Pavel G. Zaykov

Multithreading for Embedded Reconfigurable Multicore Systems

Multithreading for Embedded Reconfigurable Multicore Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.Ch.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 4 november 2014 om 15:00 uur

door

Pavel G. Zaykov

Master of Science in Computer Systems and Technologies
Technical University of Sofia, Bulgaria
geboren te Plovdiv, Bulgaria

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. K.G.W. Goossens

Copromotor:

Dr. G. K. Kuzmanov

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. K.G.W. Goossens	Technische Universiteit Delft, promotor
Dr. G. K. Kuzmanov	Technische Universiteit Delft, copromotor
Dr. A. M. Molnos	CEA, advisor
Prof. K.L.M. Bertels	Technische Universiteit Delft
Prof. dr. ir. H. J. Sips	Technische Universiteit Delft
Prof. dr. B. Juurlink	Technische Universität Berlin
Prof. dr.-ing. habil. M. Huebner	Ruhr-Universität Bochum
Prof. dr. ir. J. van den Berg	Technische Universiteit Delft, reservelid

Keywords: Multithreading, Reconfigurable Systems, Processor–Coprocessor, Execution Models, Hardware Acceleration

Pavel G. Zaykov
Multithreading for Embedded Reconfigurable Multicore Systems
Computer Engineering Laboratory
PhD Thesis Technische Universiteit Delft, The Netherlands

Printed in The Netherlands

Multithreading for Embedded Reconfigurable Multicore Systems

Pavel G. Zaykov

Abstract

In this dissertation, we address the problem of performance efficient multithreading execution on heterogeneous multicore embedded systems. By heterogeneous multicore embedded systems we refer to those, which have real-time requirements and consist of processor tiles with General Purpose Processor (GPP), local memory, and one or more coprocessors running on reconfigurable logic ((e)FPGA). We improve system performance by combining two common methods. The first method is to exploit the available application parallelism by means of multithreading program execution. The second method is to provide hardware acceleration for the most computationally intensive kernels. More specifically our scientific approach is as follows: we categorize the existing program execution models from the processor–coprocessor synchronization prospective and we introduce new parallel execution models. Then, we provide a high-level architectural abstraction of those execution models and programming paradigm that describes and utilizes them. Furthermore, we propose a microarchitectural support for the identified execution models. The functionality of the microarchitectural extensions is encapsulated in a new reconfigurable coprocessor, called Thread Interrupt State Controller (TISC). To improve the overall system performance, we employ the newly proposed program execution models to transfer highly time-variable and time-consuming Real-Time Operating System (RTOS) and application kernels from software, i.e., executed on the GPPs, to hardware, i.e., executed on the reconfigurable coprocessors. We refer to this reconfigurable coprocessor as Hardware Task Status Manager (HWTSM). Due to the properties of the newly introduced execution models such as parallel execution and constant response time, we preserve the predictability and composability at application level. Last but not least, we introduce a framework for distribution of slack informa-

tion (idle processor time) among processor tiles. In the proposed framework we employ one of the newly introduced parallel processor–coprocessor execution models. We refer to the new reconfigurable coprocessor as RS. We use the extra slack information obtained through our framework for Dynamic Voltage Frequency Scaling that reduces the overall energy consumption.

Based on the available experimental results with synthetic and real applications, we improve the system speedup up to 19.6 times with the help of the Thread Interrupt State Controller. Furthermore, we reduce RTOS cost with the help of the Hardware Task Status Manager, which results in additional application acceleration up to 13.3%. Last but not least, we improve the system energy consumption up to 56.7% over current state of the art with the help of inter-tile remote slack information distribution framework.

Overall, with the help of our contributions, the system performance is improved, the predictability and composability are preserved, all with reduced energy consumption.

Table of Contents

List of Tables

List of Figures

List of Acronyms and Symbols

ACET	Actual Case Execution Time
API	Application Programming Interface
BMT	Block Multithreading
CA	Communication Assistant
CCM	Custom Computing Machine
CCU	Custom Computing Unit
CDFG	Control Data Flow Graph
CG	Conjugate Gradient
CGRA	Coarse Grained Reconfigurable Array
CPU	Central Processing Unit
CRPU	Custom Reconfigurable Processing Unit
CSDF	Cycle-Static Data-Flow
DCT	Discrete Cosine Transformation
DDF	Dynamic Data-Flow
DMA	Direct Memory Access
DSP	Digital Signal Processor
DVFS	Dynamic Voltage-Frequency Scaling
DyS	Dynamic Scheduling
eFPGA	embedded Field Programmable Gate Array
FID	Function Identifier
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPP	General Purpose Processor
HA	Hardware Accelerator
HD	High Definition
HHL	High Level Language
HW	Hardware
HWTSM	Hardware Task-Status Manager
ILP	Instruction Level Parallelism
IMT	Interleaved Multithreading
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
I/O	Input/Output
KPN	Kahn Process Network
LD	Local Data

LDCop	Local Data Coprocessor
MCM	Maximum Cycle Mean
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
MRDF	Multi-rate Data-Flow
μ*architecture*	Microarchitecture
NoC	Network on Chip
OS	Operating System
PHA	Polymorphic Hardware Accelerator
PID	Process Identifier
PIPE	Plug-In Processor Element
PThread	POSIX Thread
rc	read counter
RDR	Remote Data Receive
RP	Reconfigurable Processor
RR	Round-Robin
RRH	Runtime Reconfigurable Hardware
RS	Remote Slack
RTOS	Real-Time Operating System
ρ*BMT*	Reconfigurable Block Multithreading
ρ*IMT*	Reconfigurable Interleaved Multithreading
ρ*MT*	Reconfigurable Multithreading
ρ*SMT*	Reconfigurable Simultaneous Multithreading
SI	Special Instruction
SoC	System on Chip
SMT	Simultaneous Multithreading
SRDF	Single-Rate Data-Flow
SW	Software
SWTM	Software Thread Management
StS	Static Scheduling
TDM	Time Division Multiplexing
TID	Thread Identifier
TISC	Thread Interrupt State Controller
TLP	Task Level Parallelism
VLIW	Very Long Instruction Word
VOIP	Voice Over Internet Protocol
XREG	Exchange Register
wc	write counter
WCET	Worst Case Execution Time
WCTT	Worst Case Travel Time

1 Introduction

In this dissertation, we address the problem of performance efficient multithreading execution on heterogeneous multicore embedded systems. By heterogeneous multicore embedded systems we refer to those, which have real-time requirements and consist of processor tiles with General Purpose Processor (GPP), local memory, and one or more coprocessors running on reconfigurable logic.

We organize the rest of the chapter in five sections. Section 1.1 outlines the trends in contemporary embedded systems. Section 1.2 introduces the research problems targeted by this dissertation. Section 1.3 describes the proposed solution for the identified problems. Section 1.4 lists the contributions. Section 1.5 outlines a conceptual computing system with our contributions. The introductory chapter concludes with Section 1.6, which overviews the organization of the dissertation.

1.1 Trends in Modern Embedded Systems

Many contemporary embedded systems execute an increasing number of applications that demand high performance. In what follows, we summarize the main trends in the state of the art real-time embedded systems that target high performance computing:

- *Multicore systems*: Initially, the multicore systems have been designed to accelerate the computationally intensive applications in the general purpose domain on desktop platforms. Nowadays, we can observe a

clear trend of multicore and manycore processors to be employed in commercial embedded products.

- *Reconfigurable systems*: By reconfigurable systems, we intend (embedded) FPGAs. In the recent years, reconfigurable systems are considered by many platform designers as a flexible solution in hardware acceleration for computationally intensive applications. Moreover, most of the high-end FPGA families have hard-coded processors and provide support for multiple softcores, as well. Examples of such heterogeneous reconfigurable systems are Spartan™ and Virtex™ product families by Xilinx, and Cyclone™ product families by Altera.

- *Concurrent programming paradigms*: There are different approaches to efficiently exploit the application parallelism and utilize the platform resources. One of them is to partition applications into multiple threads and execute them in parallel, also known in literature as Thread Level Parallelism (TLP). Examples of such programming paradigms for threads are POSIX Threads (PThreads), while for task examples are OpenMP, Message Passing Interface (MPI), Kahn Process Networks (KPN), data-flow, etc.

The execution of the concurrent programming paradigms on underlying heterogeneous resources can be managed statically at design time, e.g., by a compiler or static schedulers, or dynamically at run-time, e.g., by a Real-Time Operating System (RTOS). Such an RTOS is responsible for managing the available system resources, while preserving the application functionality. Throughout this dissertation, we mainly consider the dynamic (RTOS) approach, because the application domain properties of the targeted embedded devices, i.e., hand-held devices, requires frequently changes of the set of active applications or threads at run-time.

1.2 Research Problems

Many recent real-time embedded computer systems such as mobile phones and smart TVs need to run multiple applications in parallel to ensure their complete functionality. This is often achieved by multithreading. On such embedded devices, the multithreading concept has been thoroughly developed at software level with adequate hardware support. Nevertheless, the problem has been even less explored in the case of reconfigurable multicore systems,

which need multithreading execution to efficiently utilize the system resources. In this dissertation, we investigate embedded multicore reconfigurable systems in a multithreading context. Our goal is to propose software and hardware solutions for performance efficient multithreading on multicore reconfigurable systems. To achieve our goal, we identify the following sub-problems which are then addressed in this dissertation:

- *Facilitate programmability while providing high performance*: The overall system performance is usually defined by the system throughput, latency, and responsiveness. Very often, the overall system performance depends on the way programmers encode application algorithms. Therefore, it is vital to employ a proper programming paradigm, which facilitates the programmers to create performance efficient execution codes for their algorithms.

- *Preserve predictability*: The predictability is a required property of real-time embedded systems. Usually, the predictability is associated with guaranteeing the worst-case bounds of the application and RTOS execution. In this dissertation, we target the problem of preserving RTOS predictability by limiting the dependencies between RTOS execution time and application properties, i.e., the analyses of the worst-case bounds are leveraged.

1.3 Proposed Approach

To solve the above-mentioned general research questions, in this dissertation we propose the following approach:

- To improve system programmability, we propose a programming paradigm with a set of new processor–coprocessor execution models. The newly introduced programming models are verified with high performance, predictability, composability, and energy consumption criteria. Furthermore, the proposed approach is independent of the functionality of the considered application or Operating System.

- To improve the system performance, we reduce the RTOS cost by employing architectural and microarchitectural extensions for managing multithreading workloads and moving selected RTOS services on a dedicated hardware (coprocessor).

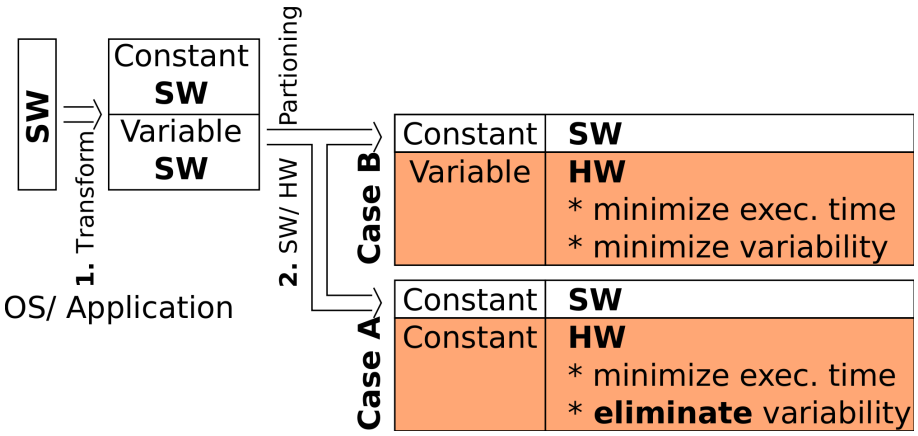


Figure 1.1: Software /hardware partitioning

- To improve the system predictability, we reduce the execution time variations of the software kernels that can be part of a user application or RTOS services.

