run-time scheduler and used during the scheduling process. There are three states: *free*, *preserved* and *running*. The state *free* means that any task can be assigned to the tile. The state *running* means that a task is assigned to this tile and this task is being loaded, or loaded but not running, or running. The state *preserved* means that a task has finished on this tile and in the future only this task can be assigned to this tile. The state transition diagram is shown in Figure 25. The function $order(task_i)$ returns the position of $task_i$ in a list, which is sorted with decreasing order using the times that tasks are executed. Thus, $order(task_i) \leq N_{tp}$ means that $task_i$ is now one of the N_{tp} most frequently used tasks. The run-time scheduler manages the states of the tiles and makes scheduling decisions based on these states. When a task is finished, the run-time scheduler updates the execution counts, sorts the list and decides which tiles are now in the state *preserved*. For example, when a task is finished and it is now one of the most frequently used tasks, the tile on which the task is assigned will be put into *preserved*, as in transition (3), otherwise the tile becomes *free*, as in transition (2). Transitions (4) and (5) mean that another task, $task_k$, assigned to a different tile has became more frequently used and the task, $task_i$, on this tile is not one of the N_{tp} most frequently used tasks.

4.5 Case studies

The case studies are divided into two parts. The first part concentrates on evaluating the three static scheduling techniques that have been described in section 4.3. Both randomly generated cases and task models derived from practical applications are used. Although configuration prefetching is the main technique used in these approaches to hide configuration latency, the purpose of this case study is not to evaluate the benefit of using configuration prefetching, as it has been done in others' work [115, 116]. Our main focus is to quantitatively study the scheduling efficiency of these algorithms. This is done by comparing the computer run-time and the deviation from optimal solutions. A quantitative comparison of our scheduling techniques with the existing static DRHW scheduling techniques, which are summarized in section 2.3.3, is not practical, because all these techniques more or less focus on different problems. The scheduling algorithm in [82] is designed for a HW/SW partitioning approach, where tasks can also be mapped onto processors. The goal of the static DRHW scheduling in [120] is to identify the configurations that have the most negative impact on the system performance. In [98], a list-based approach is

used, but it is for task grouping. The genetic algorithm in [134] is designed to solve the resource fragmentation problem.

In the second part, the configuration locking technique is evaluated with practical applications and compared with the results where configuration caching is used but no locking effort is applied. Although our locking technique is embedded into a run-time task scheduling, comparison with other run-time schedulers is not available. The reason is that most of such schedulers focus on how to efficiently solve the task allocation problem in 2D DRHW devices, which however suffer from a very critical problem, e.g., run-time routing. In addition, some run-time schedulers assume that DRHW tasks are preemptive, which however results in substantially high switching overhead.

4.5.1 Evaluation of the static scheduling approaches

The list-based heuristic scheduler and the GA-based scheduler are implemented in C++, and they are included in our design space exploration toolset for DRHW [33]. The toolset can also automatically generate the constraint models for given task graphs. The constraint models are solved using a third-party tool, SICSTUS finite domain solver [168]. The computing environment is a workstation equipped with two AMD Opteron 252 processors, but only one processor actually contributes to the performance results, because no parallel programming is used in the implementation.

4.5.1.1 Computation effort of the CP-based approach

We used 10 randomly generated task graphs with each graph containing 10 tasks. These graphs had different levels of depth and different tree structures, so they could be seen as representations of widely different applications. The number of required tiles of an individual task was randomly generated with uniform distribution in the range of [1, 3], and the total nodes of these graphs are in the range of [28, 32]. Different device models were used by setting the number of tiles, N_{tile}, to iterate from 4 to 7 and the number of controllers, N_{ctrl}, to iterate from 1 to 3. The average ratio of the task configuration time to the average of the task execution time was set to 0.2. Therefore, in total we had 120 test cases.

The results in terms of computer run-time are shown in Figure 26. It can be seen that the consumed computer run-times are widely distributed, from 21 ms in the fastest

run to nearly 8 hours in the slowest run. On average, each CP-based scheduling took 8.4 minutes. The averaged computer run-time of each device is marked with the connected triangle points. It can be seen that for devices of more tiles and more controllers the required effort to find the optimal solutions decreases. This is because with more tiles and more controllers, fewer conflicts are generated and therefore the amount of backtracking is reduced, which helps to reduce the searching effort. The results also show that for a single task, the required computer run-time may vary significantly. For example, the two points marked with arrows are from the same task but on different devices, (4,1) and (7,1). The former is the slowest run, but the latter shows that the optimal result can be found within 35 ms.

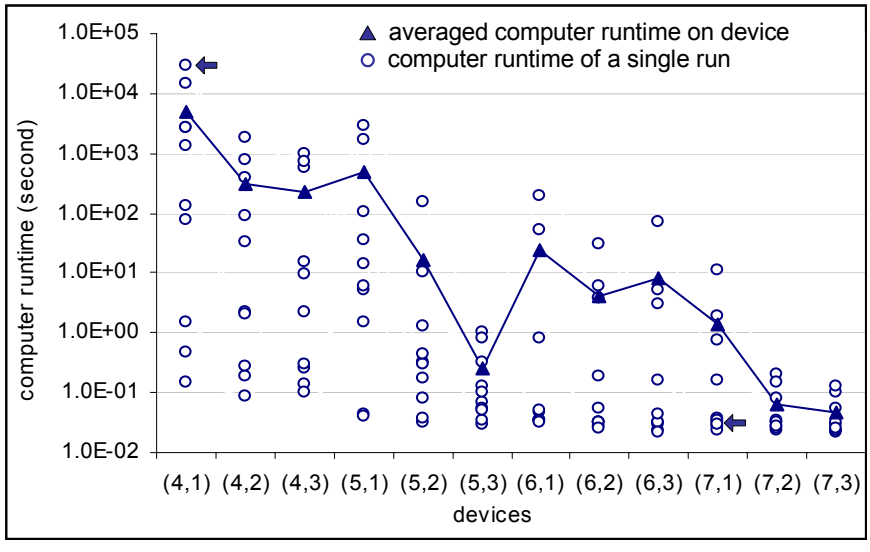


Figure 26. The consumed computer run-time of the CP approach.

To study the scalability of the approach, we selected the task graph that had the fastest run, and gradually added more nodes and applied the CP-based approach to them. When we slightly increased the number of tasks to 16 and the total number of nodes to 51, the required computer run-time exceeds 20 hours. Therefore, the CP-based approach is less favorable in practical applications because of its unpredictable computer run-time. However, with the guarantee to produce optimal solutions, it can be used as a reference to evaluate other approaches.

4.5.1.2 Accuracy and performance of the sub-optimal approaches

The list-based scheduler and the GA scheduler are sub-optimal approaches. The purpose of this case study is to evaluate how close their results are when compared to optimal ones. Because the optimal solutions of the randomly generated task graphs have already been found using the CP approach, as presented in the previous subsections, we used the same task graphs and settings. For the GA-based scheduling approach, the parameters used in the test are as follows:

- initial mutation probability: 0.1
- crossover probability: 0.95
- number of individuals in one generation: 100
- number of generations (evolving steps): 100.

Table 4. Comparisons of the GA with other approaches.

	Scheduling Deviations		**Average Computer Run-time**		
	$\frac{GA-CP}{CP}(\%)$	$\frac{List-CP}{CP}(\%)$	GA	List	CP
DAG1	2.59	8.38	0.93 sec	4.49 ms	8.45 min
DAG2	1.61	3.10	0.98 sec	4.83 ms	23.35 min
DAG3	1.58	1.84	0.79 sec	3.16 ms	0.20 min
DAG4	0.11	1.32	0.94 sec	5.07 ms	0.11 min
DAG5	0.31	2.19	0.95 sec	4.86 ms	0.02 min
DAG6	0.36	3.19	0.86 sec	5.35 ms	0.01 min
DAG7	0.12	11.11	0.97 sec	4.72 ms	6.15 min
DAG8	0.36	5.02	0.87 sec	5.56 ms	41.59 min
DAG9	0.28	0.98	0.91 sec	3.88 ms	4.18 min
DAG10	0.10	0.62	0.88 sec	6.21 ms	0.01 min
Average	0.85	3.78	0.91 sec	4.81 ms	8.41 min

The scheduling deviations are averaged for each individual task graph and presented in Table 4. For GA, 10 runs are performed for each case, and the smallest schedule length of the 10 runs is used in calculating the scheduling deviations. It can be seen that the results of the GA approach are significantly better than the results of the list-

based approach, as the average deviation in the former case is less than 1%, but the average deviation in the latter case is about 5%.

For the 10 GA runs on each setting, we calculate the coefficient of variation (the standard deviation divided by the mean) to study the repeatability of our GA approach. The results are shown in Figure 27. It can be seen that in all cases, the coefficients of variation are below 2.5%. In fact, the majority are in the range of [0.5%, 1.5%]. This shows that our GA approach has rather good convergence.

The average computer run-times are also presented in Table 4. When comparing the accuracy, it can be seen that the GA-based approach is consistently better than the list-based approach. The average deviation in the former case is 0.85%, but 3.78% in the latter case. Considering the computer run-time, the list-based approach is the most efficient, requiring less than 5 ms on average. On average, a single GA run took less than 1 second, but a single CP-based scheduling took more than 8.4 minutes. Although the list-based approach takes a significantly shorter time, in the range of milliseconds, we consider using the GA approach to be much more beneficial, because of its higher accuracy and very tolerable computer run-time. In addition, the GA approach offers the flexibility of choosing between accuracy and computer run-time. If more generations are evolved, more accurate results are likely to be achieved.

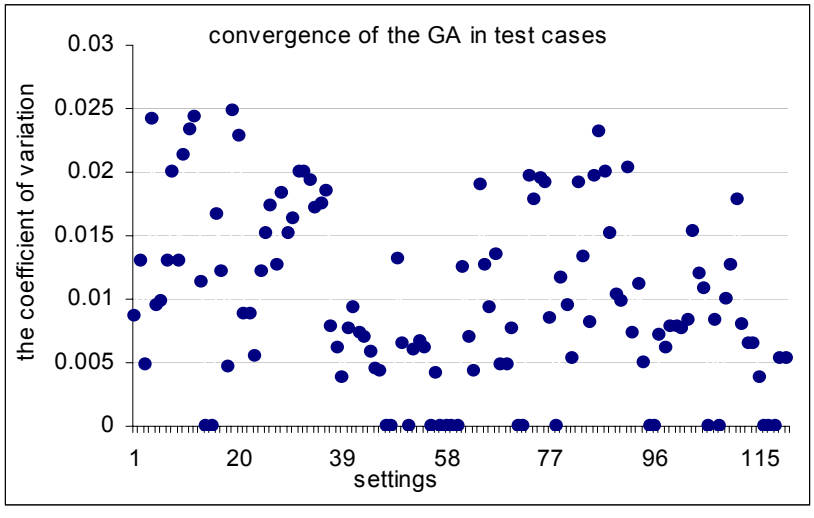


Figure 27. The coefficients of variation of the GA in test cases.

4.5.1.3 Scalability of the sub-optimal approaches

The sub-optimal approaches have also been tested with larger task graphs to study their scalability. The test task graphs were generated using TGFF [169], but they had different sizes. The same TGFF settings were used so that they had similar dependences and parallelisms. The task execution times were randomly generated and the average configuration time was set to about half the average task execution time. In addition, tasks were set to require two tiles on average. The device model with seven tiles and two controllers was used.

Some GA parameters were set differently from the previous test. Due to the fact that these task graphs had different sizes, instead of using a fixed population size we set the population size to be about 50% of the total number of nodes. To ensure that the results were at least suboptimal, the termination criterion was so that the GA stops when all of the results converge into a single point for five continuous generations (the average fitness is equal to the best fitness of five continuous generations). In all of these graphs, the CP-based approach could not find the optimal solutions within two days. Therefore, we used the GA results generated from evolving 1500 generations as the reference values.

The GA approach was performed 10 times for each task graph. The best GA results out of 10 runs and the list-based results are compared in Table 5. It can be seen that the GA results are consistently better than the list-based approach, 8.6% better on average. For each GA run, we have also collected the number of generations that have evolved until the convergence criterion is met. These numbers are averaged and also shown in Table 5. It can be seen that more generations are required to converge the GA for larger task graphs and this effort is approaching the linear property, about 40 additional generations are needed for every 20 additional nodes.

Table 5. Scalability results.

Task set	**Total nodes**	**Num. tasks**	$\frac{GA - Ref}{Ref}$ **(%)**	$\frac{List - Ref}{Ref}$ **(%)**	**Avg. stop generation**
1	49	15	6.42	13.49	96
2	64	20	3.9	18.32	137
3	80	25	5.56	22	199
4	87	30	3.9	6.78	150
5	105	35	3.37	11.29	231
6	123	40	2.73	5.4	280

4.5.1.4 Experiments with practical applications

We have also tested the three different schedulers with five practical applications. Each application is divided into a number of dependent tasks. VHDL codes for these tasks are manually generated. The required resources and the execution time are derived from the synthesis results and the simulation results. The brief explanations of these applications are as follows.