In Figure 1.1, we summarize the general steps necessary to transfer a given software kernel to a dedicated coprocessor. Initially, the pure software version of the investigated application or RTOS service is assumed to be available. In the first step, by profiling, we transform the original application to a part with constant and a part with variable execution time software. In the second step, the most computationally intensive and time variable parts of the applied algorithms are identified for acceleration. We consider two possibilities - case A and case B, of the variable software which can be candidate for hardware acceleration. In case A, the hardware variability is minimal, while in case B, the hardware has constant execution time. By hardware, we refer to either reconfigurable or fixed, depending on the available platform resources. As a result of the introduced hardware, the remaining software kernel is expected to have shorter and possibly constant execution time compared to the original version. Based on the application properties, the introduced hardware, i.e., coprocessors, should execute the applications faster in hardware than software-only implementation. In such a way, we improve the performance and the predictability.

1.4 Dissertation Contributions

In this dissertation, our main contributions are summarized as follows:

- We propose architectural extensions that allow multithreading applications and RTOS to co-execute in software and in reconfigurable hardware (coprocessors). More specifically, we extend the processor interrupt system, the register file organization, and we propose support for hardware task synchronization at the instruction level. We encapsulate the newly introduced microarchitectural extensions in a Thread Interrupt State Controller (TISC). Furthermore, we provide a new Real-Time Interrupt Service Routine (ISR) to support the new interrupt system. We provide analytical and experimental comparison of our proposal to the state of the art proposals in terms of performance-portability and performance-flexibility characteristics.

- We generalize and classify the existing processor–coprocessor concurrent execution models with respect to the employed synchronization mechanism in the following categories: processor *only*, processor–coprocessor *sequential*, processor–coprocessor *parallel blocking*, processor–multicoprocessors *parallel blocking*.

- We introduce new execution models for the processor–coprocessor paradigm, called processor–coprocessor *parallel non-blocking* and processor–multicoprocessors *parallel non-blocking*. Unlike the processor–coprocessor *sequential* and *parallel blocking* models, in parallel *non-blocking* models, software thread is never blocked during processor–coprocessor call, which potentially gains performance and preserves predictability.

- We introduce a hierarchical programming model capable of providing flexible task migration from software to hardware, exploiting inter- and intra-thread parallelism. These types of parallelism are investigated on a real reconfigurable system working in processor–coprocessor execution models.

- We provide a comprehensive survey on the existing reconfigurable multithreading (ρMT) architectures. We propose a taxonomy that classifies these architectures in three distinctive categories with respect to their architectural support of reconfigurable multithreading. These categories are: reconfigurable architectures with *explicit* ρMT support, with

implicit ρMT support, and *no* ρMT *support*. Moreover, we list the most common design problems and we state some of the open research questions addressing performance efficient management, mapping, sharing, scheduling and execution of threads on reconfigurable hardware resources.

The proposed processor-coprocessor execution models are a general solution for various problems in the real-time embedded systems. First, we apply the execution models to improve the performance and second to guarantee composability and reduce energy consumption. Composability means that the behaviour of an application, including its timing, is independent of the presence or absence of any other application. With respect to the composability and energy consumption, our contributions are as follows:

- We propose a Hardware Task-Status Manager (HWTSM) responsible for tracking and computing the status of user tasks. The HWTSM targets data-flow real-time applications employing First-In-First-Out (FIFO) communications. The HWTSM operates in the newly introduced parallel non-blocking model.
- We propose a run-time framework for slack computation, allocation, and distribution targeting applications with tasks mapped on multiple tiles. We augment the tiles of an existing MPSoC with hardware that generates timestamps and we extend the RTOS accordingly. Since the newly introduced hardware is related with slack received from others, i.e., remote tiles, we called this hardware RS (Remote Slack). The RS operates in the newly introduced parallel non-blocking execution model.

1.5 Conceptual Computing System

In Figure 1.2, we introduce hardware and software of a conceptual computing system extended with our contributions. As an example of the conceptual computing system, we choose a Multiprocessor System-on-Chip (MPSoC). The exemplary MPSoC is composed of tiles, connected through a Network on Chip (NoC). Each tile has a processor (e.g. RISC core), instruction and data memory, and two types of coprocessors - fixed (e.g. Direct Memory Access (DMA) controller) and reconfigurable.

From a software perspective, we consider multiple user applications to be executed on the conceptual MPSoC. Furthermore, we assume that the computing

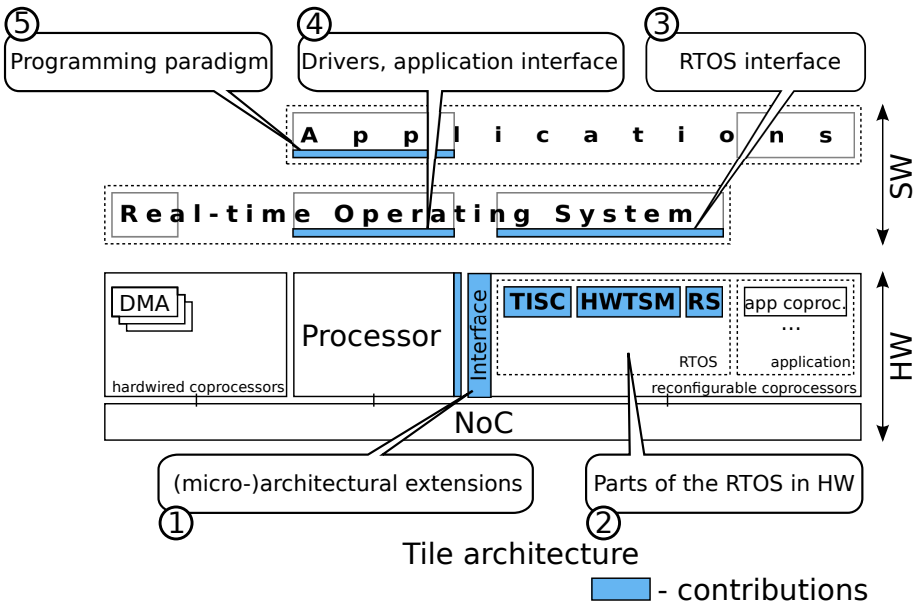


Figure 1.2: Conceptual computing system extended with our contributions

resources in the tile processor can be shared in time among multiple applications. We deliver the temporal management through an instance of a Real-Time Operating System (RTOS). The RTOS is responsible for scheduling the applications.

In Figure 1.2, we illustrate our contributions in shaded blocks. We introduce a set of (micro-)architectural extensions (see ①) to support the various processor-coprocessor execution models. As a result of our approach, reconfigurable coprocessors are shared among RTOS services and user applications. Furthermore, we introduce parts of the RTOS in hardware (see ②), i.e, TISC, HWTSM, and RS, and the corresponding RTOS interface (see ③). These RTOS reconfigurable coprocessors are accessible through RTOS drivers and application interface (see ④). At application level, we provide support for processor-coprocessor execution models in various programming paradigms (see ⑤).

1.6 Dissertation Organization

The rest of the dissertation is organized as follows: background Chapter 2 provides an overview of the reference architectures, RTOSes, and programming

models considered throughout the dissertation for proofs of concepts and prototyping.

Chapter 3 defines our general approach towards solving the targeted research problems. First, we introduce a conceptual model of the targeted class of reconfigurable system and we specify our contributions. In what follows, we describe the existing and newly introduced concurrent execution models for processor–coprocessor paradigms. We propose a new programming paradigm and we provide a comparison to data-driven programming models. We conclude the chapter by listing the proposed reconfigurable coprocessors, implemented as Molen-style CCUs operating in the newly introduced execution models.

Chapter 4 applies the general approach to accelerate parts of the RTOS such as Thread Interrupt State Controller (TISC). The TISC executes in the processor-coprocessor/multiprocessor parallel blocking execution model. We describe the proposed architecture and microarchitectural extensions in detail – hardware components and interfaces, including XREGs, polymorphic instruction implementations, controller and interrupt management.

Chapter 5 also applies the general approach to accelerate parts of the RTOS such as the Hardware Task-Status Manager (HWTSM). The HWTSM is a Molen-style CCU that accelerates part of the RTOS scheduling routines. The HWTSM executes in the processor-coprocessor parallel non-blocking model. We provide a quantitative comparison of the possible microarchitectural implementations. We describe the internal organization of the HWTSM, and we cover the relevant software integration details to the existing system.

Chapter 6 applies framework for slack information distribution among the processor tiles in a MPSoC. We employ the slack to reduce the energy consumption of the system. We achieve the goal by introducing a Molen-style CCU called Remote Slack (RS) CCU. The RS CCU operates in processor-coprocessor parallel non-blocking model.

Chapter 7 presents the experimental results for TISC, HWTSM, and RS CCUs. For the TISC, we evaluate the performance improvement and we compare our proposal with the most relevant research projects in terms of performance-portability and performance-flexibility characteristics. For the HWTSM CCU, we list the potential application performance improvement. For the RS CCU, we provide the experiments for the obtained frequency levels in each one the cores on which the targeted application is mapped. Moreover, we compare the energy consumption of our proposal with existing state of the art solutions. We also provide an estimation on the overall potential gains on a conceptual

architecture, which combines all three CCUs.

Chapter 8 introduces a taxonomy of the existing reconfigurable architectures with respect to their support of multithreading. Furthermore, we summarize several relevant design problems.

Finally, Chapter 9 summarizes the dissertation, outlines the contributions, and points to potential future directions.

2 Background

In this chapter we introduced the reference architectures, corresponding RTOSes, and accompanying terminology for the considered programming models used in this dissertation for proofs of concept and prototyping.

2.1 Reference Architectures and RTOSes

In this dissertation, we consider two types of architectures, which we augment with the Molen processor–coprocessor prototype [111]. We choose these architectures to be representatives of single and multicore platforms. As a single-core platform, we employ the MIPS R3000 RISC core [83]. As a multicore platform template, we use the CompSoC platform [37]. We conclude the section with the list of FPGA chip generations on which we implement the targeted architectures.

2.1.1 The Molen Machine Organization

In Figure 2.1, we present the Molen Polymorphic Processor organization [111]. The Molen Polymorphic Processor consists of a General Purpose Processor (GPP) and a Reconfigurable Processor (RP) operating under the processor–coprocessor architectural paradigm. In the Molen context, the implementations of application specific functionalities in reconfigurable hardware are called Custom Computing Units (CCUs), therefore we assumed the same terminology further in this dissertation. The processor has an arbiter,

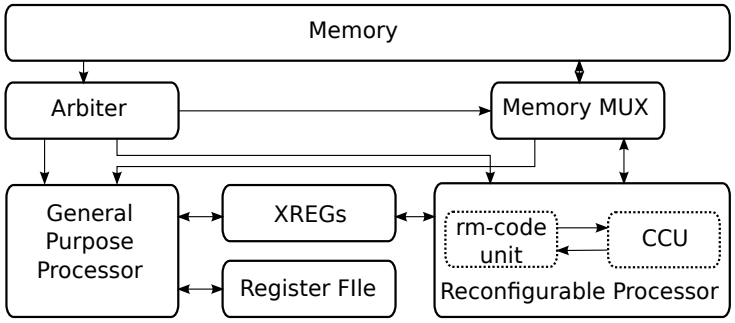


Figure 2.1: The Molen polymorphic processor

which partially decodes and issues instructions to either of the GPP or the RP. A general one-time extension of the instruction set is proposed to support an arbitrary functionality implemented in the CCU. Six of the eight additional instructions are related to the RP and two to the parameters transferred between the GPP and the RP through exchange registers (XREGs). The RP related instructions, support different variations of the set–execute paradigm, described in detail in [110]. The very basic operations of the RP are "set" and "execute". The "set" instruction configures the CCU for a particular functionality and the "execute" instruction performs the actual computation on the CCU. The set–execute model can be supported by an additional "break" instruction, providing synchronization in a sequential consistency programming paradigm.

2.1.2 PlasmaCPU (MIPS R3000) and RTOS

We choose the MIPS R3000 32-bits RISC micorprocessor as a representative of a single-core architecture. More specifically, we employ the Plasma CPU [83] which has been already implemented as a softcore on an FPGA chip. In Figure 2.2, we introduce a block-diagram of the PlasmaCPU microarchitecture. The PlasmaCPU has three-stage pipeline and supports bidirectional serial port, interrupt controller, and hardware timer. The PlasmaCPU is shipped with a fully functional Real-Time Operating System (RTOS) that supports threads, semaphores, mutexes, message queues, timers, heaps, an interrupt manager. As a scheduling scheme, the authors use Round-Robin. The user application can be organized in one or multiple threads following a simplified POSIX PThreads-like [22] application model.

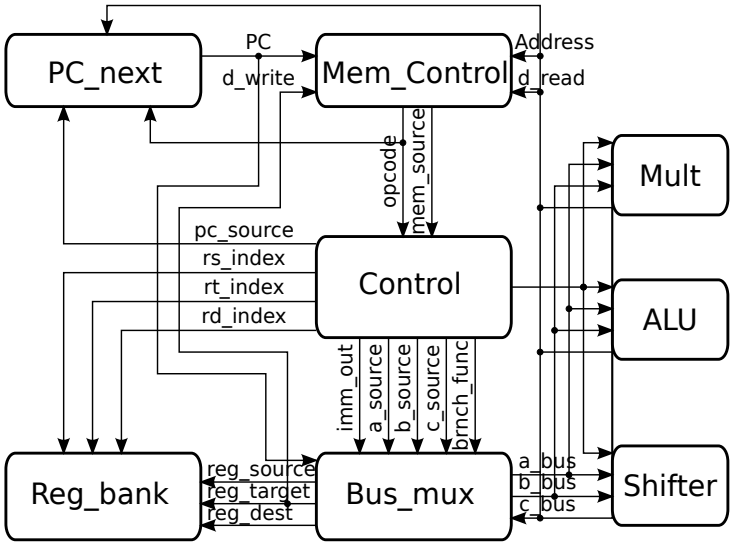


Figure 2.2: The PlasmaCPU (MIPS R3000) architecture

2.1.3 CompSoC and CompOSe

We employ the tiled CompSoC platform [9] as a baseline template for our multiprocessor design. More specifically, we employ the organization of the tiles presented in [11]. Each tile contains one processor core and multiple local memory modules. In Figure 2.3, we present a simplified top-view of the CompSoC platform. In this particular implementation, the system is configured with two tiles connected through an dAElite NoC [96]. The local data memory in each of the tiles is organized in three blocks. The first one is *Dmem*

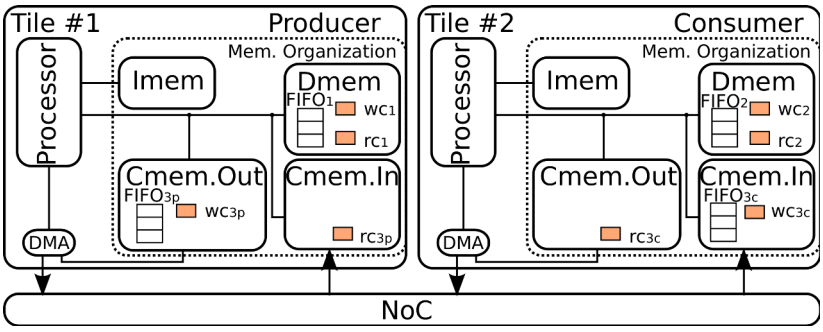


Figure 2.3: Baseline CompSoC architecture

which is employed for local data storage only. The second and the third ones, *Cmem.In* and *Cmem.Out* respectively, are dual-port memories, used for inter-tile communication. The *Imem* is used for storing the applications and RTOS executable binaries. All these memories are accessible by the processor [11].

Each user application is partitioned into tasks, following a data-flow [54] application model. The data-flow graph is mapped on a MPSoC, thus multiple tasks might be running and communicating to each other in parallel. The communication between the tasks is performed through FIFO circular-queues that are memory-mapped and implemented in software using the C-HEAP protocol [73]. A task is ready, i.e., eligible for execution, if there is enough data to operate on, i.e., the input FIFOs are not empty, and there is enough space to produce their data in, i.e., the output FIFOs are not full. Reading and writing in a circular FIFO is implemented with a read counter (*rc*) and a write counter (*wc*). Thus the amount of data in the queue, hence the task status, is determined by the values of these two counters. In Figure 2.3, we depict the *rc*/*wc* and FIFO memory locations. The local (per tile) data memory (*Dmem*) hosts synchronization and data information for tasks which communicate locally, i.e, within the tile. In tile 1, locally communicating tasks exchange data though $FIFO_1$ with the help of wc_1 and rc_1 counters. In tile 2, the FIFO for local communication is $FIFO_2$ with rc_2 and wc_2. In Figure 2.3, we also visualize the location of the *rc*/*wc* and FIFOs in the *Cmem.In* and *Cmem.Out* for a case when a task mapped on tile 1 ($FIFO_{3p}$) communicates with a task on tile 2 ($FIFO_{3c}$).

The CompSoC platform is designed to be predictable and composable. These characteristics are delivered by the hardware and the light-weighted RTOS called CompOSe [36]. The RTOS provides two-level scheduling, *intra-application* and *inter-application*, on each core. The first level uses a static scheduling policy, i.e., Time Division Multiplexing (TDM). The second level is responsible for task scheduling and may follow various policies, such as Round-Robin.

In Figure 2.4, we list the sequence of steps during the communication of two data-flow tasks through a FIFO. A producer is a task, which writes tokens to a FIFO and a consumer is a task, which reads those tokens. Furthermore, the FIFO is implemented by read and write counters, and a circular queue. In Figure 2.4 at instance (1), the producer task checks for its firing rules (i.e, whether there is sufficient space in the FIFO) by: queue_size – (*wc*–*rc*) $\geq$ req_size, where req_size is the required size for the tokens to be written. If the requested space is available, the producer task proceeds to the computation operation. At

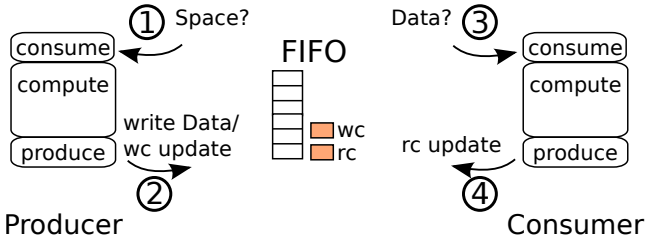


Figure 2.4: Producer–consumer implementation of data-flow programming model

instance ②, producer task completes its computation and writes token(s) and updates wc to the FIFO. Later, at instance ③, the consumer task checks for its firing rules (i.e., whether there is data available) by: $wc–rc \geq$ req_size, where req_size is the required size (e.g. number of tokens) for a single task iteration. If the condition is satisfied, the consumer task proceeds to the compute operation. At instance ④, during produce operation, the consumer updates the rc to the FIFO.

2.1.4 Considered FPGA Chip Families

In this dissertation, we consider three FPGA chip families by Xilinx. We implement the PlasmaPCU augmented with the Molen-style coprocessor on the Xilinx Virtex II XC2VP30 FPGA chip using the XUPV2P Prototyping Board. For the CompSoC platform augmented with the Molen-style coprocessor, we employ two implementations - on the Xilinx Virtex 5 ML510 (XC5VFX130T) and Xilinx Virtex 6 ML605 (XC6VLX240T) evaluation boards. Nevertheless, the presented ideas in the dissertation are not limited to these FPGA chip families.

2.2 Data-driven Parallel Programming Models

The "killer" performance application for the contemporary real-time embedded devices such as mobile phones and smart TVs are encoding and decoding applications for various audio and video formats. Such applications are often referred as streaming applications [103]. Recently, it becomes a common practise to represent streaming applications with data-driven programming paradigm [18]. The data-driven paradigm is a model of computation [70] in which program statements describe the data to be matched and the processing required rather than defining a sequence of steps to be taken [100]. In the

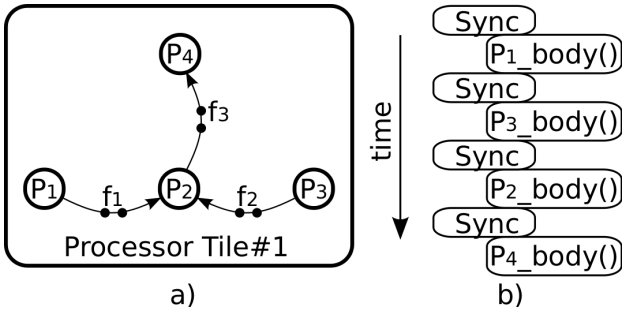


Figure 2.5: a) An application with four processes mapped on a single processor; b) an exemplary execution schedule of the same application;

data-driven programming model, a streaming application, written in high-level abstraction language, is presented as a set of autonomous code segments.

In the domain of streaming applications, the real-time embedded system is often required to deliver predefined performance. By performance, we mean to guarantee end-to-end throughput or latency requirements. Therefore, researchers apply different set of restrictions on the data-driven programming model in order to improve the execution time analysis. Two popular data-driven programming paradigms are Kahn Process Network (KPN) [45] and data-flow [54]. In these programming paradigms, the autonomous computational code regions are referred to *processes* (KPN) or *actors* (data-flow). Both entities have clearly defined input and output communication channels. Each communication channel is presented by First In First Out (FIFO) queue. The synchronization is achieved by exchanging atomic data elements, called *tokens*, passed through the communication channels. Throughout the dissertation, we apply the same terminology. By introducing the following example, we clarify the differences between processes (KPN) and actors (data-flow).

In Figure 2.5.a, we present an application executed on a single processor. The application is composed of four processes - P_1, P_2, P_3, and P_4, respectively. The processes are communicating through FIFO queues - f_1, f_2, and f_3. Furthermore, each FIFO queue can accommodate up to 2 tokens. In Figure 2.5.b, we present an exemplary execution schedule of the same processes.

In Algorithm 2.1, we provide a code snippet of the process body function $P_2\_body(in\, f_1, in\, f_2, out\, f_3)$ as an example of a KPN process. The KPN process is characterized by three basic operations – *read*, *compute*, and *write*. A distinctive KPN property is that these operations can be invoked at any order. During the *read* operation a data preserved in input FIFOs, e.g., f_1 and f_2, is

read. During the *compute* operation, the same set of data is processed and during *write* operation it is written to the output FIFOs. In Algorithm 2.1, depending on the value of the input token, preserved in X, the process *compute* operation varies. Thus, the value of the generated result, denoted as Z, depends on the value of X, i.e., the process can return variable output. Therefore, KPN processes are good design choice to express complex dynamic behaviours, dependent on the values of the inputs. The main drawback of the KPN is its dynamicity, which do not allow computation of throughput or latency of the graph.