- Sobel: Image sharpening application using sobel masking. The execution time is based on processing 256 x 256 pixels. There are six tasks in this application.
- Sobel & Noise: Image sharpening (sobel masking) and noise reduction application (noise reduction is performed separately in each color domain). The execution time is based on processing 256 x 256 pixels. There are 17 tasks.
- JPEG Encoder: A JPEG Encoder. The compression is performed in parallel for the luminance and the two chrominance spaces. The execution time is based on processing 256 x 256 pixels. There are 11 tasks.
- MPEG Encoder: The core functions of MPEG2 encoding. The execution time is based on encoding a frame of a CIF picture (352 x 288 pixels). There are seven tasks.
- WCDMA detector: Part of a Wideband CDMA decoder. The execution time is based on processing four slots of data (Each slot contains 2560 chips). There are four functions, including an adaptive filter, a channel estimator, a multi-path combiner and a correlerator.

Table 6. Practical applications for testing static scheduling techniques.

Task set	**Total nodes**	**Num. tasks**	**Schedule length (us)**			**Computer run-time**			**Average GA stop generations**
			CP	List	GA	CP	List	GA	
Sobel	14	6	12750	12750	12750	20 ms	1.3 ms	20 ms	10
Sobel&Noise	39	17	17480	18088	17480	30 h	15.3 ms	1038 ms	80
JPEG encoder	26	11	11248	11552	11248	25 h	6.1 ms	448 ms	62
MPEG encoder	17	7	8626	8721	8626	17.3 s	13.2 ms	155 ms	37
WCDMA	8	4	5408	5408	5408	9.5 ms	400 us	10 ms	6

The results are presented in Table 6. For GA runs, the shortest schedule length of 10 runs is used. It can be seen from Table 6 that in all these cases the GA scheduler is able to find optimal solutions. In addition, the computation effort of using GA is considerably less than that of using CP, in which two of the five cases require more than 24 hours to solve. Similarly as in the randomly generated task graphs, the list-based scheduler is the most efficient but produces the least accurate solutions.

4.5.2 Evaluation of the configuration locking technique

We used four real applications to validate the configuration locking technique. These four applications are the same as used in the previous case study, as presented in section 4.5.1.4. The "sobel & noise" image sharpening application was not used, because its functionality was similar to that of the "sobel" image sharpening application. In a real case, only one of these two would be applied, and we selected the simpler one for testing. The GA-based static scheduler was used to generate the profiles. For each application, we generated the first profile using the setting of one tile, and then more profiles by gradually increasing the number of tiles. We stopped when the configuration latency could not be reduced with more tiles. We assumed that each tile had an equal amount of resources as in Xilinx XC2V250 FPGA [27]. (Some tasks were too big to fit into smaller devices.) A brief explanation of the settings for the four applications is shown in Table 7.

The run-time scheduler is also implemented in C++. Pareto profiles, application DAGs and device settings are given in text files. The scheduler reads these inputs and simulates a pre-defined simulation period. Different statistic results are automatically collected during simulation and saved in a text file at the end of simulation. The communication overhead was ignored during simulation. We assumed that all these applications were running periodically. Devices ranging from 6 tiles to 14 tiles were explored. In addition, we tried to use a different number of preserved tiles. The value N_{tp} was set to sweep from 0 to $N_{tile} - 1$. Therefore, the setting, $N_{tp} = 0$, means that configuration locking is not applied. In total, there were 90 different device settings. In the following context, we use the notation (N_{tile}, N_{tp}) to represent the device with N_{tile} tiles and N_{tp} preserved tiles.

For each setting, we randomly generated the starting time of each application and performed 10 simulations with a different initial seed each time. Each simulation ran

for 10^6 simulation cycles. Results of the 10 simulations are averaged and shown in Figure 28. It can be seen that when more tiles are available more processes can finish before deadline. However, the improvements without using our locking technique are very limited. For example, at (10,0), about 17 processes can finish before deadline, but by preserving four tiles about 96 processes can finish before deadline. This is mainly because more tasks are reused, 430 compared to 325. In addition, the result at (10,4) is much better than that at (14,0), which shows that simply using more computation resources is not as efficient as preserving resources for dedicated purpose.

Table 7. Practical applications for testing the configuration locking technique.

Application	**Pareto profiles**			**Periodicity**
	num of profiles	profile ID	schedule length (us)	
Sobel	3	1	25056	processing two pictures with size of 1024 x 768 in every second period = 41667 us
		2	16896	
		3	15612	
JPEG encoder	3	1	38806	processing two pictures with size of 1024 x 768 in every second period = 41667 us
		2	23851	
		3	22567	
MPEG encoder	3	1	26011	processing CIF video encoding period = 33333 us
		2	16486	
		3	14958	
WCDMA	3	1	13636	processing 15 slots of data in 10 ms period = 10000 us
		2	8616	
		3	8072	

It is obvious that using more tiles to preserve tasks is not always beneficial. This is because when only a smaller amount of tiles are available for free allocation, ready tasks have to be put into pending. For example, after (14,6) using more preserved tiles

can still increase the number of reused tasks, but the number of processes that can finish before deadline also starts to decrease. In addition, at a certain point the system performance starts to drop sharply, because many ready tasks are competing for small amount of free tiles. This is more visible when reviewing the average waiting time as shown in Figure 29. The waiting time is defined as the difference between the time a process starts to run and the time the process is ready to run. The execution time is defined as the difference between the time a process finishes its execution and the time the process starts to run. It can be seen from Figure 29 that the average waiting time is decreasing initially when more resources are reserved for locking tasks. However, at the point when the amount of tiles that are allowed to be shared becomes too small, processes need to spend more time waiting for the tiles to become available.

It is not surprising to see that when resources are too limited, $N_{tile} < 7$, our locking technique is not effective. This is because the ready tasks already have to compete for the small amount of free tiles, using some tiles to preserve loaded subtasks can make the situation only worse. On the other hand, when there are too many resources, $N_{tile} > 13$, our locking technique tends to be less effective. This is because less swapping will happen, and thus trying to avoid swapping is not effective. The case study shows that our locking technique is not effective when resources are too limited or too prosperous. However, in other cases, using the proper amount of tiles to preserve loaded subtasks can significantly improve the system performance.

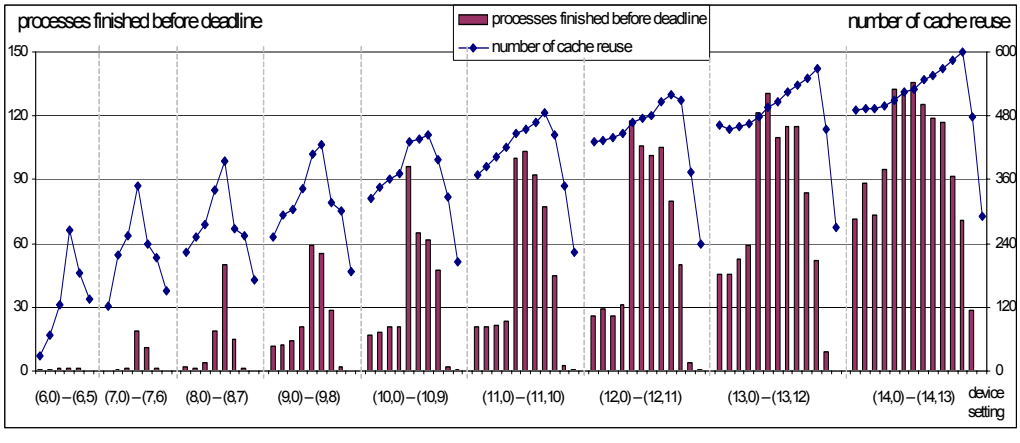


Figure 28. Scheduling results of real applications (average over 10 simulations of using different seeds).

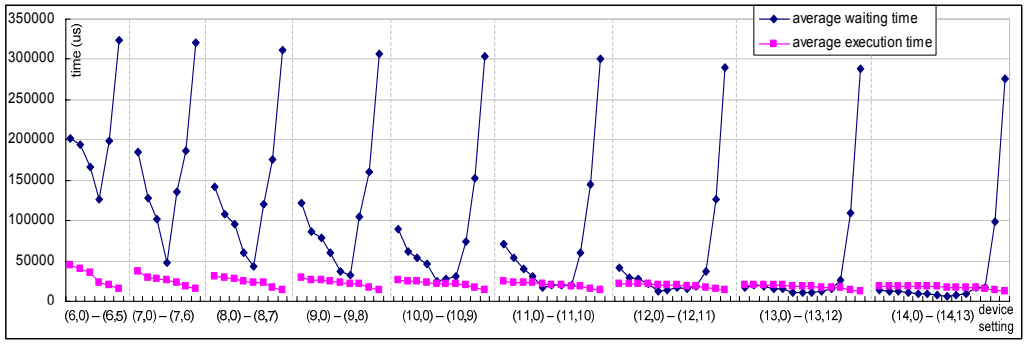


Figure 29. Average waiting and execution time of real applications (average over 10 simulations of using different seeds).

4.6 Discussion

A major benefit of using DRHW is that they have high silicon reusability. This allows multiple tasks to be running simultaneously in the same DRHW or multiplexed in time domain. With the multitasking feature, tasks should be carefully managed to efficiently utilize the device. The DRHW scheduling problem is similar to multiprocessor scheduling, but it is more complicated because both task allocation and the configurations have to be considered.

In this work, a quasi-static task scheduling approach has been used as the basic scheduling framework. At design time, configuration prefetching is applied. The main focus is the development and evaluation of three approaches with different searching strategies. The first one is a heuristic approach based on a traditional list-based scheduler. The second one is based on a full-domain search. Constraint programming and a third-party solver are used. The last one is based on a guided random search. A set of customized GA operators are developed.

Randomly generated task graphs and practical applications have been used in the case studies. The results show that the list-based approach is the most efficient but the least accurate. Although the CP-based approach can be guaranteed to generate optimal solutions, the required computer run-time depends on many things, including the device settings and the number of tasks, which makes it very unpredictable. For the CP-based approach, constructing a custom labeling algorithm taking into account the domain knowledge might considerably reduce the searching effort. However,

considering the problem itself is NP-hard, using a custom labeling algorithm cannot fundamentally change the characteristics of the CP-based approach. In comparison, the GA-based approach shows high accuracy and reasonable efficiency. In addition, the GA-based approach has shown good convergence and almost linear scalability in terms of the number of generations required to converge.

At run-time, a novel configuration locking technique is applied. It can effectively reduce the configuration overhead by reducing the amount of required configurations. The idea is to always lock the most frequently used tasks on the device so that they have better chances to be reused. The most frequently used tasks are dynamically tracked, because the run-time status depends on users' behavior, which cannot be decided at design time. However, reserving too much space on DRHW (keeping too many of the most frequently executed tasks) will reduce the amount of resource that could be shared by the rest of the tasks. This might result in more reconfigurations and eventually degrade the system performance. The performance improvement of using configuration locking and the negative impact of resource over-reservation have been studied with a number of real applications. The results show that when resources are not too limited or too prosperous, preserving proper amount of tiles to lock the most frequently tasks can significantly improve the system performance.

5. Novel techniques to reduce the configuration overhead

The main drawback of using DRHW is the configuration overhead related to each reconfiguration process. In Chapter 4, we presented different scheduling techniques that can reduce the effect of the configuration overhead. However, in some cases scheduling cannot effectively reduce the impact, and in addition the cost of each reconfiguration process remains unchanged (the same amount of energy and latency has to be paid). In this chapter, we present two novel techniques, one for reducing configuration latency and another for reducing configuration energy.

5.1 Configuration parallelism

We refer to the first technique as configuration parallelism [33, 37]. The principle is to divide the entire configuration-SRAM into several small segments and enable the configuration data to be written into the different segments simultaneously. The benefit is that more task parallelism can be exploited when such configuration parallelism is available. In addition, we demonstrate that combining configuration parallelism and configuration prefetching can more effectively hide configuration latency.

5.1.1 Motivation

We use a simple example (shown in Figure 30) to demonstrate how task parallelism can be better exploited with configuration parallelism and how additional improvements can be achieved together with configuration prefetching. Figure 30(a) shows three dependent tasks. We assume that computation time and configuration time are all equal in the three tasks. Current DRHW has only a single configuration controller, so multiple configurations can be performed only in sequence. Its effect is shown in Figure 30(c). The effect of configuration of task 2 can be eliminated by prefetching, but task 3 has to be delayed because its configuration cannot start earlier. However, if we could use two controllers to reconfigure different portions of the DRHW in parallel, execution of task 3 can start immediately after its predecessor, task 1, finishes, as depicted in Figure 30(d). When we assume that the execution time and the configuration time are equal, using two controllers can speed up the system

by 25% compared to using only one controller in the case. In this the following section, we present an implementable model to realize the configuration parallelism.