Algorithm 2.1 An example of KPN process.
Function $P_2\_body(in\,f_1, in\,f_2, out\,f_3)$

```
1: read(f1, X);
2: if X > 0 then
3:     Z = compute1(X);
4: else
5:     read(f2, Y)
6:     Z = compute2(X, Y)
7: end if
8: write(f3, Z);
```

Algorithm 2.2 An example of CSDF actor.
Function $P_2\_body(in\,f_1, in\,f_2, out\,f_3)$

```
1: consume(X = f1, Y = f2);
2: Z = compute(X, Y);
3: produce(f3 = Z);
```

In Algorithm 2.2, we list the content of $P_2\_body(in\,f_1, in\,f_2, out\,f_3)$ function in case the data-flow programming model is considered. Similarly to KPN processes, the data-flow actors have three operations – *consume*, *compute*, and *produce*. As the operation names suggest, during *consume* operation all input tokens are read from the input FIFOs. During the *compute* operation, all manipulations are performed over the input tokens. During *produce* operation, the data are transferred to the output FIFOs. Contrary to the KPN process, the sequence of these three operations should always remain constant. Moreover, an actor is started only when its *firing rules* are satisfied. By *firing rules*, we refer to the required number of input and output FIFO tokens during single actor iteration. As described in [19], based on the consumption and production rate together with the *firing rule* setup, there are nu-

merous data-flow paradigms such as Single-Rate Data-Flow (SRDF), Multi-rate Data-Flow (MRDF), Cycle-Static Data-Flow (CSDF), and Dynamic Data-Flow (DDF). In this dissertation, we consider CSDF as representative of the analysable data-flow graphs. The CSDF model has the following key characteristics: periodically varying rates and bounded execution time per actor iteration. The main drawback of the CSDF is that it is difficult to express the behaviour of complex dynamic applications.

3

Proposed Approach – Concurrent Execution Models and Programming Paradigm

Note. The content of this chapter is based on the following paper:

P. G. Zaykov and G. K. Kuzmanov and A. M. Molnos and K. G. W. Goossens, ***Hardware Task-Status Manager for RTOS with FIFO Communication****, To appear in Proc. Int'l Conf. on ReConFigurable Computing and FPGAs (ReConFig), 2014*

In this chapter, we describe in detail two of our main contributions, namely: a classification of the concurrent execution models for the processor–coprocessor concept and new programming paradigm supporting concurrent multithreading execution on reconfigurable and multicore platforms. Furthermore, we provide a comparison between the proposed programming paradigm and two popular data-driven programming models - Kahn Process Networks (KPN) and Cycle-Static Data-Flow (CSDF). The chapter concludes with details on the practical applicability of the introduced execution models and the proposed programming paradigm.

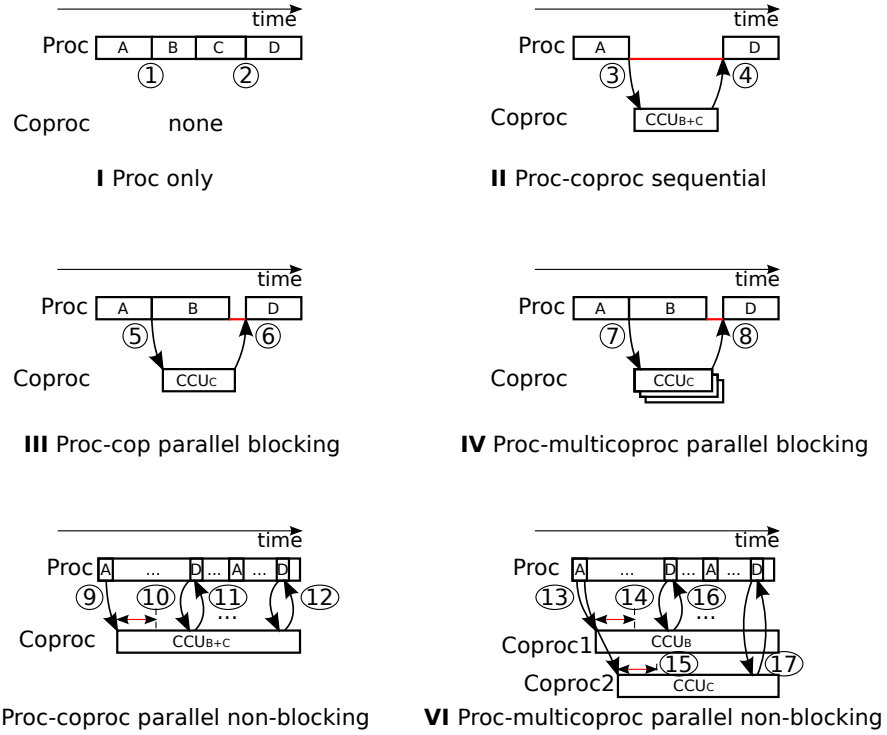


Figure 3.1: Processor–coprocessor execution models

3.1 Concurrent Execution Models

Once the functionality of the coprocessor is identified, the next step is to choose the synchronization model between the processor and the coprocessors and the execution model of the coprocessor. We split processor–coprocessor execution models in six categories based on the employed synchronization mechanism, as visible in Figure 3.1. For the sake of the example we consider four computationally intensive kernels, represented as A, B, C, and D. In the discussion that follows, we assume that only B and C are accelerated on coprocessors.

Below, we describe each processor–coprocessor execution model:

I *Processor only*, as presented in Figure 3.1.I, is used as reference. We consider B and C to be eligible for acceleration. In Figure 3.1.I, the B starts at instance ① and C finishes at instance ②.

II *Processor–coprocessor sequential*, as depicted in Figure 3.1.II, is commonly used to accelerate various computation intensive kernels in a coprocessor. To preserve the functional consistency, a software application is blocked after its coprocessor has started. In Figure 3.1.II, at instance ③, the coprocessor is executed on a hardware Custom Computing Unit (CCU_{B+C}). When CCU_{B+C} finishes execution, it returns the program control to the processor, illustrated in Figure 3.1.II, at instance ④. Depending on the duration of the coprocessor execution and the system requirements, CCU_{B+C} can generate an interrupt or can raise a flag on which the application waits or polls.

III *Processor–coprocessor parallel blocking*, as presented in Figure 3.1.III, allows concurrent execution of a processor (software) and a coprocessor (hardware). With the help of hardware synchronization blocks, like those described in Chapter 4, the CCU_C is started at instance ⑤ and synchronized with processor at instance ⑥. In Figure 3.1.III, the software functionality, denoted as B, finishes earlier than CCU_C. To preserve the application consistency, the software remains blocked until CCU_C completes its execution. Alternatively, B can finish later than CCU_C.

IV *Processor–multicoprocessor parallel blocking*, as presented in Figure 3.1.IV, is an extension of the model in Figure 3.1.III, with multiple coprocessors executed in parallel. With the help of processor-multiprocessor parallel blocking, we gain performance from hardware acceleration and

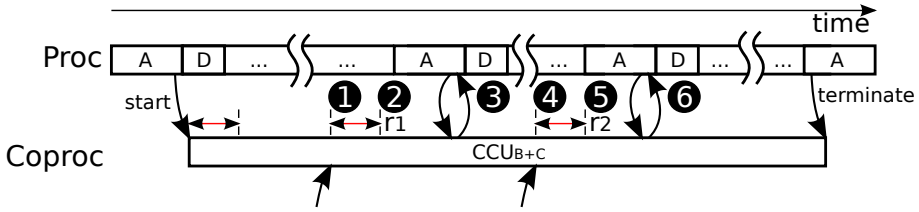


Figure 3.2: Processor–coprocessor parallel non-blocking: an example

parallelism. In Figure 3.1.IV, at instance (7), the coprocessors are started. At instance (8), only after all coprocessors are finished, the software execution is resumed. Cases II and IV in Figure 3.1 are in essence the *task-sequential* and *task-parallel* modes in Chapter 4, respectively.

V *Processor–coprocessor parallel non-blocking*, as introduced in Figure 3.1.V, is one of the contributions in this thesis. To our best knowledge, we are the first to use this execution model in the processor–coprocessor context. Compared to the processor–coprocessor *sequential* and *parallel blocking* models, the software execution in the *parallel non-blocking* model is never blocked during the coprocessor execution. Note that once CCU_{B+C} is started, it finishes only at the request of the processor, i.e., the processor does not need to restart CCU_{B+C} every-time its results are needed. Therefore, in *parallel non-blocking* execution model, the cost to restart the CCU is entirely avoided. In Figure 3.1.V, at instance (9), after CCU_{B+C} is started from the processor, CCU_{B+C} needs several cycles until the newly computed result is available at instance (10) in Figure 3.1. Later, at instances (11) and (12) in Figure 3.1.V, the processor reads back the CCU_{B+C} result.

In Figure 3.2, we present the processor–coprocessor *parallel non-blocking* execution model in details. At instances ❶ and ❹, the CCU_{B+C} processes new data. At instances ❷ and ❺, the coprocessor result, marked as r_1 and r_2, is available to the processor. Note that a new coprocessor r_{i+1} value always overwrite an existing r_i value. Later, at instances ❸ and ❻, the processor fetches the new result from the coprocessor. Within the *parallel non-blocking* execution model, we differentiate two modes of operation: 1. successive approximation and 2. state approximation, respectively. In the successive approximation execution, each consecutive coprocessor result has higher accuracy than the previous one. Therefore, the processor might read multiple coprocessor results until the value with

the required accuracy is available. In the state approximation, the coprocessor result has the most recent status of a software or hardware component. Examples of modules capable to operate in state approximation mode could be potentially any of the services in a single-threaded OS, such as checking task-status and task scheduling policy. We will demonstrate these processor-coprocessor models in Chapter 5. Note that for both *parallel non-blocking* execution models, we assume that the coprocessor result r_i always remain valid until r_{i+1} is available.

VI *Processor–multicoprocessor parallel non-blocking* in Figure 3.1.VI is an extension to case *IV* from Figure 3.1. In Figure 3.1.VI, we attach multiple coprocessors to the processor, where each coprocessor runs in *parallel non-blocking* execution model. At instance (13), both CCUs, CCU_B and CCU_C, are started. The CCUs need several cycles until the newly computed result is ready, marked by instances (14) and (15). Later, at instances (16) and (17), the processor fetches the CCUs status.

3.2 Programming Paradigms

In this section, we introduce a new programming model that supports the processor–coprocessor execution models from Section 3.1. Then, we compare our programming model to two popular data-driven programming models, in particular, Kahn Process Networks (KPN) [45] and Cycle-Static Data-Flow (CSDF) [19], respectively.

3.2.1 The Proposed Programming Model

Nowadays, there are several widespread multithreading paradigms, such as POSIX Threads (PThreads) [22] and OpenMP [24]. Because of the fact that any of the existing multithreading paradigms needs to be modified in order to accommodate management for reconfigurable resources, we propose a new hierarchical programming model. The proposed programming model is applicable as an extension to any of the existing standards.

Hierarchical Programming Model: The proposed hierarchical programming model has explicit communication between threads and between successive tasks of each thread, i.e., communication and synchronization is performed before a task starts or after task completes. We address an embedded system,

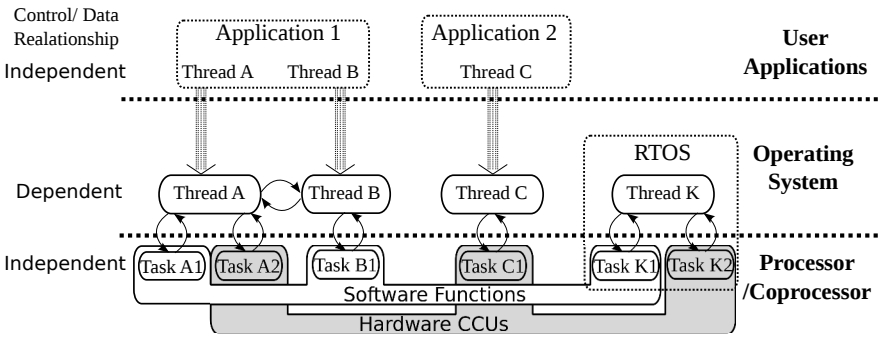


Figure 3.3: The proposed hierarchical programming model: an example

where a processor core has multiple reconfigurable coprocessors. A reconfigurable coprocessor is presented by Custom Computing Unit (CCU), which is an evolution of the Molen reconfigurable microcode processor. In order to simplify the software complexity, we partitioned the executed programming code in three abstraction layers - application, thread, and task. An example is illustrated in Figure 3.3. The application layer accommodates multiple user applications, running independently from each other. Each one of the applications could be composed of one or multiple threads, dynamically created and terminated.

The second level of this abstraction model is the thread layer, where application threads and RTOS kernel service threads co-exist. At this level only, we positioned control and data dependencies between the threads. In the example of Figure 3.3, we assume that Application 1 has two threads - Thread A and B, which are communicating between each other. In our programming model, the communication/synchronization channel is established through the RTOS in tasks K1 and K2.

A user thread contains one or multiple tasks. These tasks are the building blocks of the third layer. Depending on where a task is executed, we distinguish two types of the tasks: a function and a CCU task. When a task is executed in software, we refer to it as to a software function; when a task is executed in reconfigurable hardware, we refer to it as to a CCU task. We adopted the term from the Molen Machine Organization (see Section 2.1). All tasks are non-blocking, i.e., have the following property - when started, they do not communicate with each other, i.e., they do not exhibit any control or data dependencies. In software, a task, being a function, receives a set of input parameters, performs computations and returns a result. These input parameters are transferred through the processor registers and the program stack. In

hardware, the task input parameters are transferred through preassigned exchange registers, described in Section 4.3. A special "execute" instruction is invoked to start the execution of a task on a reconfigurable logic. When the CCU completes, it writes back the computed result to a dedicated exchange register or in a designated location in the shared memory.

Inter-thread and intra-thread parallelism: For simplicity, we assume that the RTOS is running on the processor only, scheduling two user threads - Thread A and Thread B, depicted in Figure 3.4. The corresponding execution pseudo-code of these threads is listed in Figure 3.5. Each thread is composed of multiple tasks, some executed on CCUs, others in software. In Thread A, f_{11} runs CCU_1 and in Thread B, tasks f_{21}, f_{22} are executed on CCU_2 while f_{23} runs on CCU_3. An example of software executed task/function is f_{12} from Thread A. The time slots during which the thread runs on reconfigurable logic are marked by solid lines. The thread execution time on the processor is denoted by a dashed line. The thick solid line marks the time when Thread A is blocked during communication/synchronization with Thread B.

The programming model supports two levels of parallelism - inter- and intra-thread corresponding to two execution modes - task-sequential (processor–coprocessor *sequential*) and task-parallel (processor–multiprocessor *parallel blocking*). The type of the execution mode is determined whether one or more tasks in a thread are running in parallel at the same time. For task-sequential and task-parallel modes, the task parallelism is determined by the location of the special "barrier" instruction in the programming code. Although processor–coprocessor parallel non-blocking execution model has a similar properties to task-parallel mode, we exclude it from the classification for the sake of simplicity.

Task-sequential mode addresses inter-thread parallelism - in this mode, each CCU is executed sequentially. When it is finished - it signals back the processor, and the corresponding thread continues its execution. In task-sequential mode, only one task per thread can be running at the same time. An example is task f_{11} from Thread A in Figure 3.5.

Task-parallel mode addresses intra-thread parallelism - in this mode, multiple hardware CCUs and/or software functions from the same thread could be co-executed in parallel. In Figure 3.5, such tasks are f_{21}, f_{22}, and f_{23} from Thread B. The concurrent execution of CCUs inside of a single thread mimics the traditional out-of-order execution. The CCU synchronization is controlled at the software level by a dedicated barrier instruction, described in more detail in Section 4.3.

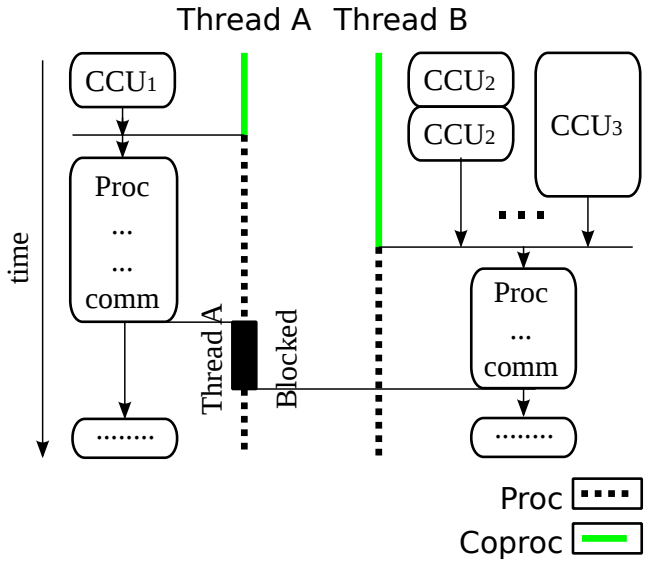


Figure 3.4: Inter- and intra-thread parallelism: an example

Figure 3.4 also visualizes a scenario when the system has to execute CCUs acquiring more reconfigurable resources than the available ones. We assume that all CCU resource requests are always granted and multiple CCUs can be sequentially executed on the same hardware. For example, in Figure 3.5, Thread B - tasks f_{21} and f_{22} use identical CCUs. The only differences between the CCU invocations, from f_{21} and f_{22}, are the values of their input parameters. Therefore, they can be mapped on the same CCU and can be executed sequentially, one after another.

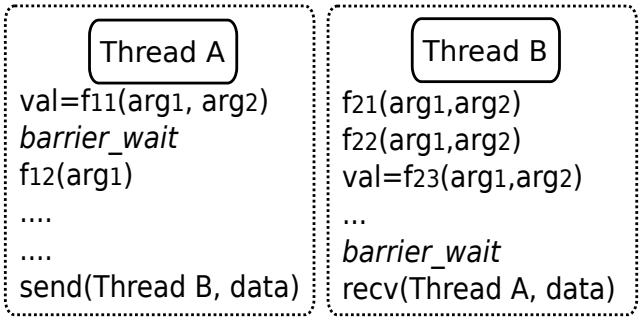


Figure 3.5: Execution code of Thread A and Thread B

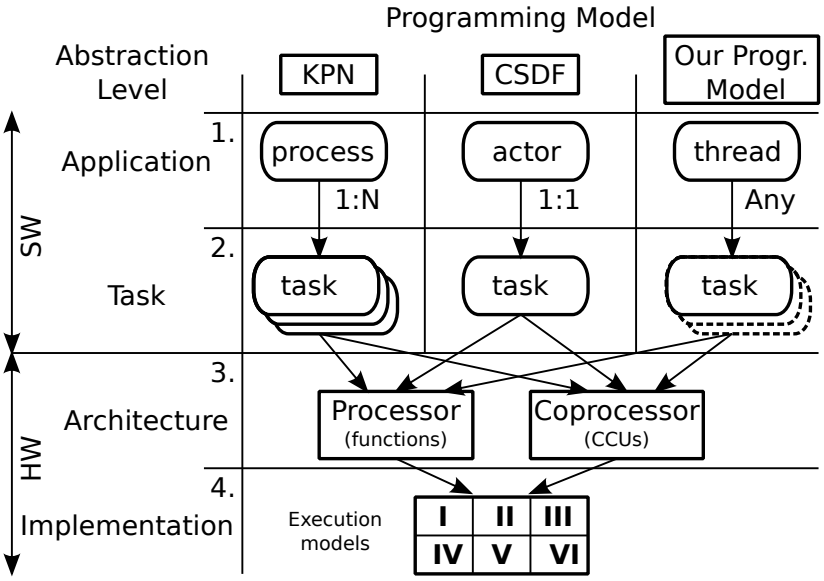


Figure 3.6: Comparison of CSDF, KPN, and our programming model

3.2.2 Comparison of the Proposed Programming Model to KPN and CSDF

In this section, we provide a comparison between the proposed programming model to KPN and CSDF. In Figure 3.6, we present the three programming models altogether. As it was introduced in Section 2.2, the basic entity of the KPN is the *process*. A process intersperses computation and synchronization sections, and thus has computation sections with variable size. That's why, a process can have one or multiple *tasks*. The mapping between process and task is 1:N. The basic entity of the CDSF is the *actor*. An *actor* always have read, compute, and write sections, i.e., actor has one task for a specific input *tokens* and *firing rule* setup. That's why, we define the mapping between an *actor* and a *task* to be 1:1.

In our programming model, we have threads composed of one or multiple tasks. Furthermore, thread communication and synchronization is not limited to FIFO channels only. For example, it could be done through complex data structures using semaphores and mutexes. Moreover, multiple tasks can be executed in parallel, exploiting the intra-thread parallelism. Therefore, we identify that the mapping between thread and task to be *any*, i.e., 1:1 or 1:N.

Independently from the applied programming model, a task can be executed in

one of the two types of computing resources – either on the processor or on one of the reconfigurable coprocessors. If a task is executed on the processor, then we refer to it as a to software function. If it is executed on a coprocessor, i.e., reconfigurable logic, then we refer to it as to a CCU task. We synchronize the processor and the coprocessor, implementing the execution models presented in Section 3.1.

3.3 Conclusions

In the chapters to follow, we apply the processor–coprocessor execution models on multiple Molen-Style coprocessors, referred as Custom Computing Units (CCUs). The proposed coprocessors accelerate computationally intensive and highly variable execution time kernels in hardware. The examined kernels contain parts of application and RTOS, employing different application execution paradigms, e.g., data-flow and PThreads. In such a way, we demonstrate that the proposed execution models are independent from the application execution paradigm and they are very general. Briefly, the functionalities of the proposed Molen-style coprocessors are as follows:

- **Thread Interrupt State Controller**: We examined the processor–coprocessor/multicoprocessor *parallel blocking* execution model by the Thread Interrupt State Controller (TISC). The TISC allows parallel execution of tasks over one or multiple coprocessors. The TISC is implemented as an extension of the Molen-style "barrier" instruction. Our experiments contain synthetic and real applications. Further details of the TISC are revealed in Chapter 4.

- **Hardware Task-Status Manager**: We examined the processor–coprocessor parallel *non-blocking* model by the Hardware Task-Status Manager (HWTSM). The HWTSM CCU is applied on Multiprocessor System-on-Chip (MPSoC), targeting data-flow applications composed on multiple tasks. More precisely, the HWTSM CCU determines the execution eligibility of tasks from FIFO-filling information. More details on the HWTSM CCU are provided in Chapter 5.

- **Remote Slack Distribution**: We apply the processor–coprocessor parallel *non-blocking* execution model for a CCU. The CCU takes part in a run-time technique for slack computation, allocation, and distribution framework targeting applications with tasks mapped on multiple tiles in

an MPSoC. We refer to the slack transferred from one tile to another as Remote Slack. We name the CCU as RS CCU, respectively. We provide more detail for the RS CCU in Chapter 6.

4

Thread Interrupt State Controller

Note. The content of this chapter is based on the the following papers:

P. G. Zaykov and G. K. Kuzmanov, ***Architectural Support for Multithreading on Reconfigurable Hardware****, Proc. Int'l. Symp. on Applied Reconfigurable Computing (ARC), 2011, pp. 363–374*

P. G. Zaykov and G. K. Kuzmanov, ***Multithreading on Reconfigurable Hardware: an Architectural Approach****, Microprocessors and Microsystems (MICPRO), Vol. 36, Issue 8, 2012, pp. 695–704*

In this chapter, we address the problem of organization and management of threads on a multithreading custom computing machine composed of a General Purpose Processor (GPP) and Reconfigurable Coprocessors. We target higher portability, flexibility, and performance of the perspective design solutions by means of a strictly architectural approach. Our proposal to improve overall system performance is twofold. First, we provide architectural mechanisms to accelerate applications by supporting computationally intensive kernels with reconfigurable hardware accelerators. Second, we propose an infrastructure capable of facilitating thread management. Besides the architectural and microarchitectural extensions of the reconfigurable computing system, we also propose a hierarchical programming model. The model supports balanced and performance efficient SW/HW co-execution of multithreading applications. We demonstrate that our approach provides better performance-portability and performance-flexibility trade-off characteristics compared to other state of the art proposals through experiments reported in Chapter 7.