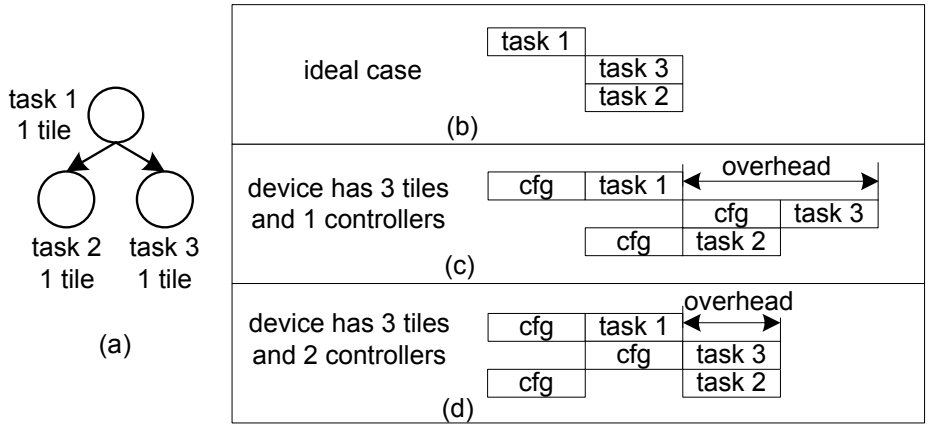


Figure 30. A simple example to illustrate the benefits of configuration parallelism.

5.1.2 The parallel reconfiguration model

The basic idea of our parallel reconfiguration model is to divide the entire configuration-SRAM into separated individual segments and use multiple configuration controllers to control the multiple segments in parallel. The parallel reconfiguration model and the way to integrate it into a multi-core SoC platform are depicted in Figure 31. Different units are connected via the communication network. The in/out memory serves as the shared memory by which the reconfigurable logic communicates with other units. The local memories are attached to the reconfigurable logic through a memory crossbar. They are used as shared memories for the tasks that are mapped onto the DRHW. The configuration manager controls the tasks and their reconfiguration processes by sensing the status signals and sending the control signals. For different applications, the reconfiguration manager behaves as a run-time scheduler operating in a first-come-first-serve fashion. The configuration locking technique can be placed on it. For each individual application, because its tasks have static dependence, reconfiguration decisions of these tasks can be pre-computed at the design time and stored in a table, from which the reconfiguration manager needs only to fetch the next decision during the run-time. The DRHW is used to accelerate computation-intensive tasks. When an application, which usually consists of a number

of dependent tasks, needs to be accelerated, a processor dumps the data that is to be processed into the in/out memory, and then calls the configuration manager to start loading tasks and then executing the tasks. After the set of tasks have finished, the last task writes the results to the in/out memory, and the master or other modules can read the results for further processing. To avoid data hazard between different applications, the in/out memory can be made as a pair of separate input and output memories.

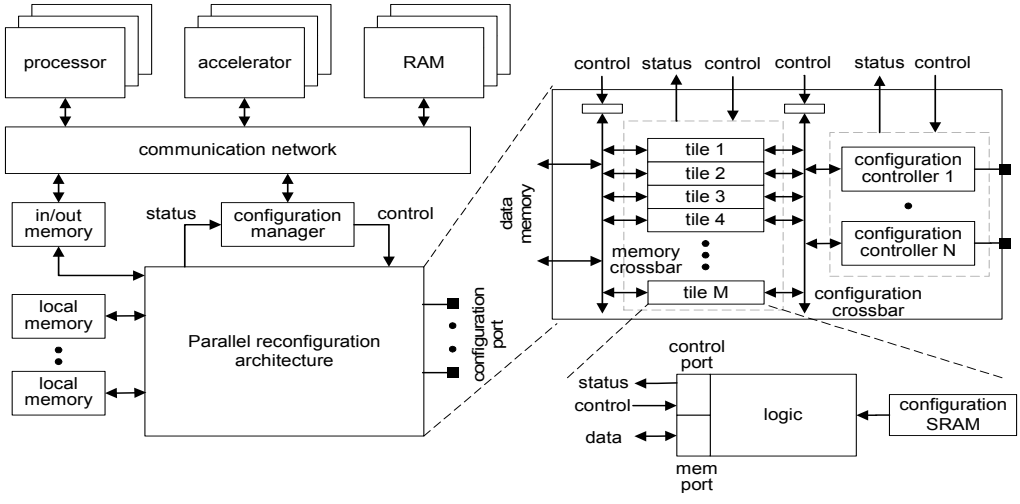


Figure 31. The parallel configuration model.

The key characteristics of the parallel reconfiguration model are as follows. Firstly, the reconfigurable logic consists of a number of continuously connected homogeneous tiles, and each tile consists of the circuit and its own configuration-SRAM that controls the circuit. A task that requires *m* tiles of resources can use any set of *m*-connected tiles. Each tile has its own control port and memory port. If a task requires more than one tile, only one pair of the ports is used. Secondly, a multiplexer-type memory crossbar is used to connect the tiles to the directly accessible local memories. Data transfers between tasks always go through the local memory banks instead of directly passing the data through the boundaries between tiles. Therefore, routing contention can be avoided when tasks are relocated. Thirdly, vertical lanes crossing the boundaries of tiles are used only for a task that is mapped onto the tiles. For tasks that are mapped onto adjacent tiles, the cross-boundary lanes are disabled by simply turning off the pass transistors that control these lanes, so possible glitches during reconfiguration will not affect any running task. Finally, a multiplexer-type configuration crossbar is used to connect the configuration-SRAMs

of the tiles to a number of parallel configuration controllers. The crossbar is controlled by the configuration manager, which can ensure that any configuration-SRAM can be accessed by any controller but only one at a time. Thus, reconfigurations can be performed in parallel on different tiles.

5.1.3 Evaluation of configuration parallelism

We performed three different case studies. The first one focused on the benefits of combining configuration parallelism and configuration prefetching. The second one focused on studying the effects of using different number of tiles and different number of controllers. 10 randomly generated DAGs (each DAG contains 10 tasks) were used in these two evaluations. To avoid being misled by non-optimal values, the CP-based optimal scheduling approach was applied in these first two evaluations. The last one was an evaluation of different device settings using real applications. Due to the size of the applications, the CP-based approach could not produce the result of a single device setting within 48 hours. Therefore, the GA approach was applied in the last case study.

For the randomly generated DAGs, they had different levels of depth and different tree structures, so they could be seen as representations of widely different applications. Tasks were set to have from 0 to 3 successors, but on average 1 successor. Different devices were evaluated by setting the number of tiles, Ntile, to iterate from 3 to 7 and the number of controllers, N_{ctrl}, to iterate from 1 to 3. Resource utilization of tasks was set as $\lceil RR_j / ST \rceil \in [1,3]$, and the average ratio was 2. Three settings of the configuration latency, *CL*, were used in the case studies. Corresponding to the three settings, the ratio of average configuration time to the average execution time, g, is 0.2, 0.5, and 1.0 respectively. The value of g is calculated as:

$$\sum_{j=1}^{10} \left[CL * (\lceil RR_j / ST \rceil) / (10 * EX_j) \right] = g \qquad (9)$$

In the following context, we use (N_{tile}, N_{ctrl}) to refer to the device with N_{tile} tiles and N_{ctrl} configuration controllers. In addition, we use the term (N_{tile}, N_{ctrl}, *P*) to refer to configuration prefetching, and (N_{tile}, N_{ctrl}, *NP*) for non-prefetching. For non-prefetching scheduling, the results were generated from the CP-based approach by extending the constraint model with an additional constraint that forces configurations of a task cannot start before all the predecessors of the task have finished.

5.1.3.1 Combining configuration parallelism and prefetching

Scheduling results for the 10 DAGs are averaged and shown in Figure 32. The first three graphs present the results for different settings of the ratio of average configuration time to the average execution time, *g*. The values were set to be 0.2, 0.5 and 1.0 separately. Results in these graphs are divided into four sections with each section representing one setting of the number of tiles, N_{tile}, and the values in each section represent the speedups when compared to the scheduling results with (N_{tile}, 1, *NP*). The left two columns represent the results of non-prefetch scheduling but with configuration parallelism, so these are the pure contribution of using only configuration parallelism. The columns in the middle represent the results of using only configuration prefetching, in which only a single configuration controller is used. The right two columns represent the achievable speedups of combining prefetching with different levels of configuration parallelism. Figure 32(d) presents the average configuration overheads of the test cases. The configuration overhead, as illustrated in Figure 30, is defined as the difference between a task schedule result and the optimal schedule result, in which the configuration latency is zero. The latten one can be generated in our scheduling approaches by setting the parameter *CL* to 0.

The results shown in Figure 32 clearly indicate that the benefit of using configuration parallelism is not as linearly increased as the number of controllers. Experiments with four controllers have also been carried out, but no extra improvement can be achieved. This non-linear effect is mainly because of the limitation of the take parallelism and the intra-task parallelism of the randomly generated task graphs. In fact, this is common in most other systems. For example, in multi-processor environment, double the number of processors usually does not double the performance. In this case study, we set that each task, on average, required two tiles and had one successor. Therefore, the significant improvements appear with two controllers and most of such improvements saturate when three controllers are used. For applications that have more successors or require more tiles on average per task, increasing the number of controllers will then become useful.

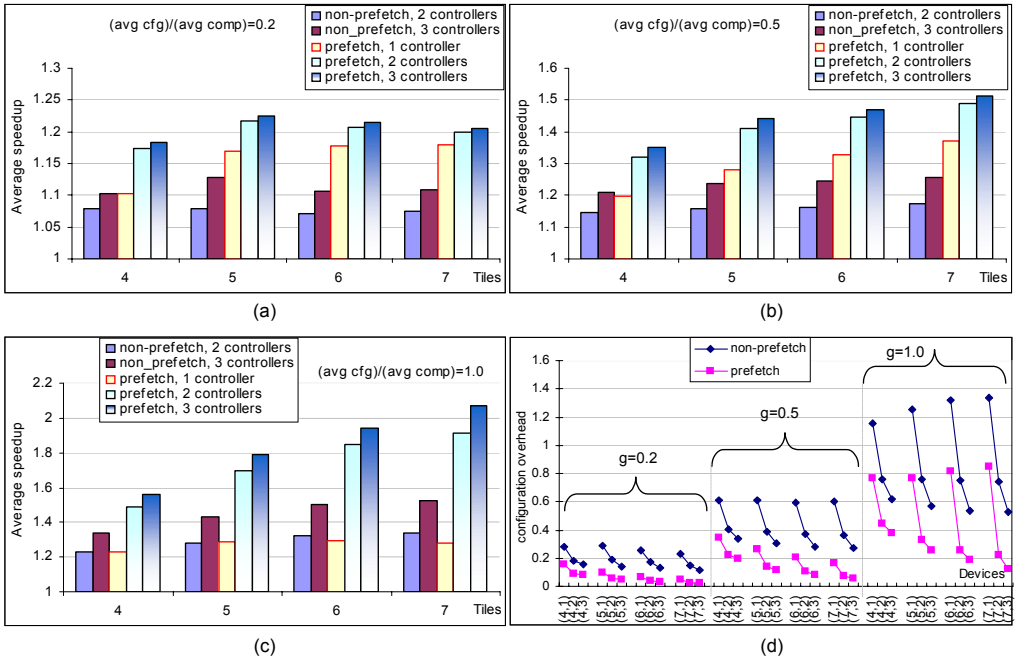


Figure 32. Average results of the randomly generated task graphs.

It is obvious that combining configuration parallelism and prefetching can always bring better results, because the two are orthogonal techniques and each one has a positive contribution. This is verified from the results shown in Figure 32. For example, results on (N_{tile}, 2, P) are always better than that on (N_{tile}, 1, P) and also better than that on (N_{tile}, 2, NP). Prefetching alone is already an effective approach. If we compare the (N_{tile}, 1, P) and the (N_{tile}, 1, NP) at $g = 1$, the reduction of overhead is about 37%. Furthermore, with 2 controllers, (N_{tile}, 2, NP), an additional 38.4% of overhead can be reduced (total reduction is 74.5%). In order to study how configuration parallelism can make prefetching more beneficial, we compare the speedups of the (N_{tile}, 3, P) with that of the (N_{tile}, 1, P). We refer to these differences as the additional speedups of using configuration parallelism. Because most speedups saturate when $N_{ctrl} = 3$ in the case studies, as explained in the previous section, the differences between (N_{tile}, 3, P) and (N_{tile}, 1, P) can also be seen as the maximally achievable additional speedups for prefetching. It can be seen that when the average configuration latency is relatively large compared to the average computation time ($g = 1.0$), more additional speedup can be achieved when more tiles are used. This is because using more tiles makes it possible to exploit more task parallelism and thus configuration parallelism can be more useful. However, when the ratio, g, is relatively

small (g = 0.2), the additional speedup decreases as the number of tiles increases. This is because for a small value of g, a long task execution can effectively hide several short configurations even without using configuration parallelism. Using more tiles makes it more possible that executions and configurations of different tasks can be done in parallel, and thus reduces the effect of using configuration parallelism.

5.1.3.2 Speedups of using more tiles and more controllers

A common approach to improving performance is to include more computation resources to achieve high parallelism, either at the task level or at the operation level. In DRHW, using more tiles enables more tasks to run in parallel. However, if configurations of these tasks can be done only in sequence, it is not likely that such parallelism can be exploited even if more tiles are used. In this section, we study how configuration parallelism helps in such cases. Discussions are based on the prefetch scheduling results. Because using 4 tiles and 1 controller is the lowest setting in our cases, we use the results of (4, 1, P) as the reference values. The average results of the 10 DAGs are shown in Figure 33.

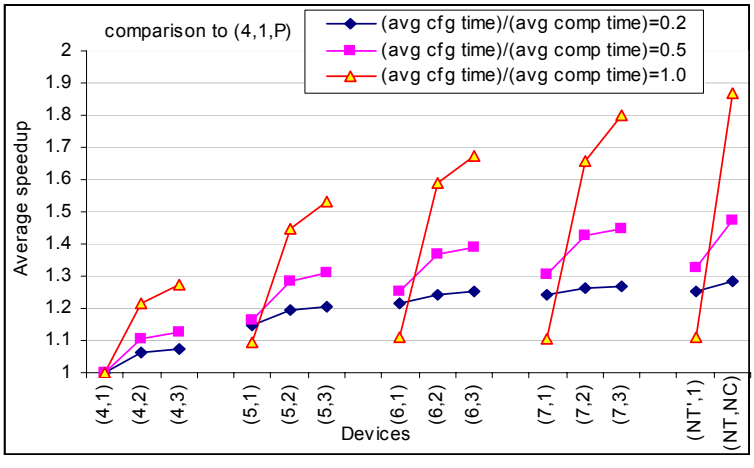


Figure 33. Speedups of using more tiles and more controllers.

Two extra types of results were generated. The first was for the cases that there was no resource limit on the number of tiles but also there was no configuration parallelism (using a single controller). So, these were the best possible speedups for increasing only the computation resources. The second was for the cases that there was no resource limit on both the number of tiles and the number of controllers. So,

these speedups presented the globally best speedups. Results are separated shown with the label (N_{tile}', 1) and (N_{tile}, N_{ctrl}) in Figure 33. It should be noted that we use N_{tile}' and N_{tile} to mark that they might be different values.

It can be seen that without configuration parallelism, increasing tiles has only limited capability for speedups, and using more controllers can always bring additional improvements. This is especially obvious when we compare the results of (N_{tile}', 1) and the results of (N_{tile}, N_{ctrl}). The first ones show that if we increase only tiles, the best speedups are 1.26, 1.33, 1.1 for g = 0.2, 0.5 and 1.0 separately. However, with configuration parallelism, the best possible speedups are 1.29, 1.48, and 1.87 for g = 0.2, 0.5 and 1.0 separately, equivalent to additional speedups of 2.3%, 11.2% and 68.5%. It is clear that in all cases when no improvement can be achieved by using more tiles, configuration parallelism can always help.

5.1.3.3 Evaluation of configuration parallelism using real applications

We have also carried out evaluation with real applications. The same four applications as used in section 4.5.2 were used. To make a more complicated case, we compiled the four applications into a large DAG. This was done by simply instantiating each application in the large DAG, so the final DAG was actually a composition of four individual DAGs, one for each application. Each tile in the target DRHW was assumed to contain the same amount of resources as in the Xilinx XC2V80 FPGA. Under this assumption, there were 37 configuration nodes and 28 task nodes in the DAG, and the ratio g was 0.4.

Different devices were evaluated by setting the number of tiles, N_{tile}, from 2 to 9, and the number of configuration controllers, N_{ctrl}, from 1 to 9. Because using more controllers than tiles does not bring benefit, our tool can automatically ingore the setting where $N_{tile} < N_{ctrl}$. We use the device (2,1) as the reference device because this setting is the minimum requirement to map these applications. The scheduling results are extracted and shown in Figure 34. There are eight sections in the graph with each section representing one setting of N_{tile} and each index within one section representing one setting of N_{ctrl}. In line with the evaluation results derived using the randomly generated task graphs as presented in section 5.1.3.2, the results show that more speedups could be achieved by using more tiles. It is clear from the results that using more controlles can bring additional improvements. For example, the speedup at (8,1) is 2.6, but the value is 3.2 at (8,2). This is because with one additional configuration

controller the configuration overhead drops to 11% from 36%. Although more performance improvements can be achieved with additional controllers, we consider that using more than 2 controllers is not practical from the implementation point of view. However, reasonably good improvement can already be achieved with 2 controllers. For example, when $N_{tile} > 4$, using 2 controllers is always better than using one additional tile.

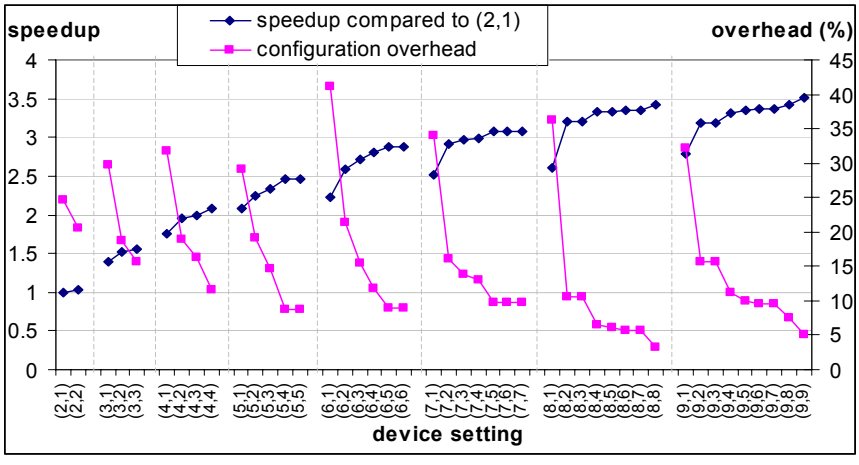


Figure 34. Evaluation of configuration parallelism with real applications.

5.2 Using dynamic voltage scaling to reduce the configuration energy

The second technique is to apply dynamic voltage scaling technique (DVS) on the reconfiguration process. The basic idea is to use configuration prefetching and parallelism to create excessive system idle time and apply DVS on the configuration process when such idle time can be utilized. Therefore, for configuration processes on which lower supply voltage is applied, lower configuration energy is required.

5.2.1 Motivation

The dynamic power consumption of a circuit, P_{dyn}, satisfies the relation that $P_{dyn} \propto CV^2f$, where C is the capacitance of the circuit, V is the supply voltage and f is the operation frequency. Because the supply voltage has a quadratic effect on the

dynamic power consumption, reducing the supply voltage is the most effective approach to lower P_{dyn}, but low supply voltage will increase the configuration latency and degrade the performance.

However, by using configuration prefetching and parallelism, we can create excessive system idle time and thus benefit from using the DVS. Simple examples are shown in Figure 35. Figure 35(a) shows the case where the idle time is created by prefetching. Such idle time can then be utilized to lower the supply voltage of the configuration process, as shown in Figure 35(c).Figure 35(b) shows the case that Task 2 needs two configurations. If they can be performed in parallel, the idle time marked in Figure 35(b) can then be utilized to apply DVS, as in Figure 35(d).

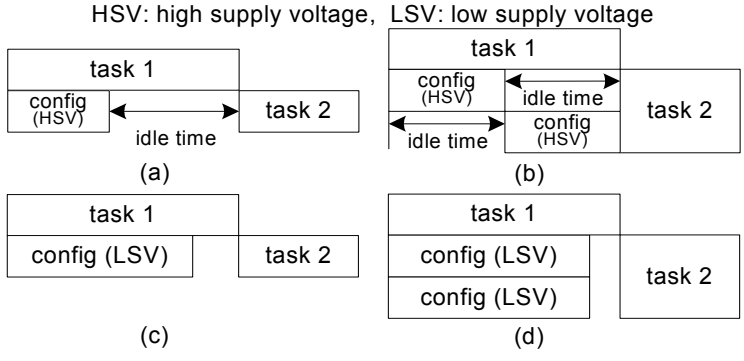


Figure 35. Using configuration prefetching and configuration parallelism to create excessive idle time.

5.2.2 Device model and evaluation technique

The device model is based on the parallel configuration model described in Section 5.1. Because each tile has its own configuration-SRAM, this allows us to apply DVS on the configuration-SRAM and the corresponding configuration controller for each individual configuration process. However, applying low supply voltage on the configuration-SRAM will degrade the circuit performance. Therefore, buffers are needed at the output of the configuration-SRAM to boost the output voltage level to the same level as used in the circuit. These buffers do not cause delays at run-time, because the configuration-SRAM supply DC signals to the circuit. In this phase of the work, our main objective is to reduce the configuration energy, therefore we do not consider applying DVS on the circuit, as in [170].

The goal of using DVS is to reduce configuration energy. However, when lower supply voltage is applied, configuration latency is increased, which might increase the overall scheduling and decrease the system performance. Therefore, we need an approach that can optimally assign the DVS states in such a way that the lowest configuration energy is achieved without increasing the overall schedule length. This requires a task scheduler that tries to reduce both schedule length and configuration energy while considering task allocation, configuration prefetching, configuration parallelism and DVS state assignments at the same time. To solve this multi-objective NP-hard optimization problem, we extend the GA-based scheduler, which has been described in Section 4.3.3.

To represent the DVS state, the chromosome of the GA-based scheduler is extended to include a string of paired tokens to represent the DVS states of configurations, one pair for one configuration process. The first token of a pair denotes the configuration, and the second denotes the DVS state. For example, a complete chromosome including the DVS tokens is shown in . Correspondingly, modifications of GA operators are needed. When generating the initial population, the second token value (supply voltage) is randomly selected from a predefined set of possible supply voltage states. During crossover, the configuration nodes and the task nodes are swapped as described in Section 4.3.3.4. DVS states of the configuration nodes of $child_1$ in the left graph remain the same as that in par_1. The rest will use the DVS states as in par_2. During the mutation phase, an additional mutation scheme is added. It randomly selects a configuration process and then changes its DVS state into another randomly selected DVS state.

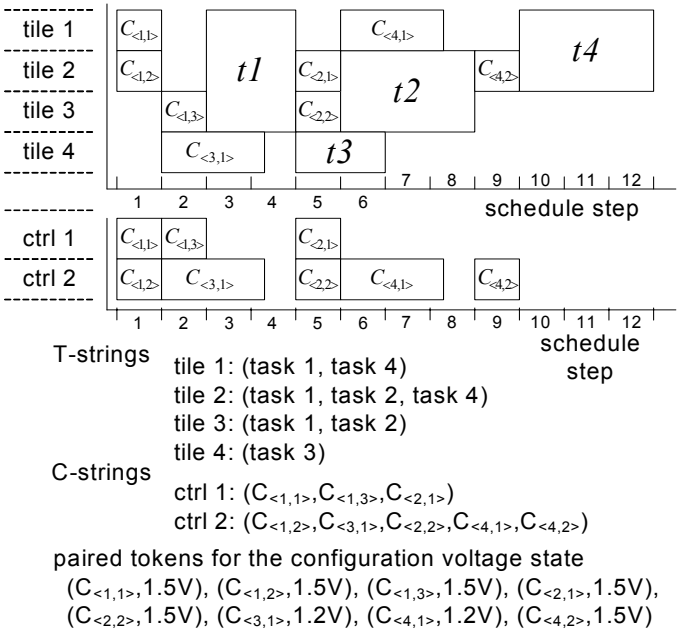


Figure 36. A chromosome in DVS-enabled task scheduler

Considering the configuration energy, the fitness value is calculated as:

$$fitness = \frac{reference\_length}{current\_length} + a * \frac{reference\_energy}{current\_energy} \tag{10}$$

where a = 0, if *current_length* > *reference_length*
a = 1, if *current_length* <= *reference_length*

The reference_length and the reference_energy are derived from non-DVS scheduling. In fact, we run the modified GA-based task scheduler twice for solving the multi-objective optimization problem. In the first run, we do not consider DVS (all configuration processes are assigned to the highest supply voltage) and try to find the shortest schedule length. The evaluator and termination criteria as described in Section 4.3.3.4 are used. In the second run, we take DVS state assignment into account and use the shortest schedule length, derived from the first run, as the reference_length. The reference_energy is the sum of configuration energy in the case that all configuration processes use the highest supply voltage.

5.2.3 Case studies

5.2.3.1 Evaluation with pseudo tasks

We used 10 randomly generated task graphs with each graph containing 10 tasks. These graphs had different levels of depth and different tree structures, so they could be seen as representations of widely different applications. The number of required tiles of an individual task was randomly generated with uniform distribution in the range of [1, 3]. Different device models were used by setting the number of tiles, N_{tile}, to iterate from 4 to 7 and the number of controllers, N_{ctrl}, to iterate from 1 to 3. In the following context, we use (N_{tile}, N_{ctrl}) to refer to the device with N_{tile} tiles and N_{ctrl} controllers. The ratio of the average configuration time to the average computation time, g, was set to be 0.2, 0.5, and 1.0 separately. Four supply voltages were used. The power-delay profile of the configuration process is shown in Table 8. The 1.5 V profile was estimated based on the XC2V80 FPGA datasheet [27], and others were derived from the power-voltage relation ($P_{dyn} \propto CV^2f$). The following GA parameters were used.

Table 8. Power-delay profile of the configuration process.

Supply voltage	Delay	Power
1.2 V	374 us	192 mw
1.3 V	346 us	225 mw
1.4 V	323 us	261 mw
1.5 V	304 us	300 mw

- mutation probability: 0.15
- crossover probability: 0.95
- replacement percentage in one generation: 80%
- number of individuals in one generation: 60.