4.1 Introduction

Reconfigurable embedded devices often require multiple applications to be executed concurrently. A common strategy to encapsulate various application functionalities in a conventional software system environment is to use multithreading. Typically, a Real-Time Operating System (RTOS) is employed to manage the dynamic creation, execution, and termination of multiple threads. If the hardware platform is composed of a reconfigurable logic and a General Purpose Processor (GPP), the RTOS should be capable of efficiently mapping the running threads on the available reconfigurable hardware resources. Due to its heterogeneity, the platform complexity and respectively RTOS service cost have grown rapidly. As a result, some of the conventional RTOS kernel services should be optimized to be able to fully exploit the new high performance system capabilities.

The primary objective of this work is to improve the overall performance of the heterogeneous reconfigurable systems following the multithreading execution paradigm. We provide architectural and microarchitectural mechanisms to accelerate RTOS kernels and applications in hardware as an extension to the Molen reconfigurable coprocessor [110]. The proposed programming model efficiently exploits the hardware architectural and microarchitectural augmentations. The introduced architectural model does not depend on either a specific GPP architecture, nor on any reconfigurable fabrication technology. Therefore, our approach is highly flexible and allows easy portability to different reconfigurable hardware platforms. More specifically, the main contributions of this chapter are:

- Clear definition of the problem of multithreading execution on reconfigurable machines in terms of portability, flexibility, and performance.
- Architectural extensions that allow multithreading applications and RTOS to co-execute in software and in reconfigurable hardware. More specifically, we extend the processor interrupt system, the register file organization, and we modify hardware task synchronization at the instruction level.
- Microarchitectural extensions, which support the newly introduced Thread Interrupt State Controller (TISC). A Real-Time Interrupt Service Routine (ISR) is provided to support the new interrupt system.
- A hierarchical programming model capable of providing flexible task migration from software to hardware, exploiting inter- and intra-thread

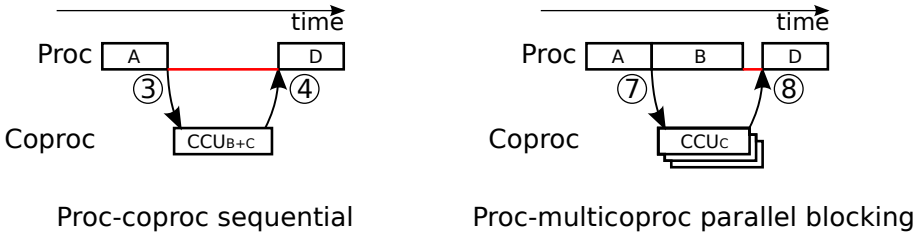


Figure 4.1: Processor–coprocessor sequential and processor–multicoprocessor parallel blocking execution models considered for the TISC

parallelism. These types of logic parallelism are investigated in a real reconfigurable system working physically in task-parallel and task-sequential modes.

- Analytical and experimental comparison between state of the art proposals in terms of performance-portability and performance-flexibility characteristics.

The remainder of the chapter is organized as follows. The problem definition and related work are presented in Section 4.2. Section 4.3 describes the proposed architecture and microarchitectural extensions in detail – hardware components and interfaces, including XREGs, polymorphic instruction implementations, a TISC controller and interrupt management. Finally, Section 4.5 concludes the chapter and outlines some future research directions.

4.2 Problem Definition and Related Work

In this chapter, by introducing a strictly architectural approach, we address the problem of multithreading on heterogeneous systems containing reconfigurable resources. More specifically, we investigate this problem in terms of portability, flexibility, and performance. We consider portability being the ability to port a hardware design to different computing systems. Furthermore, we define flexibility as the ability to change, add, or extract new functionalities from the software into the reconfigurable hardware. Proposing proper architectural support for multithreading on reconfigurable devices can guarantee that the programmer has the necessary control over the system resources while fully exploiting the system performance capabilities. Furthermore, we investigate the behaviour of the reconfigurable system in two modes of multithreading execution, namely: *task-sequential* and *task-parallel*. In task-sequential

mode, multiple reconfigurable accelerators are assigned to a single software thread and executed sequentially, while in task-parallel they are executed in parallel. The task-sequential and task-parallel modes correspond to processor–coprocessor sequential (see Figure 3.1.II) and processor–multiprocessor parallel blocking (see Figure 3.1.IV) execution models, respectively. A snapshot of Figure 3.1 is provided in Figure 4.1 More detail on the two processor–coprocessor execution models are revealed in Section 3.2.1.

An extended version of the related work will be discussed in Chapter 8. Here, we briefly discuss the most relevant works to the TISC functionality and execution model. The problem of efficiently sharing hardware computing resources among multiple threads or processes could be solved statically or dynamically. The former involves the usage of advanced compiler techniques and the latter employs some sort of a run-time system, say an Operating System, or a dynamic resource scheduler. The compiler approach solves the resource management problem by performing different optimizations on the application control dataflow graph. Examples of such embedded architectures with static resource management are: MT-ADRES [119] and UltraSonic [40]. In our work, we focus on the infrastructure for dynamic run-time approaches for resource management, therefore, we do not address any compiler related optimizations.

In the dynamic scheduling approach, assuming an RTOS is employed, parts of the programming code, both from the Operating System and/or from the applications, can be transferred onto reconfigurable logic. A detailed classification of the existing reconfigurable multithreading architectures is presented in [122]. Based on this classification, in the category of dynamic approaches, we discuss the works reported in [64], [112] and [123]. There, the designers improve the system performance and lower the energy consumption by transferring parts of the Operating System kernels to hardware. A similar approach is followed by other authors, e.g., [62], [43], where a dedicated hardware resource manager is proposed. The manager makes decisions using heuristic scheduling and placement algorithms. In contrast to the related works cited above, we propose a system capable of accommodating in reconfigurable logic parts of the user applications, as well as parts of the RTOS. These RTOS services could be responsible not only for the scheduling of hardware tasks, but also for the management of software tasks. The idea of accelerating RTOS routines on reconfigurable logic, such as the scheduling of software tasks only, has already been presented in several research projects, e.g., [25]. Our architectural proposal, however, allows to co-execute the management routines for both software and hardware tasks on reconfigurable hardware.

To our best knowledge, ReconOS [58] and Hthreads [79] are the most relevant projects to our current proposal. In ReconOS [58], the authors propose the idea of having a *delegate* software thread per hardware kernel. The synchronization to/from the hardware kernels is delivered through a software intra-thread synchronization routines. As a result, the authors increase the system portability to different heterogeneous platforms. Compared to [58], our approach achieves better performance than ReconOS while preserving the system portability. Moreover, we demonstrate that our proposal is more flexible compared to ReconOS. It is because the ReconOS design has a limited ability for transferring in hardware RTOS services of single-threaded Operating Systems. In a single-threaded RTOS, the kernel services are sequentially invoked after the occurrence of an external event, e.g., an interrupt. For such a scenario, the concept of having a *delegate* thread for each computationally intensive kernel service has a limited applicability.

In [79], the authors use dedicated RTOS modules for communication and synchronization procedures among hardware threads. The inventors of the Hthreads use a programming model [12] to distribute the running threads among GPP and reconfigurable logic. The major difference between [79] and our proposal is that we are able to migrate both user application and RTOS thread functions while only complete application threads are moved to hardware in [79]. Our model is more flexible, because it supports migration of parts of the user thread and RTOS kernels in hardware identically.

The authors of [53] propose a heterogeneous system running an RTOS Linux. Their system is composed of a single GPP core combined with multiple Hardware Accelerators (HA). One of the contributions in [53] is the communication cost optimization between the HAs and the GPP while an IRQ scheme is considered. The approach followed by [53] is complementary to ours, because the authors reduce the communication cost by improving the software IRQ protocol stack. Therefore, the hardware "barrier" instruction, proposed in this chapter, can be employed at architecture level for achieving even lower system cost. Furthermore, the "barrier" instruction might improve the system scalability when a high number of HAs are executed concurrently on platforms such as [53].

There are also projects such as Silicon OS [69], where the run-time system is completely transferred into hardware. Such an approach is not flexible at all and has only limited potential for future improvements, because the complete run-time system is represented by a complex Finite State Machine (FSM).

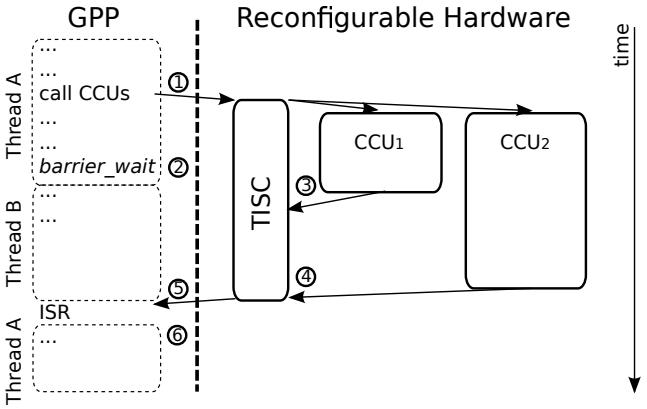


Figure 4.2: A conceptual model of the Thread Interrupt State Controller (TISC) operation

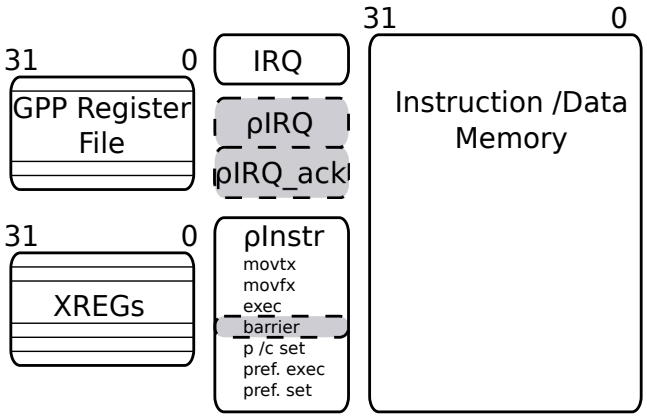


Figure 4.3: Proposed architectural extensions compared to [111] (shaded blocks)

4.3 Proposed Architectural Extensions

We choose the Molen Polymorphic Processor [111] as a base architecture for our current work. The background for the Molen Polymorphic Processor can be found in Chapter 2.1.1. Therefore, we directly proceed with the proposed innovations.

In the original Molen design [111], multithreading was not considered, but in [107], interleaved multithreading (IMT) was addressed and a hardware scheduler was used instead of an RTOS. The achieved simulation performance speedup reported in [107], with an MJPEG benchmark was 2.75, having a theoretical maximum of 2.78. Since these results were quite appealing, we decided to design a system with an RTOS managing multiple concurrent user applications. Moreover, we provide the infrastructure to partition the RTOS and transfer parts of its functionality on reconfigurable logic.

In Figure 4.2, we present a conceptual model of the TISC operation. We demonstarte the TISC operation with an application composed of two threads - Thread A and Thread B, respectively. Furthermore, we assume that Thread A employs two CCUs - CCU_1 and CCU_2, respectively. The threads are scheduled by a preemptive RTOS. In Figure 4.2 at instance ①, Thread A calls the TISC to initiate the parallel execution of the two CCUs. After executing user code, at instance ②, Thread A blocks on a barrier function call and waits for the completion of the associated CCUs. Since the Thread A is blocked, the RTOS schedules Thread B for execution. At instance ③, CCU_1 completes its execution and signls the TISC. At instance ④, CCU_2 completes its execution and signals the TISC as well. At instance ⑤, the TISC identifies that all CCUs linked with barrier has completed. As a result, the TISC initiates an IRQ to the GPP to notify Thread A. Later, at instance ⑥, the RTOS schedules Thread A for execution. With the help of the TISC, we demonstrate the task-parallel mode.