In order to use DVS to minimize the configuration energy but without increasing the schedule length when compared to no-DVS scheduling, we set that the GA termination criteria should satisfy the following two conditions: 1) The average schedule length in the current generation is equal to the no-DVS schedule length,

which can be derived by using only the highest supply voltage state in the scheduling process; and 2) The difference between the average configuration energy and the lowest configuration energy in the current generation is within 0.1% for 5 continuous generations. We stopped the no-DVS scheduling after 1000 generations. The average run-time was 6.5 seconds. For the scheduling including DVS, the average run-time was 25 seconds under the above termination criteria. The best result out of 10 runs is used in the following analysis.

The reduced configuration energy is extracted and averaged over the 10 DAGs. The results are presented in Figure 36. When considering individual cases, the maximal reduction of the configuration energy is 20.2%. When we average the results for each setting of g, the average reduction of the configuration energy are 15.7%, 12.5%, and 6.9% separately for $g = 0.2$, 0.5, and 1.0. It can be seen that for a smaller configuration latency ($g = 0.2$), using a single configuration controller (N_{tile},1) can already significantly reduce the configuration energy. This is because for smaller g using only prefetching has already created enough idle time that can be utilized to apply DVS on the configuration process, as shown in Figure 35(a, c). For larger configuration latency, it can be seen that excessive idle time is created only when multiple controllers are applied, as shown in Figure 35(b, d). The results of $g = 0.5$ on (5,N_{ctrl}) show that using 3 controllers tends to be less effective than using 2 controllers. This is because the additional controller is busy at configuring tasks (reducing the total schedule length is also one of our objectives). Therefore, less excessive idle time is available.

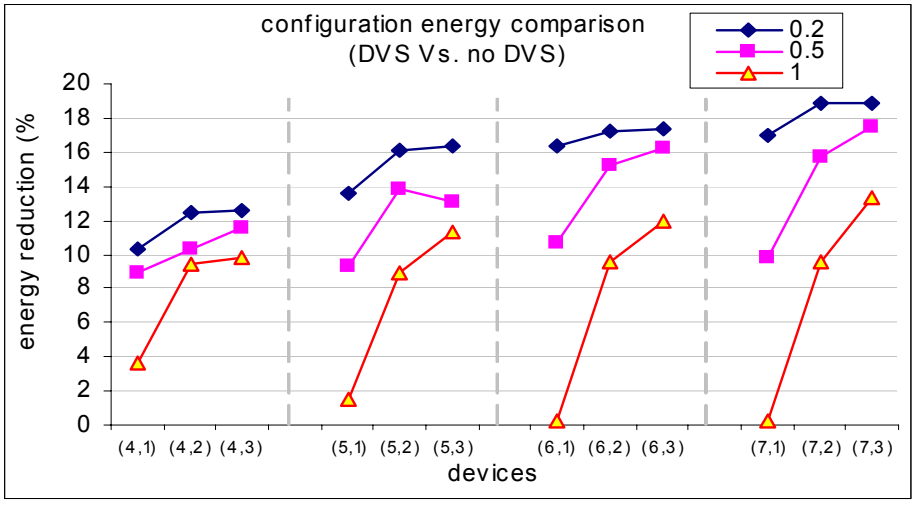


Figure 36. Comparison of energy reduction of using DVS and without using DVS.

In Figure 37, we depict the voltage distribution on (7,N_{ctrl}) to present more details of the results. For small configuration latency, it can be seen that majority of the configuration processes are assigned to the lowest supply voltage for a single controller case. In addition, using configuration parallelism barely changes the voltage distribution. In contrast, for a large configuration latency, using additional controllers allows more high-voltage states to be replaced with low-voltage states.

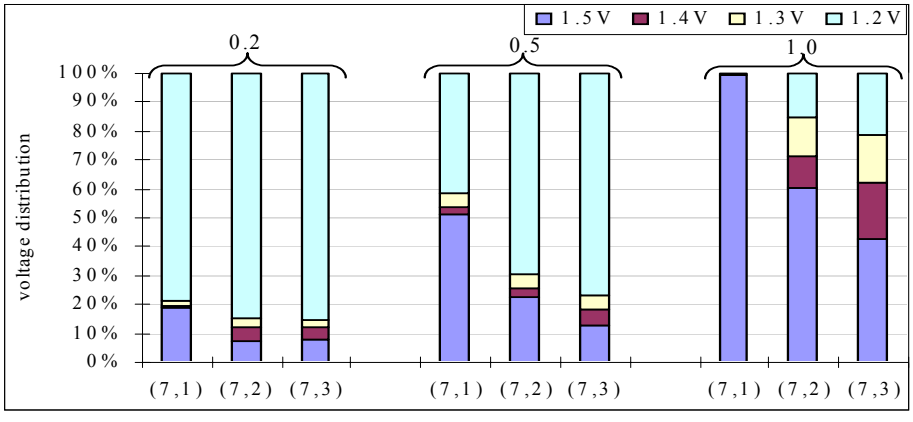


Figure 37. Voltage assignment distribution.

5.2.3.2 Evaluation with real applications

We also tested the approach with seven real applications, sobel (image sharpening using sobel masking), unsharp (image sharpening with blur), laplacian (image sharpening using laplacian filter), sobel & noise (image sharpening with noise reduction), JPEG decoder, MPEG decoder and WCDMA detector (four core functions for channel equalization). Each application was divided into a number of tasks, and each task was manually coded in VHDL. The resources and the execution time were derived from synthesis results and simulation results. We evaluated on devices that contained from 4 tiles to 7 tiles with one configuration controller. We assumed that each tile consisted of the same amount of resources and had the same configuration overhead as in the XC2V80 FPGA. This gave us that the ratio g was in the range of [0.18, 0.27] for these applications. The same GA settings as in the previous case were used. On average, each GA run took 8.7 s. The results of configuration energy reduction are depicted in Figure 38. On average, configuration energy could be reduced by 15.4% without increasing the schedule length. In the best case, sobel & noise on device (7,1), 19.3% was theoretically achievable.

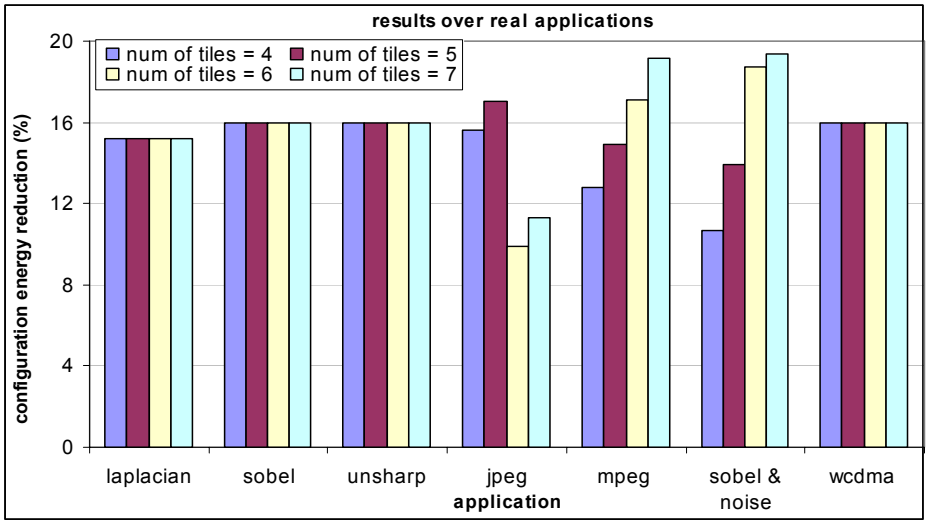


Figure 38. Configuration energy reduction of real applications.

5.3 Discussion

In this section, we have presented two techniques to reduce the impact of the configuration overhead. Although different techniques have been proposed, as summarized in section 2.3, these two techniques are novel because they tackle the problem from different aspects. This also makes it infeasible to directly compare our techniques from previous ones. To validate our techniques, we compare the difference between applying them and not applying them on the same systems.

The first technique is referred to as configuration parallelism, which is supported by a novel configuration model. The model consists of multiple homogeneous tiles and each tile has its own configuration-SRAM that can be individually accessed. The configuration-SRAMs are connected to multiple individual configuration controllers by a crossbar, and tasks can be loaded in parallel. This allows more task parallelism to be exploited. The configuration parallelism is an approach orthogonal to the configuration prefetching and both can reduce the configuration overhead, so they should be applied together if possible. Results of using the randomly generated task graphs as well as using real applications show that system performance can be improved using more tiles (computation resources), but without configuration parallelism such benefit is limited.

The second technique is to apply DVS on the configuration process to reduce the configuration energy. The idea is to use configuration prefetching and configuration parallelism to create system idle time and then apply DVS on configuration processes when such idle time can be utilized. The GA task scheduling approach, as presented in section 4.3.3, has been extended to solve the multi-objective optimization problem, e.g., task allocation, scheduling, configuration prefetching, and DVS state assignment. A set of randomly generated tasks is used in evaluation. Considering the reduction of configuration energy, the results show that using more tiles is more beneficial when the configuration latency is relatively small and using more controllers is more beneficial when the latency is relatively large. Evaluation with real applications shows that up to a 19.3% reduction of configuration energy is achievable.

6. Conclusions

The increasing complexity of computation-intensive applications is driving the development of high performance computation engines. In most cases, ASIC is the main solution, because it has the ability to customize the design down to the silicon level in order to achieve optimal performance. However, performance is usually not the only goal of a design. Many systems are also in favor of flexibility in order to enable post-fabrication upgrading of functionality and easier bug-fixing ability. In addition, although the development of semiconductor technology has provided us enough transistors in a single chip, it is inefficient to build a system that contains lots of components that are not used simultaneously at run-time. One possible solution is to build such systems based on processors, which have been pre-verified and have lower design costs. Applications implemented on processors can be easily modified. However, in many cases processor technology fails to deliver the required performance because it sacrifices too much performance for flexibility.

One alterative is dynamically reconfigurable hardware (DRHW), which at run-time enables us to modify the functions that are mapped onto it. This feature is similar to software multi-tasking on processors. However, the design of DRHW uses a similar flow as in ASIC design in the sense that application customization, such as parallelism and pipeline, can be applied. Therefore, although a design is finally implemented on pre-fabricated components, such as LUT, DRHW can still provide significantly higher performance when compared to software implementation. However, a main drawback of DRHW is the configuration overhead related to each reconfiguration process. It can largely degrade the system performance. In addition, design supports at the system level for systems including DRHW are missing.

6.1 Summary of contributions

In this thesis work, we have presented several approaches from different aspects to tackle the design problems of DRHW, especially at the system level. They are summarized as follows.

- We have presented system-level design supports for reconfigurable system-on-chip in which DRHW is frequently used as a coprocessor to accelerate computation-intensive tasks. Our approach can help designers to easily evaluate

the effect of moving some tasks, which are traditionally implemented in fixed hardware, to DRHW. The supports that we provide are an estimation approach and a SystemC modeling technique for DRHW. The estimation approach starts from function blocks represented in ANSI-C language, and it produces hardware execution time and resource utilization estimates for each function block by applying a set of high-level synthesis algorithms. In DRHW modeling, behavior of the reconfiguration process is modeled instead of the real reconfiguration process. In addition, a number of related parameters are specified, which can be tuned to target a particular configuration technology. To reduce the coding effort, a tool to automatically generate DRHW SystemC models is created. The system-level design approach has been applied on a WCDMA case study. The estimation technique has been used at the system level to support us in partitioning functions into two contexts, which in the implementation phase are mapped onto the same region of a commercial FPGA that supports partial reconfiguration. The effect of run-time reconfiguration has also been evaluated from simulation using the modeling technique. When implementing the design on the demonstration environment, the results showed that more than 40% of resource reduction in terms of LUT can be achieved over a completely fixed implementation and 30 times speedup can be achieved over software implementation.

- We have also presented several static scheduling techniques to optimally or near-optimally schedule tasks onto DRHW. Three static scheduling techniques embedding configuration prefetching have been developed and quantitatively evaluated. Different problem solving strategies are used. The first is a list-based heuristic approach; the second is an optimal approach based on constraint programming (CP); the last is a guided random search technique developed using a genetic algorithm (GA). Randomly generated task graphs and practical applications are used in the case studies. The list-based approach is the most efficient but the least optimal approach. On the other hand, the CP-based approach can be guaranteed to generate optimal solutions. However, the required computer run-time is much longer and very unpredictable. For larger cases, optimal solutions cannot be found within days. In comparison, the GA-based approach shows high accuracy and reasonable efficiency. In addition, the GA-based approach has shown good convergence and almost linear scalability in terms of the number of generations required to converge.
- We have presented a run-time scheduling approach with a novel configuration locking technique. The basic idea is to monitor at run-time the execution times of

tasks and always lock a number of the most frequently used tasks on DRHW. A number of real applications are used to validate the approach. The results show that when resources are not too limited or too prosperous, preserving the proper amount of tiles to lock the most frequently used tasks can significantly improve the system performance.

- To reduce the impact of configuration latency, we have presented a technique, configuration parallelism. It is supported by a novel configuration model. The model consists of multiple homogeneous tiles. Each tile has its own configuration-SRAM that can be individually accessed, and the configuration-SRAMs are connected to multiple individual configuration controllers by a crossbar. Therefore, different configuration-SRAMs can be accessed simultaneously and thus tasks can be loaded in parallel. Results of using the randomly generated task graphs as well as using real applications show that system performance can be improved using more tiles (computation resources), but without configuration parallelism such benefit is limited.
- We have also presented a technique to reduce the configuration energy. The idea is to use configuration prefetching and configuration parallelism to create system idle time and then apply DVS on configuration processes when such idle time can be utilized. The technique is evaluated using a GA-based task scheduler, the goal of which is to first find minimal schedule length and then achieve minimal configuration energy. A set of randomly generated tasks is used in evaluation. Considering the reduction of configuration energy, the results show that using more tiles is more beneficial when the configuration latency is relatively small and using more controllers is more beneficial when the latency is relatively large. Evaluation with real applications shows that up to 19.3% reduction of configuration energy is achievable.

6.2 Future work

There are different ways to extend the research work that has been presented in this thesis. For the SystemC-based design supports, it is necessary to improve it to form a complete design approach for embedded systems. The current supports cannot automatically generate synthesizable code for DRHW implementation. High-level SystemC synthesis or C-based synthesis need to be studied. Our HW estimator can also benefit from the study to be more accurate. In addition, the system performance

in terms of power consumption is not addressed in the current approach. Methods and tools for power analysis will be of great interest in the near future.

The different static task scheduling techniques and the run-time scheduling technique involving configuration locking are evaluated in a simulation environment. In the future, a RTOS will be developed to implement these techniques in a practical manner. The RTOS will be instantiated in a real demonstration environment, which can be either a real physical hardware system or a virtual hardware platform. Considering the static task scheduling techniques, we foresee that combining the list-based heuristic approach and the GA-based approach will result in a more efficient task scheduling technique. For the configuration locking technique, we have presented that the number of tiles used to lock tasks has a significant impact on the performance. It is necessary to study how to decide such number based on the run-time system status. At the moment, the configuration parallelism technique and the DVS technique are analyzed theoretically. The cost of additional hardware for implementing such techniques is not considered. In the future, implementation cost will be taken into account to thoroughly evaluate these techniques. The static power consumption is not taken into account in the DVS approach. It will be included and system-level power reduction techniques with applying DVS on the circuit itself will be studied in the next step.

References

1. Moore, G.E. Progress in digital integrated electronics. IEEE International Electron Devices Meeting Technical Digest. 1975. Pp. 11–13.

2. ITRS. The international technology roadmap for semiconductors (ITRS) 2005 edition. http://www.itrs.net/Links/2005ITRS/Home2005.htm. (May, 2007).

3. Intel Corporation. Dual-Core Intel® Xeon® Processor 7100 Series Product Brief. http://download.intel.com/products/processor/xeon/7100_prodbrief.pdf. (May, 2007).

4. Hennessy, J.L., and Patterson, D.A. Computer architecture: a quantitative approach. Second edition. Morgan Kaufmann. 1995. 760 p.

5. Sylvester, D. and Keutzer, K. Getting to the bottom of deep submicron. Proceedings of the IEEE/ACM International Conference on Computer–aided design (ICCAD). 1998. Pp. 203–211.

6. Sylvester, D. and Keutzer, K. Getting to the bottom of deep submicron II: a global wiring paradigm. Proceedings of the 1999 International Symposium on Physical Design. 1999. Pp. 193–200.

7. Kalva, H. The H.264 video coding standard. IEEE Multimedia, 2006. Vol. 13, No. 4, pp. 86–90.

8. Altera Corporation. White paper: FPGAs provide reconfigurable DSP solutions. http://www.altera.com/literature/lit-wp.jsp. (May, 2007).

9. Fry, T.W. and Hauck, S. SPIHT image compression on FPGAs. IEEE Transactions on Circuits and Systems for Video Technology, 2005. Vol. 15, No. 9, pp. 1138–1147.

10. Skliarova, I. and Ferrari, A.B. Reconfigurable hardware SAT solvers: a survey of systems. IEEE Transactions on Computers, 2004. Vol. 53, No. 11, pp. 1449–1461.

11. Goodman, J. and Chandrakasan, A.P. An energy-efficient reconfigurable public-key cryptograph processor. IEEE Journal of Solid-State Circuits, 2001. Vol. 36, No. 11, pp. 1808–1820.

12. Brown, S. and Rose, J. FPGA and CPLD architectures: a tutorial. IEEE Design & Test of Computers, 1996. Vol. 13, No. 2, pp. 42–57.

13. Black, W. and Das, B. Programmable logic using giant-magneto-resistance and spin-dependent tunnelling devices. Journal of Applied Physics, 2000. Vol. 87, pp. 6674–6679.

14. Bruchon, N., Torres, L., Sassatelli, G. and Cambon, G. New non-volatile FPGA concept using magnetic tunnelling junction. Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI). 2006. Pp. 269–276.

15. Masui, S., Ninomiya, T., Ouya, M., Yokozeki, W., Mukaida, K. and Kawashima, S. A ferroelectric memory-based secure dynamically programmable gate array. IEEE Journal of Solid-State Circuits, 2003. Vol. 38, No. 5, pp. 715–725.

16. Borgatti, M., Calì, L., De Sandre, G., Forêt, B., Iezzi, D., Lertora, F., Muzzi, G., Pasotti, M., Poles, M. and Rolandi, P.L. A reconfigurable signal processing IC with embedded FPGA and multi-port Flash Memory. Proceedings of the ACM/IEEE Design Automation Conference (DAC). 2003. Pp. 691–695.

17. Borgatti, M., Calì, L., De Sandre, G., Forêt, B., Iezzi, D., Lertora, F., Muzzi, G., Pasotti, M., Poles, M. and Rolandi, P.L. A 1GOPS reconfigurable signal processing IC with embedded FPGA and 3-port 1.2GB/s Flash memory subsystem. Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC). 2003.

18. Lattice Semiconductor Corporation. LatticeXP2 low-cost non-volatile FPGA family handbook. http://www.latticesemi.com/products/fpga/xp2/. (May, 2007).

19. Compton, K. and Hauck, S. Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys, 2002. Vol. 34, No. 2, pp. 171–210.

20. Strunk, E.A. and Knight, J.C. Dependability through assured reconfiguration in embedded system software. IEEE Transactions on Dependable and Secure Computing, 2006. Vol. 3, No. 3, pp. 172–187.

21. Zhang, Y., Murata, M., Takagi, H. and Ji, Y. Traffic-based reconfiguration for logical topologies in large-scale WDM optical networks. Journal of Lightwave Technology, 2005. Vol. 23, No. 10, pp. 2854–2867.

22. Lysecky, R., Stitt, G. and Vahid, F. Warp processors. ACM Transactions on Design Automation of Electronics Systems (TODAES), 2006. Vol. 11, No. 3, pp. 659–681.

23. Lysecky, R. Low-Power warp processor for power efficient high-performance embedded systems. IEEE Proceedings of the Design, Automation, and Test in Europe Conference (DATE). 2007. Pp. 141–146.

24. McKay, N., Melham, T. and Susanto, K.W. Dynamic specialization of XC6200 FPGAs by partial evaluation. Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM). 1998. Pp. 308–309.

25. Laffely, A., Liang, J., Jain, P., Burleson, W. and Tessier, R. Adaptive systems on a chip (aSoC) for low-power signal processing. Conference Record of the 35th Asilomar Conference on Signals, Systems and Computers, Vol. 2. 2001. Pp. 1217–1221.

26. Barco. MPEG-4 simple profile decoder: BA132MPEG4D factsheet. http://www.barco.com/subcontracting/Downloads/IPProducts/BA132MPEG4D Factsheet.pdf. (May 2007).

27. Xilinx Incorporated. Virtex-II complete data sheet (all four modules). http://direct.xilinx.com/bvdocs/publications/ds031.pdf. (May, 2007).

28. Qu, Y., Tiensyrjä, K., Soininen, J.-P. and Nurmi, J. SystemC-based Design Methodology for Reconfigurable System-on-Chip. IEEE Proceedings of the 8th Euromicro Symposium of Digital System Design (DSD). 2005. Pp. 364–371.

29. Qu, Y., Tiensyrjä, K., Soininen, J.-P. and Nurmi, J. System-Level Design for Partially Reconfigurable Hardware. IEEE International Symposium on Circuits and Systems (ISCAS). 2007. Pp. 2738–2741.

30. Qu, Y. and Soininen, J.-P. Estimating the utilization of embedded FPGA co-processor. IEEE Proceedings of the 6th Euromicro Symposium of Digital System Design (DSD). 2003. Pp. 214 – 221.

31. Qu, Y., Tiensyrjä, K. and Masselos, K. System-level modeling of dynamically reconfigurable co-processors. Proceedings of International Conference on Field Programmable Logic and Applications (FPL). 2004. Pp. 881–885.

32. Qu, Y., Soininen, J.-P. and Nurmi, J. Static Scheduling Techniques for Dependent Tasks onto Dynamically Reconfigurable Devices. Journal of Systems Architecture. 2007. Vol. 53, No. 11, pp. 861–876.

33. Qu, Y., Soininen, J.-P. and Nurmi, J. A parallel configuration model for reducing the run-time reconfiguration overhead. IEEE Proceedings of the Design, Automation, and Test in Europe Conference (DATE). 2006. Pp. 965–970.

34. Qu, Y., Soininen, J.-P. and Nurmi, J. Using constraint programming to achieve optimal prefetch scheduling for dependent tasks on run-time reconfigurable devices. IEEE International Symposium on System-on-Chip. 2006. Pp. 83–86.

35. Qu, Y., Soininen, J.-P. and Nurmi, J. A genetic algorithm for scheduling tasks onto dynamically reconfigurable hardware. IEEE International Symposium on Circuits and Systems (ISCAS). 2007. Pp. 161–164.

36. Qu, Y., Soininen, J.-P. and Nurmi, J. Improving the Efficiency of Run Time Reconfigurable Devices by Configuration Locking. IEEE Proceedings of the Design, Automation, and Test in Europe Conference (DATE). 2008. (In press).

37. Qu, Y., Soininen, J.-P. and Nurmi, J. Using multiple configuration controllers to reduce the configuration overheads. IEEE Proceedings of the 23rd IEEE Norchip Conference. 2005. Pp. 86–89.

38. Qu, Y., Soininen, J.-P. and Nurmi, J. Using dynamic voltage scaling to reduce the configuration energy of run time reconfigurable devices. IEEE Proceedings of the Design, Automation, and Test in Europe Conference (DATE). 2007. Pp. 147–152.

39. Bondalapati, K. and Prasanna, V. Reconfigurable computing systems. Proceedings of the IEEE, 2002. Vol. 90, No. 7, pp. 1201–1217.

40. Hartenstein, R. A decade of reconfigurable computing: a visionary retrospective. IEEE Proceedings of the Design, Automation, and Test in Europe Conference (DATE). 2001. Pp. 642–649.

41. Tessier, R. and Burleson, W. Reconfigurable computing for digital signal processing: A survey. Journal of VLSI Signal Processing, 2001. Vol. 28, No. 1, pp. 7–27.

42. Shoa, A. and Shirani, S. Run-time reconfigurable systems for digital signal processing applications: a survey. Journal of VLSI Signal Processing System, 2005. Vol. 39, No. 3, pp. 213–235.

43. Alsolaim, A., Becker, J., Glesner, M. and Starzyk, J., Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2000. Pp. 205–214.

44. Becker, J., Pionteck, T., Habermann, C. and Glesner, M. Design and implementation of a coarse-grained dynamically reconfigurable hardware architecture. Proceedings of the IEEE Computer Society Workshop on VLSI. 2001. Pp. 41–46.

45. Rath, K., Tangirala, S., Friel, P., Balsara, P., Flores, J. and Wadley, J. Reconfigurable array media processor (RAMP). Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2000. Pp. 287–288.

46. Fujii, T., Furuta, K., Motomura, M., Nomura, M., Mizuno, M., Anjo, K., Wakabayashi, Hirota, K.Y., Nakazawa, Y., Ito, H. and Yamashina, M. A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture. Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC). 1999. Pp. 364–365.

47. Shibata, Y., Uno, M., Amano, H., Furuta, K., Fujii, T. and Motomura, M. A virtual hardware system on a dynamically reconfigurable logic devices. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2000. Pp. 295–296.

48. Bobda, C., Majer, M., Ahmadinia, A., Haller, T., Linarth, A. and Teich, J. Increasing the flexibility in FPGA-based reconfigurable platforms: the Erlangen Slot Machine. Proceedings of the IEEE Conference on Field-Programmable Technology (FPT). 2005. Pp. 37–42.

49. Bobda, C., Majer, M., Ahmadinia, A., Haller, T., Linarth, A., Teich, J., Fekete, S.P. and van der Veen, J. The Erlangen Slot Machine: a highly flexible FPGA-based reconfigurable platform. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2005. Pp. 319–320.

50. Majer, M., Teich, J., Ahmadinia, A. and Bobda, C. The Erlangen Slot Machine: a dynamically reconfigurable FPGA-based computer. The Journal of VLSI Signal Processing, 2007. Vol. 47, No. 1, pp. 15–31.

51. Schmit, H. Incremental reconfiguration for pipelined applications. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1997. Pp. 47–55.

52. Luk, W., Shirazi, N., Guo, S.R. and Cheung, P.Y.K. Pipeline morphing and virtual pipelines. Proceedings of International Conference on Field Programmable Logic and Applications (FPL). 1997. Pp. 111–120.