The **proposed architectural extensions** with respect to [111] are visualized in Figure 4.3 by shaded blocks. More specifically, they are: 1. new XREG file organization; 2. modified interrupt system, extended with two software accessible registers - ρIRQ/ρIRQ-ack; 3. we extend the “break” instruction to support synchronization in a multithreading scenario and we call it a “barrier” instruction.

Table 4.1: Original Molen XREGs organization

XREG#	Value
0	CCU Offset
1	Input params, CCU_1
...	...
m	Output param, CCU_1
m+1	Input params, CCU_2
...	...
n	Output param, CCU_2

Table 4.2: Proposed XREGs organization

XREG#	Value
0	Input Thread CCU offset
1	PID_OUT TID_OUT –FREE–
...	...
m	PID_IN TID_IN FID Priority
m+1	Input parameter#1 CCU_1
m+2	Input parameter#2 CCU_1
m+3	Output parameter CCU_1

4.3.1 XREG Organization

In Table 4.1, the original Molen XREG organization is presented. The XREG#0 stores an offset, interpreted as a starting XREG address of the input parameters to the corresponding CCU. In our design, the XREGs are integrated into the GPP core as an extension of the existing register file.

Because the RTOS is running concurrently to the hardware tasks, some of the CCUs might finish at a time when a different thread has started on the GPP. Therefore, a mechanism is needed that allows the CCU to inform the RTOS which thread it corresponds to. Another problem occurs, if a context switching is performed after the CCU input parameters and XREG#0 offset are loaded, just before the "execute" instruction is fetched. Later, when the hardware task starts, it might read a wrong offset value at XREG#0, if it has been changed by another thread. We solve these problems by: 1. modifying the XREG organization, as suggested by Table 4.2; and 2. pushing and popping the contents of XREG#0 to/from the program stack during context switching.

The interpretation of the XREG parameter abbreviations in Table 4.2 is as follows: Process Identifier (PID_IN), Thread Identifier (TID_IN), Function Identifier (FID) and Priority. Note, the Priority might be equal to the Thread priority or custom set by the programmer. The FID is used to differentiate multiple hardware tasks executed in task-parallel mode and having the same $\rho\mu$-code

Table 4.3: Barrier instruction format

Opcode	FID_Num	FID_1	FID_2
...	...	...	...
FID_N	FID_N+1	FID_N+2	FID_N+3

Table 4.4: Example of the barrier instruction

Software Thread	ISR
<do smth>	
Barrier Instruction	
<do smth>	
OS_Semaphore_Pend	
Blocked	
Blocked	OS_Semaphore_Post
<do smth>	

address. If the number of tasks that acquire the same CCU is higher than the number of the available CCUs, then these tasks will be executed sequentially, according to their assigned priorities. We assume that the tasks scheduling is managed by the hardware as a TISC extension. After a CCU computation completes, it writes the result back to an XREG address calculated as the sum of the offset address and the number of input parameters. It also writes back its PID and TID to the PID_OUT and TID_OUT fields of XREG#1. They are used by the RTOS to identify which one of the threads is ready for execution.

There is a possibility that multiple CCUs simultaneously acquire read/write access to the XREGs. The requests are granted according to task Priorities through an XREG Controller, designed as part of the exchange register file. Similarly, we also design an appropriate Memory Controller.

4.3.2 Barrier Instruction

The "barrier" instruction provides synchronization mechanism used by the RTOS to manage the CCUs execution. It is an extended version of the Molen "break" instruction. In task-sequential execution mode, the barrier instruction participates in each CCU invocation. In task-parallel mode, one barrier instruction corresponds to multiple CCU invocations indicating which of them will be executed in parallel.

An exemplary instruction format of the barrier instruction is presented in Table 4.3. The "barrier" instruction is encoded by the "Opcode". The

"FID_Num" field corresponds to the number of CCUs synchronized by the current barrier instruction. The multiple "FID_*" fields indicate the Function IDs of the corresponding hardware tasks blocked.

An example of the usage of the barrier instruction is presented in Table 4.4. After the barrier instruction call is performed, the software thread is blocked on a semaphore by the OS_Semaphore_Pend call. Later, the thread is unblocked after all CCUs, marked by the barrier, have finished their execution. The unblocking is done by the OS_Semaphore_Post function during the Interrupt Service Routine (ISR).

4.3.3 Interrupt Handling

In task-sequential execution mode, after a hardware task has completed, it acquires access to the XREGs. When such access is granted, the CCU writes the computed result back to the corresponding XREG. Next, an interrupt is issued to the RTOS indicating that a task has completed. In task-parallel execution mode, depending on the position of the barrier instruction, the CCU could be marked as finished and possibly be reused by another task without generating any interrupt. After an interrupt is asserted by a CCU, the ISR fetches the content of XREG#1 in Table 4.2 and unblocks the corresponding thread or kernel service. Then, the thread is placed in the RTOS Ready queue and an ρIRQ-ack is sent back to signal the TISC.

4.4 The Microarchitecture

We assume that the GPP has already been extended with the original Molen architecture features and our microarchitectural augmentations are denoted as shaded blocks in Figure 4.4. The "ρInstr. unit" is a Molen style Arbiter/Decoder [110], integrated in the GPP Decode stage.

Our main contribution at microarchitecture level is the Thread Interrupt State Controller (TISC). This unit allows concurrent execution of multiple threads having tasks co-executed in software and in hardware. The TISC controller, illustrated in Figure 4.5, has two finite state machines (FSMs) responsible for instruction predecoding, synchronization and interrupt management of multiple CCUs. The TISC executes the "barrier" instruction at FSM 1: "TISC - New ρ-instruction" in state "Update". The rest of the ρ-instructions are redirected to the Molen style coprocessor at FSM 1 in state "Redirect". When a CCU completes, it uses "end_op" signal to inform FSM 2: "TISC - IRQ Management",

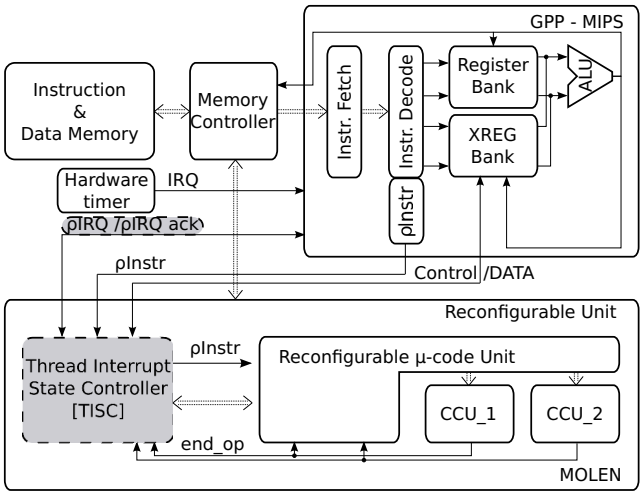


Figure 4.4: Proposed microarchitectural extensions compared to [111] (shaded blocks)

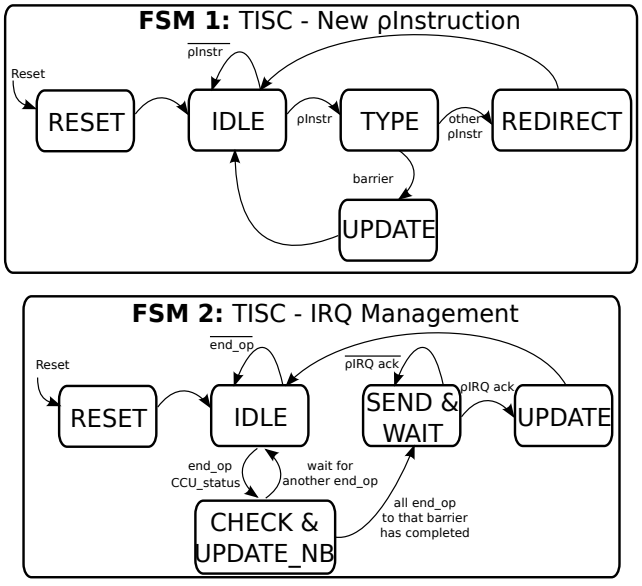


Figure 4.5: TISC Finite State Machines

which switches to "Check & Update_NB" state. The TISC checks whether all "end_op" signals assigned to the corresponding barrier instruction have been activated. If this is the case, FSM 2 asserts interrupt and jumps to "Send & Wait" state to wait for an "ρIRQ-ack" signal. When a hardware task completes execution, it generates an interrupt to the processor. The interrupts are sequentially dispatched to the GPP by the TISC Controller according to their priority. Contrary to other approaches found in literature, e.g., [105], which connect each hardware kernel to a separate interrupt vector, we decided to use only one interrupt vector for all active kernels (CCUs). Thus, the achieved system portability is at the cost of minimal time cost - no more than six additional clock cycles are necessary for the FSMs (see Figure 4.5). It must be noted that the proposed interrupt mechanism is applicable both in preemptive and non-preemptive execution modes of the CCUs.

4.5 Conclusions

In this chapter, we proposed a holistic architectural support for performance efficient multithreading execution on reconfigurable hardware. More specifically, a new programming model for inter- and intra-thread parallelism was introduced and several architectural and microarchitectural improvements were proposed. Experimental results, reported and discussed in Chapter 7 suggest that in order to benefit from our Custom Computing Machine model, a system programmer should consider the following recommendations: First, restructure the programming code and employ CCUs in task-parallel execution mode whenever possible - especially when tasks have short execution times. Second, carefully select the number and type of threads working on CCUs, because such an action could even decrease the system performance. If the system has hard realtime requirements, more advanced scheduling algorithms should be employed.

5

Hardware Task-Status Manager

Note. The content of this chapter is based on the the following paper:

P. G. Zaykov and G. K. Kuzmanov and A. M. Molnos and K. G. W. Goossens, ***Hardware Task-Status Manager for RTOS with FIFO Communication****, To appear in Proc. Int'l Conf. on ReConFigurable Computing and FPGAs (ReConFig), 2014*

In this chapter, we address the problem of improving the performance of real-time embedded Multiprocessor System-on-hip (MPSoC). Such MPSoCs often execute data-flow applications composed of multiple tasks, which communicate through First-In-First-Out (FIFO) queues. The tasks on each processor in the MPSoC are scheduled for execution by an instance of a Real-Time Operating System (RTOS). To improve performance, we propose a Hardware Task-Status Manager (HWTSM) block that reduces the Worst Case Execution Time (WCET) of the RTOS. The HWTSM is a Molen-style Custom Computing Unit (CCU), a coprocessor that determines the execution eligibility of tasks from FIFO-filling information. Furthermore, we propose a new processor–coprocessor execution model, denoted as *parallel non-blocking*. In this model, the HWTSM execution overlaps with the execution of RTOS and user applications. The HWTSM is integrated into the existing CompSoC platform and this entire system is prototyped on a Xilinx XC5VFX130T FPGA chip. In Chapter 7, we report the experimental results with the prototyped HWTSM.

5.1 Introduction

Contemporary embedded systems execute an increasing number of applications that demand high performance. Many of these applications belong to the signal processing domain and usually require the execution of complex audio, video, and telecommunication algorithms in real-time. Often, such applications are partitioned into multiple tasks, which are concurrently executed on a Multiprocessor System-on-Chip (MPSoC). Furthermore, efficient task scheduling on each one of the processors of the MPSoC is required. A widely accepted solution for this scheduling problem is to employ a Real-Time Operating System (RTOS) in software.