53. Cadambi, S., Weener, J., Goldstein, S.C., Schmit, H. and Thomas, D.E. Managing pipeline-reconfigurable FPGAs. Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA). 1998. Pp. 55–64.

54. Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M. and Taylor, R.R. PipeRench: a reconfigurable architecture and compiler. Computer, 2000. Vol. 33. No. 4, pp. 70–77.

55. Schmit, H, Whelihan, D., Moe, M., Levine, B. and Taylor, R. PipeRench: a virtualized programmable datapath in 0.18 micron technology. Proceedings of the 24th IEEE Custom Integrated Circuits Conference (CICC). 2002. Pp. 63–66.

56. Moe, M., Schmit, H. and Goldstein, S.C. Characterization and parameterization of a pipeline reconfigurable FPGA. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1998. Pp. 294–295.

57. Lysecky, R., Vahid, F. and Tan, S. Dynamic FPGA routing for just-in-time FPGA compilation. IEEE Proceedings of the Design Automation Conference (DAC). 2003. Pp. 334–337.

58. Lysecky, R., Vahid, F. and Tan, S. A study of the scalability of on-chip routing for just-in-time FPGA compilation. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2005. Pp. 57–62.

59. Barat, F., Lauwereins, R. and Deconinck, G. Reconfigurable instruction set processors from a hardware/software perspective. IEEE Transactions on Software Engineering, 2002. Vol. 28, No. 9, pp. 847–862.

60. Hauck, S., Fry, T.W., Hosler, M.W. and Kao, J.P. The chimaera reconfigurable functional unit. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1997. Pp. 87–96.

61. Razdan, R. and Smith, M.D. A high-performance microarchitecture with hardware-programmable functional units. Proceedings of the 27th Annual International Symposium on Microarchitecture. 1994. Pp. 172–180.

62. Kastrup, B., Bink, A. and Hoogerbrugge, J. ConCISe: a compiler-driven CPLD-based instruction set accelerator. Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1999. Pp. 92–101.

63. Wittig, R. and Chow, P. OneChip: an FPGA processor with reconfigurable logic. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1996. Pp. 126–135.

64. Goldstein, S.C., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R.R. and Laufer, R. PipeRench: a coprocessor for streaming multimedia acceleration. Proceedings of the 26th Annual International Symposium on Computer Architecture. 1999. Pp. 28–39.

65. Hauser, J.R., Wawrzynek, J. Garp: a MIPS processor with a reconfigurable coprocessor. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1997. Pp. 12–21.

66. Becker, J., Thomas, A., Vorbach, M. and Baumgarte, V. An industrial/academic configurable system-on-chip project (CSoC): coarse-grain XPP-/Leon-based architecture integration. IEEE Proceedings of the Design, Automation and Test in Europe Conference (DATE). 2003. Pp. 1120–1121.

67. Lu, G., Singh, H., Lee, M., Bagherzadeh, N., Kurdahi, F.J. and Filho, E.M.C. MorphoSys: an integrated re-configurable architecture. Proceedings of the 5th International Euro-Par Conference. 1999. Pp. 727–734.

68. Vuillemin, J., Bertin, P., Roncin, D., Shand, M., Touati, H. and Boucard, P. Programmable Active Memories: Reconfigurable Systems Come of Age. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 1996. Vol. 4, No. 1, pp. 56–69.

69. Xilinx Incorporated. Virtex-5 family overview – LX, LXT and SXT platforms. http://direct.xilinx.com/bvdocs/publications/ds100.pdf. (May, 2007).

70. Altera Corporation. Stratix III device family – the lowest power high-performance FPGAs. http://www.altera.com/products/devices/stratix3/st3-index.jsp. (May, 2007).

71. PACT XPP Technologies. XPP-III processor overview (white paper). http://www.pactxpp.com/main/download/XPP-III_overview_WP.pdf. (May, 2007).

72. Singh, H., Lu, G., Lee, M., Kurdahi, F.J., Bagherzadeh, N., Filho, E.M.C. and Maestre, R. MorphoSys: case study of a reconfigurable computing system targeting multimedia applications. IEEE Proceedings of the Design Automation Conference (DAC). 2000. Pp. 573–578.

73. Singh, H., Lee, M., Lu, G., Kurdahi, F.J., Bagherzadeh, N. and Filho, E.M.C. MorphoSys: an integrated reconfigurable systems for data-parallel and computation-intensive applications. IEEE Transactions on Computers, 2000. Vol. 49, No. 5, pp. 465–481.

74. Atmel Corporation. FPSLIC (field programmable system level integrated circuits) product overview. http://www.atmel.com/products/FPSLIC/overview.asp. (May, 2007).

75. Noguera, J. and Badia, R.M. A HW/SW partitioning algorithm for dynamically reconfigurable architectures. IEEE Proceedings of the Design, Automation and Test in Europe Conference (DATE). 2001. Pp. 729–734.

76. Noguera, J. and Badia, R.M. HW/SW codesign techniques for dynamically reconfigurable architectures. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2002. Vol. 10, No. 4, pp. 399–415.

77. Harkin, J., McGinnity, T.M. and Maguire, L.P. Partitioning methodology for dynamically reconfigurable embedded systems. IEE Proceedings of Computer Digital Techniques, 2000. Vol. 47, No. 6, pp. 391–396.

78. Berthelot, F., Nouvel, F. and Houzet, D. Design methodology for runtime reconfigurable FPGA: from high level specification down to implementation. IEEE Workshop on Signal Processing Systems Design and Implementation. 2005. Pp. 497–502.

79. Handa, M., Radhakrishnan, R., Mukherjee, M. and Vemuri, R. A fast macro based compilation methodology for partially reconfigurable FPGA designs. Proceedings of the 16th International Conference on VLSI. 2003. Pp. 91–96.

80. Chatta, K.S. and Vemuri, R. Hardware-software codesign for dynamically reconfigurable architectures. Proceedings of International Conference on Field Programmable Logic and Applications (FPL). 1999. Pp. 175–184.

81. Shang, L. and Jha, N.K. Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs. IEEE Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)/VLSI Design. 2002. Pp. 345–352.

82. Shang, L., Dick, R.P. and Jha, N.K. SLOPES: hardware-software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable FPGAs. IEEE Transactions on Computer-Aided Design (CAD) of Integrated Circuits and Systems, 2007. Vol. 26, No. 3, pp. 508–526.

83. Fröhlich, D., Steinbach, B. and Beierlein, T. UML-Based Co-Design for Run-Time Reconfigurable Architectures. Proceedings of Forum on Specification & Design Languages (FDL). 2003. Pp. 285–296.

84. Schattkowsky, T., Mueller, W. and Rettberg, A. A Model-Based Approach for Executable Specifications on Reconfigurable Hardware. IEEE Proceedings of the Design, Automation and Test in Europe Conference (DATE). 2005. Pp. 692–697.

85. Turner, R., Woods, R., Sezer, S. and Henon, J. A virtual hardware handler for RTR systems. Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1999. Pp. 262–263.

86. IPFlex Incorporated. DAPDNA Specifications. http://www.ipflex.com/en/E1-products/dd2Spec.html. (May, 2007).

87. Malik, U. and Diessel, O. The entropy of FPGA reconfiguration. IEEE Proceedings of International Conference on Field Programmable Logic and Applications (FPL). 2006. Pp. 1–6.

88. Li, Z. and Hauck, S. Configuration compression for Virtex FPGAs. Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2001. Pp. 147–159.

89. Dandalis, A. and Prasanna, V.K. Configuration compression for FPGA-based embedded systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2005. Vol. 13, No. 12, pp. 1394–1398.

90. Farshadjam, F., Fathy, M. and Dehghan, M. A new approach for configuration compression in Virtex based RTR systems. Canadian Conference on Electrical and Computer Engineering, Vol. 2. 2004. Pp. 1093–1096.

91. Hauck, S. and Wilson, W.D. Runlength compression techniques for FPGA configurations. Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1999. Pp. 286–287.

92. Hauck, S., Li, Z. and Schwabe, E. Configuration compression of the Xilinx XC6200 FPGA. IEEE Transactions on Computer-Aided Design (CAD) of Integrated Circuits and Systems, 1999. Vol. 18, No. 8, pp. 1107–1113.

93. Luk, W., Shirazi, N. and Cheung, P.Y.K. Modeling and optimizing run-time reconfigurable systems. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1996. Pp. 167–176.

94. Luk, W., Shirazi, N. and Cheung, P.Y.K. Compilation tools for run-time reconfigurable designs. Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1997. Pp. 56–65.

95. Shirazi, N., Luk, W. and Cheung, P.Y.K. Automating production of run-time reconfigurable designs. Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 1998. Pp. 147–156.

96. Marshall, A., Stansfield, T., Kostarniv, I., Vuillemin, J. and Hutchings, B. A reconfigurable arithmetic array for multimedia applications. Proceedings of the International Symposium on Field Programmable Gate Arrays. 1998. Pp. 65–74.

97. Li, Z. Configuration management techniques for reconfigurable computing. PhD Thesis. Department of ECE, Northwestern University. 2002. ISBN 0-493-65106-3.

98. Li, Z., Compton, K. and Hauck, S. Configuration caching management techniques for reconfigurable computing. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). 2000. Pp. 22–36.

99. Taher, M. and El-Ghazawi, T. Exploiting processing locality through paging configurations in multitasked reconfigurable systems. IEEE Reconfigurable Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium. 2006.

100. Compton, K., Cooley, J., Knol, S. and Hauck, S. Configuration relocation and defragmentation for reconfigurable computing. Proceedings of the IEEE Symposium of Field-Programmable Custom Computing Machines (FCCM). 2000. Pp. 279–280.

101. Compton, K., Li, Z., Cooley, J., Knol, S. and Hauck, S. Configuration relocation and defragmentation for run-time reconfigurable computing. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2002. Vol. 10, No. 3, pp. 209–220.

102. Diessel, O. and Elgindy, H. Partial FPGA rearrangement by local repacking. Technical report 97-08. Department of Computer Science and Software Engineering, The University of Newcastle. 1997.

103. Diessel, O. and Elgindy, H. Run-time compaction of FPGA designs. The 7th International Workshop on Field-Programmable Logic and Applications (FPL). 1997. Pp. 131–140.

104. Brebner, G. and Diessel, O. Chip-based reconfigurable task management. The 11th International Conference on Field-Programmable Logic and Applications (FPL). 2001. Pp. 182–191.

105. van der Veen, J., Fekete, S., Majer, M., Ahmadinia, A., Bobda, C., Hannig, F. and Teich, J. Defragmenting the module layout of a partially reconfigurable device. Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA). 2005. Pp. 92–101.

106. Septien, J., Mecha, H., Mozos, D. and Tabero, J. 2D defragmentation heuristics for hardware multitasking on reconfigurable devices. IEEE Reconfigurable Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium. 2006.

107. Tabero, J., Septien, J., Mecha, H. and Mozos, D. A low fragmentation heuristics for task placement in 2D RTR HW management. Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL). 2004. Pp. 241–250.

108. A.-El Farag, A., El-Boghdadi, H.M. and Shaheen, S.I. Improving Utilization of Reconfigurable Resources Using Two-Dimensional Compaction. Proceedings of the 10th Design, Automation and Test in Europe Conference (DATE). 2007. Pp. 135–140.

109. Gericota, M.G., Alves, G.R., Silva, M.L. and Ferreira, J.M. Active replication: towards a truly SRAM-based FPGA on-line concurrent testing. Proceedings of the 8th IEEE International On-Line Testing Workshop. 2002. Pp. 165–169.

110. Gericota, M.G., Alves, G.R., Silva, M.L. and Ferreira, J.M. On-line defragmentation for run-time partially reconfigurable FPGAs. Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL). 2002. Pp. 302–311.

111. Gericota, M.G., Alves, G.R., Silva, M.L. and Ferreira, J.M. Run-time management of logic resources on reconfigurable systems. Proceedings of Design, Automation and Test in Europe Conference (DATE). 2003. Pp. 974–979.

112. Ejnioui, A. and DeMara, R.F. Area reclamation metrics for SRAM-based reconfigurable devices. Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA). 2005. Pp. 196–202.

113. Walder, H. and Platzner, M. Non-preemptive multitasking on FPGAs: task placement and footprint transform. Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA). 2002. Pp. 24–30.

114. Handa, M. and Vemuri, R. Area fragmentation in reconfigurable operating systems. Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA). 2004.

115. Hauck, S. Configuration prefetching for single context reconfigurable coprocessor. Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA). 1998. Pp. 65–74.

116. Li, Z. and Hauck, S. Configuration prefetching techniques for partial reconfigurable coprocessors with relocation and defragmentation. Proceedings of the 10th International Symposium on Field-Programmable Gate Arrays (FPGA). 2002. Pp. 187–195.

117. Resano, J., Mozos, D., Verkest, D., Vernalde, S. and Catthoor, F. Run-time minimization of reconfiguration overhead in dynamically reconfigurable systems. Proceedings of the International Conference on Field Programmable Logic and Applications (FPL). 2003. Pp. 585–594.

118. Resano, J., Verkest, D., Mozos, D., Vernalde, S. and Catthoor, F. A hybrid design-time/run-time scheduling flow to minimize the reconfiguration overhead of FPGAs. Journal of Microprocessors and Microarchitectures, 2004. Vol. 28, No. 5–6, pp. 291–301.

119. Resano, J. and Mozos, D. Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware. Proceedings of the Design Automation Conference (DAC). 2004. Pp. 119–124.

120. Resano, J., Mozos, D. and Catthoor, F. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. Proceedings of Design, Automation and Test in Europe Conference (DATE). 2005. Pp. 106–111.

121. Resano, J., Mozos, D., Verkest, D. and Catthoor, F. A reconfiguration manager for dynamically reconfigurable hardware. IEEE Design and Test of Computers, 2005. Vol. 22, No. 5, pp. 452–460.

122. Panainte, E.M., Bertels, K. and Vassiliadis, S. The PowerPC backend MOLEN compiler. Proceedings of the International Conference on Field Programmable Logic and Applications (FPL). 2003. Pp. 900–910.

123. Panainte, E.M., Bertels, K. and Vassiliadis, S. Instruction scheduling for dynamic hardware configurations. Proceedings of Design, Automation and Test in Europe Conference (DATE). 2005. Pp. 100–105.

124. Vassiliadis, S., Wong, S. and Coţofană, S. The MOLEN ρμ-coded processor. Proceedings of the International Conference on Field Programmable Logic and Applications (FPL). 2001. Pp. 275–285.

125. Vassiliadis, S., Gaydadjiev, G., Bertels, K. and Panainte, E.M. The MOLEN programming paradigm. Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation. 2003. Pp. 1–7.

126. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G. and Panainte, E.M. The MOLEN polymorphic processor. IEEE Transactions on Computers, 2004. Vol. 53, No. 11, pp. 1363–1375.

127. Panainte, E.M., Bertels, K. and Vassiliadis, S. Compiler-driven FPGA-area allocation for reconfigurable computing. Proceedings of Design, Automation and Test in Europe Conference (DATE). 2006. Pp. 369–374.

128. Maestre, R., Kurdahi, F.J., Fernandez, M., Hermida, R., Bagherzadeh, N. and Singh, H. Optimal vs. heuristic approaches to context scheduling for multi-context reconfigurable architectures. Proceedings of the International Conference on Computer Design (ICCD). 2000. Pp. 575–576.

129. Maestre, R., Kurdahi, F.J., Bagherzadeh, N., Singh, H., Hermida, R. and Fernandez, M. Kernel scheduling in reconfigurable computing. Proceedings of Design, Automation and Test in Europe Conference (DATE). 1999. Pp. 90–97.

130. Maestre, R., Kurdahi, F.J., Fernandez, M., Hermida, R., Bagherzadeh, N. and Singh, H. A framework for reconfigurable computing: task scheduling and context management. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2001. Vol. 9, No. 6, pp. 858–873.

131. Fekete, S.P., Kohler, E. and Teich, J. Optimal FPGA module placement with temporal precedence constraints. Proceedings of Design, Automation and Test in Europe Conference (DATE). 2001. Pp. 658–665.

132. Gu, Z., Yuan, M. and He, X. Optimal static scheduling on reconfigurable hardware devices using model-checking. Proceedings of the IEEE Real Time and Embedded Technology and Applications Symposium (RTAS). 2007. Pp. 32–44.

133. Eskinazi, R., Lima, M.E., Maciel, P.R.M., Valderrama, C.A., Filho, A.G.S. and Nascimento, P.S.B. A timed Petri Net approach for pre-runtime scheduling in partial and dynamic reconfigurable systems. Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS). 2005. Pp. 154a–.

134. Diessel, O., ElGindy, H., Middendorf, M., Schmeck, H. and Schmidt, B. Dynamic scheduling of tasks on partially reconfigurable FPGAs. IEE Proceedings of Computer Digital Techniques, 2000. Vol. 147, No. 3, pp. 181–188.

135. Bazargan, K., Kastner, R. and Sarrafzadeh, M. Fast template placement for reconfigurable computing systems. IEEE Design & Test of Computers, 2000. Vol. 17, No. 1, pp. 68–83.

136. Walder, H., Steiger, C. and Platzner, M. Fast online task placement on FPGAs: free space partitioning and 2D-hashing. Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS). 2003. P. 178.2.

137. Cui, J., Deng, Q., He, X. and Gu, Z. An efficient algorithm for online management of 2D area of partially reconfigurable FPGAs. Proceedings of Design, Automation and Test in Europe Conference (DATE). 2007. Pp. 129–134.

138. Steiger, C., Walder, H. and Platzner, M. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. IEEE Transactions on Computers, 2004. Vol. 53, No. 11, pp. 1393–1407.

139. Danne, K. and Platzner, M. A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. Proceedings of the International Conference on Field Programmable Logic and Applications (FPL). 2005. Pp. 568–573.

140. Danne, K. and Platzner, M. An EDF schedulability test for periodic tasks on reconfigurable hardware devices. Proceedings of the ACM Conferences on Languages, Compilers, and Tools for Embedded Systems (LCTES). 2006. Pp. 93–102.

141. Voros, N. S. and Masselos, K. System level design of reconfigurable systems-on-chip. Springer Publisher, 2005. 231 p. ISBN 0-387-26103-6.

142. Grötker, T., Liao, S., Martin, G. and Swan, S. System design with SystemC, Springer publisher, 2002. 240 p. ISBN 978-1-4020-7072-3.

143. IEEE 1800-2005. IEEE Standard for SystemVerilog – unified hardware design, specification, and verification language. IEEE, 2005. 648 p.

144. Gajski, D.D., Zhu, J., Dömer, R., Gerstlauer, A. and Zhao, S. SpecC: specification language and methodology. Springer publisher, 2000. 336 p. ISBN 978-0-7923-7822-8.

145. Soininen, J.-P., Kreku, J., Qu, Y. and Forsell, M. Mappability estimation approach for processor architecture evaluation. Proceeding of 20th IEEE Norchip Conference. 2002. Pp. 171–176.

146. Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J., Tjiang, S., Liao, S., Tseng, C., Hall, M., Lam, M. and Hennessy, J. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. Proceedings of the 7th ACM SIGPLAN symposium on Principles and practice of parallel pro programming. 1994. Pp. 37–48.

147. GNU. gcov manual. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html. (May, 2007).

148. Gajski, D.D., Dutt, N.D., Wu, A.C-H. and Lin, S.Y-L. High-level synthesis: Introduction to chip and system design. Kluwer Academic Publishers. 1992. 376 p.

149. Paulin, P.G. and Knight, J.P. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. IEEE Transactions on CAD of Integrated Circuits and Systems, 1989. Vol. 8, No. 6, pp. 661–679.

150. The Open SystemC Initiative (OSCI). The SystemC TLM 2.0 documentation. www.systemc.org. (May, 2007).

151. OCP-IP. OCP 2.2 specification. www.ocpip.org. (Feb, 2007).

152. Chiba, S. OpenC++ reference manual. opencxx.sourceforge.net. (May, 2007).

153. Lysaght, P. and Stockwood, J. A simulation tool for dynamically reconfigurable field programmable gate arrays. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 1996. Vol. 4, No. 3, pp. 381–390.

154. Heikkilä, M.J. A novel blind adaptive algorithm for channel equalization in WCDMA downlink. IEEE Symposium on the 12th personal, indoor and mobile radio communication. 2001. Pp. 41–45.

155. Memec. Virtex-II Pro demonstration board datasheet. www.memec.com. 2003.

156. Xilinx Incorporated. Virtex-II Pro/Virtex-II Pro X complete data sheet (all four modules). http://direct.xilinx.com/bvdocs/publications/ds083.pdf. (May, 2007).

157. Xilinx Incorporated. XPP290 two flows for partially reconfiguration module-based or difference-based. direct.xilinx.com/bvdocs/appnotes/xapp290.pdf. (May, 2007).

158. Qu, Y., Soininen, J.-P. and Nurmi, J. An efficient approach to hide the run-time reconfiguration from SW applications. IEEE Proceedings of the 15th Field Programmable Logic and Applications (FPL). 2005. Pp. 648–653.

159. Xilinx Incorporated. Platform studio and EDK design environment. http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm. (May, 2007).

160. Mentor Graphics. ModelSim datasheet. www.model.com. (May, 2007).

161. Xilinx Incorporated. The iMPACT GUI configuration tool manual. http://toolbox.xilinx.com/docsan/xilinx4/data/docs/pac/preface.html, (May. 2007).

162. Campoy, M., Ivars, A.P. and Busquets-Mataix, J.V. Static use of locking caches in multitask preemptive real-time systems. IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium). 2001.

163. Freuder, E.C. In Pursuit of the Holy Grail. Constraints, 1997. Vol. 2, No. 1, pp. 57–61.

164. Marriott, K. and Stuckey, P.J. Programming with constraints: an introduction. The MIT Press. 1998. 483 p.

165. Holland, J. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. The MIT Press (reprint edition), 1992. 211 p.

166. Hou, E., Ansari, N. and Ren, H. A genetic algorithm for multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 2. 1994, pp. 113–120.

167. Correa, R., Ferreira, A. and Rebreyend, P. Scheduling multiprocessor tasks with genetic algorithms. IEEE Transactions on Parallel and Distributed Systems, 1999. Vol. 10, No. 8, pp. 825–837.

168. SICS AB. SICSTUS manual. www.sics.se/sicstus. (Jan, 2007).

169. Dick, R.P., Rhodes, D.L. and Wolf, W. TGFF: task graphs for free. Proceedings of the International Workshop on HW/SW co-design. 1998. Pp. 97–101.

170. Lin, Y., Li, F. and He, L. Circuits and architectures for FPGA with configurable supply voltage. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2005. Vol. 13, No. 9, pp. 1037–1047.

Series title, number and
report code of publication

VTT Publications 659
VTT-PUBS-659

Author(s)
Qu, Yang

Title

System-level design and configuration management for run-time reconfigurable devices

Abstract

Dynamically reconfigurable hardware (DRHW) not only has high silicon reusability, but it can also deliver high performance for computation-intensive tasks. Advanced features such as run-time reconfiguration (RTR) allow multiple tasks to be mapped onto the same device either simultaneously or multiplexed in time domain. This new type of computing element also brings new challenges in the design process. Design supports at the system level are needed. In addition, the configuration latency and the configuration energy involved in each reconfiguration process can largely degrade the system performance. Approaches to efficiently manage the configuration processes are needed in order to effectively reduce its negative impacts. In this thesis, system-level supports for design of DRHW and various configuration management approaches for reducing the impact of configuration overhead are presented.

Our system-level design supports are based on the SystemC environment. An estimation technique for system partitioning and a DRHW modeling technique are developed. The main idea is to help designers in the early design phase to evaluate the benefit of moving some components from fixed hardware implementation to DRHW. The supports have been applied in a WCDMA case study. In order to efficiently apply the multi-tasking feature of DRHW, we have developed three static task scheduling techniques and a run-time scheduling technique. The static schedulers include a list-based heuristic approach, an optimal approach based on constraint programming and a guided random search approach using a genetic algorithm. They are evaluated using both random tasks and real applications. The run-time scheduling uses a novel configuration locking technique. The idea is to dynamically track the task status and lock the most frequently used tasks on DRHW in order to reduce the number of reconfigurations. In addition, we present two novel techniques to reduce the configuration overhead. The first is configuration parallelism. Its idea is to enable tasks to be loaded in parallel in order to better exploit their parallelism. The second is dynamic voltage scaling. The idea is to apply low supply voltage in reconfiguration process when possible in order to reduce the configuration energy.

ISBN
978-951-38-7053-9 (soft back ed.)
978-951-38-7054-6 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN
VTT Publications
1235-0621 (soft back ed.)
1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

Project number
6769

Date
November 2007

Language
English

Pages
133 p.

Name of project
ADRIATIC, MARTES

Commissioned by
European Commission, Tekes

Keywords
dynamically reconfigurable hardware, run-time reconfiguration, system-level design, task scheduling, configuration locking, configuration parallelism

Publisher
VTT Technical Research Centre of Finland
P.O. Box 1000, FI-02044 VTT, Finland
Phone internat. +358 20 722 4520
Fax +358 20 722 4374

Reconfigurability is becoming an important issue in System-on-Chip (SoC) design because of the increasing demands of silicon reuse, product upgrade after shipment and bug-fixing ability. Using dynamically reconfigurable hardware (DRHW), higher performance can be achieved than in a software implementation and more flexibility than in a fixed-hardware implementation. However, run-time reconfiguration results in latency and power consumption, which can largely degrade the system performance. This brings challenges to using DRHW in SoC design. In addition, new design methods and tools are needed.

In this thesis, system-level design supports and tools for DRHW are presented. The main idea is to help designers in the early design phase to evaluate the benefit of moving some components from fixed hardware implementation to DRHW without going into implementation details. To efficiently utilize DRHW, different static and run-time task scheduling approaches are developed. In addition, two novel techniques to reduce the negative impact of run-time reconfiguration have been proposed and evaluated.

Julkaisu on saatavana
VTT
PL 1000
02044 VTT
Puh. 020 722 4520
http://www.vtt.fi

Publikationen distribueras av
VTT
PB 1000
02044 VTT
Tel. 020 722 4520
http://www.vtt.fi

This publication is available from
VTT
P.O. Box 1000
FI-02044 VTT, Finland
Phone internat. + 358 20 722 4520
http://www.vtt.fi

ISBN 978-951-38-7053-9 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 978-951-38-7054-6 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)