The temporal behaviour of real-time applications on the MPSoC must be characterizable, i.e., predictable. In turn, this implies that the RTOS should have a worst-case bound on its execution time, i.e., a Worst Case Execution Time (WCET). Hence, to be able to improve overall system performance, the worst-case RTOS cost should be as short as possible. Last but not least, existing RTOSs may have extra properties meant to ease application design. Composability, proposed and advocated in several real-time systems [35,48], is one of them. Composability means that the behaviour of an application, including its timing, is independent of the presence or absence of any other application. This property is very important for the temporal verification of mixed-time criticality applications that run on the same MPSoC platform [9]. Solutions that improve RTOS performance should not invalidate composability.

We consider an MPSoC architecture in which each processor executes its own instance of the RTOS. The RTOS typically preempts, schedules, and loads user tasks. The exact steps involved in the RTOS execution, however, are dependent on the application programming model. One popular programming model suitable for streaming applications is data-flow [54]. A data-flow application consists of a set of tasks that communicate through First-In-First-Out (FIFO) queues. The RTOS schedules only tasks that are eligible for execution, meaning that they have input data to operate on, i.e., input FIFOs are not empty, and space to produce output data, i.e., output FIFOs are not full.

In this chapter, we address the problem of improving the performance of real-time embedded MPSoCs by reducing the WCET of the RTOS. More specifically, we map one of the most time-consuming RTOS services from software, executed on a processor, to a dedicated hardware coprocessor. In our case this is the service responsible for checking execution eligibility of tasks by using the FIFO-filling information. In Section 2.1.3, we provide more infor-

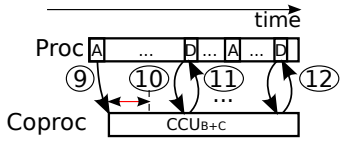


Figure 5.1: Processor–coprocessor parallel non-blocking execution model considered for the HWTSM

mation for the execution eligibility of tasks. We denote the custom coprocessor responsible for checking the task eligibility as Hardware Task-Status Manager (HWTSM). The computed task-status information is used as an input to the RTOS scheduler.

To decide how to implement the HWTSM, we first discuss different existing possibilities for execution on the processor and co-processor(s). Second, we propose a new processor–coprocessor execution model, denoted as *parallel non-blocking*. A coprocessor, operating in *parallel non-blocking* execution model, has the following characteristics: 1. it runs continuously and it does not need to be restarted every time the processor needs the coprocessor; 2. the processor can request a result from the coprocessor at any time; 3. independent of its current status, the response time of the coprocessor is always constant.

We implement the HWTSM with the *parallel non-blocking* model (see Figure 3.1.V), hence its execution overlaps with the software execution of user applications and RTOS services. A snapshot of Figure 3.1 is provided in Figure 5.1. The response time of the HWTSM is very short and constant, equal to five cycles in our prototype, which leads to a significant reduction of the RTOS worst-case cost. Furthermore, the HWTSM reduces the WCET of the RTOS. As we will see in Section 6.4, although the HWTSM executes concurrently with the applications, it does not influence the applications behaviours. The HWTSM has a constant, application-independent response time and does not introduce additional RTOS execution variability. As a result it preserves composability.

Summarizing, the main contributions in this chapter are as follows:

- We propose a Hardware Task-Status Manager (HWTSM), responsible for tracking and computing the status of user tasks. The HWTSM targets data-flow real-time applications employing FIFO communication;

- We introduce a new execution model for the processor–coprocessor paradigm, called *parallel non-blocking* model. We apply this execution model on the HWTSM;
- We explore and demonstrate a HWTSM implementation on an FPGA model of the CompSoC *predictable* and *composable* multiprocessor platform [9];

To prove our concept we experiment with two synthetic applications and two real applications, i.e., JPEG and H.264 decoders, respectively. All applications are executed on a CompSoC platform instance [9], implemented on an FPGA. With the synthetic applications, we investigate the RTOS cost reduction by varying the scheduling policies, because they are the ones that directly affect the number of the task-status computations. Hence, each one of the synthetic applications is specifically designed to explore one of the two types of scheduling policies - static, e.g., time-division multiplexing, or dynamic, e.g., Round-Robin. The experimental results on synthetic benchmarks suggest a reduction in the WCET of the RTOS, compared to a pure software implementation, between 1.1 and 1.8 times for static scheduling policies. For dynamic scheduling policies, this WCET reduction is between 1.1 and 3.0 times. With real applications, the reduction in the WCET of the RTOS with HWTSM is between 1.3 and 1.6 times, for the JPEG and H.264 decoders, respectively. Moreover, we observe that the overall performance gain varies from 2.3% to 7.5%, when the WCET of the RTOS is reduced, respectively. The hardware complexity of the HWTSM scales close to linearly with the number of tasks.

The remainder of the chapter is organized as follows. Section 6.2 discusses the related work. The problem is defined through a motivating example in Section 5.3. Section 6.4 presents the proposed solution. Section 6.5 describes the baseline hardware platform, the internal organization of the HWTSM, and covers the relevant software integration details. In Chapter 7, we report the experimental results with the prototyped HWTSM. The chapter concludes with Section 6.6.

5.2 Related Work

In this section, we discuss related projects that employ hardware acceleration for RTOS. The related work will be discussed in detail in Chapter 8. On top of it, we briefly discuss projects related to the HWTSM execution model and functionality. Then, we compare our approach with hardware acceleration for

dataflow programming models. We conclude the section, by discussing the available processor–coprocessor execution models.

There are numerous examples of embedded systems, such as [25, 57, 64, 112] using an RTOS to co-execute applications in software and in hardware. Although these approaches target coarse-grained reconfigurable embedded systems, the same ideas can be applied with minor changes to more fabrication technologies. In these approaches, the hardware co-processors implementing parts of the RTOS, are used for synchronization and communication between software and hardware. With the help of these hardware co-processors, the application execution time is reduced without affecting the RTOS execution time. Compared to those approaches, our goal is different: besides accelerating the applications, we also aim at reducing the (worst-case) execution time of the RTOS.

Other related approaches [47, 49, 87] execute the RTOS scheduling policy entirely in hardware. Compared to them, we leave the scheduling policy in software. In such a way, the system programmer is not restricted to any particular scheduling policy and can employ the most suitable one to fulfil the system specification.

Some other proposals [69] go even further in RTOS acceleration by completely implementing the RTOS in hardware. Due to the fact that RTOS is substituted by a complex Finite State Machine (FSM), the approach is less flexible and limited for future improvements, e.g., substituting scheduling policy or changing the programming model.

To the best of our knowledge, we are the first to propose a hardware coprocessor performing FIFO tracking and computing task-status for data-flow applications. Hereafter, we list the research projects that employ data-flow programming models and transfer computationally intensive kernels in hardware. The Communication Assists (CA), by Shabbir et al. [90] and by Kyriacou et al. [52], are examples of hardware acceleration in the domain of embedded data-flow systems. In [90], user tasks are executed in non-preemptive mode without employing Operating System, where FIFO communication and FIFO management are entirely integrated in hardware. Moreover, Shabbir et al. [90] do not consider a processor–coprocessor paradigm, i.e., their CA is a stand-alone hardware accelerator. Therefore, the HWTSM might be used to augment their system, if task preemption is supported and RTOS is employed. The proposed CA in [52] provides only a Network on Chip (NoC) abstraction and memory management. It is employed to decouple communication from the computation without affecting the RTOS execution time. Both

approaches [90], [52] are orthogonal to ours in the sense that they accelerate different parts of the Operating System.

Tumeo et al. [106] propose a custom Direct Memory Access (DMA) controller. Similar to CA [90], the proposed DMA controller optimizes execution time of user tasks, by reducing the time cost involved remote memory operations. The proposed Hardware Task-Status Manager has a similar execution behaviour to DMAs, in a sense that both are running in parallel to the software. In fact, the DMA controller can be considered to be executed in what we refer as processor–coprocessor *parallel blocking* model, described in Section 6.4. Furthermore, the DMA signals back the processor by setting a flag and the DMA might have a buffer. The DMA buffer enables temporary storage of multiple DMA invocations. Therefore, their work is different than our new execution model, i.e., processor–coprocessor *parallel non-blocking*. Contrary to the DMAs, the HWTSM is not influencing the communication cost, i.e., we are not directly accelerating the application execution, but rather the RTOS execution. As a result, both, the DMAs and the HWTSM, could be employed together to improve system performance.

In the domain of execution models for coprocessors, Rupnow et al. [86] introduce three preemption methods for hardware accelerators, executed on reconfigurable logic. Contrary to our new execution model, they assumed that if a software thread is preempted, the associated hardware accelerator should be *blocked*, *dropped* or *rollbacked*. A *blocked* hardware accelerator is stalled until the corresponding software thread is activated once again. The generated output of a *dropped* hardware accelerator is discarded. Finally, a *roll-backed* hardware accelerator is restated from the software when the software thread again becomes active. Compared to our classification of processor–coprocessor, their approach falls into what we refer as a processor–coprocessor *sequential* execution model for the hardware accelerators.

Lange et al. [53] propose an embedded system running RTOS Linux. Their system is composed of a single processor combined with multiple hardware accelerators. One of the contributions is an execution model for hardware accelerators. This execution model corresponds to what we refer to as a processor–coprocessor *parallel blocking* model. Therefore, their work is not related to our contributions, which consider a different processor–coprocessor execution model.

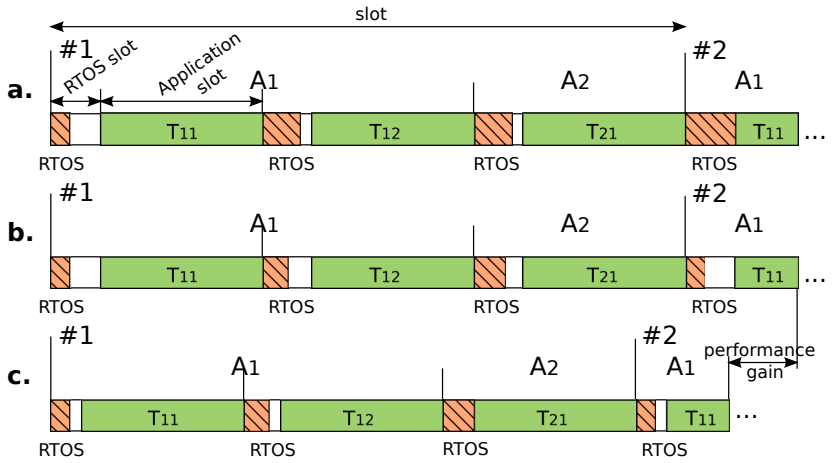


Figure 5.2: RTOS & application execution scenarios. a) RTOS in SW; b) RTOS in SW/HW with slack; c) RTOS in SW/HW with performance gain;

5.3 Motivating Example

We assume that each application is composed by a set of tasks. The tasks are scheduled by an instance of a RTOS, running on each core in a MPSoC platform. Each task is executed in one or more constant time slots, denoted as application slots. Each application slot is followed by an RTOS slot in which the RTOS stores the current task context, schedules, and loads the next task. Because the system has to be predictable, the execution time of the RTOS needs to be boundable. Moreover, if the system needs to be composable, then the size of the RTOS time slot should be constant and equal or longer than the WCET of the RTOS [9]. Our goal in this chapter is to achieve high application performance while preserving the predictability and composability of the system. We improve the application performance by reducing the WCET of the RTOS.

Figure 5.2 provides a motivating example by presenting the execution profile of a predictable and composable real-time embedded system in three scenarios. We assume that an RTOS schedules two applications - A_1 and A_2. Application A_1 comprises two tasks T_{11} and T_{12}. Application A_2 contains only one task - T_{21}. In Figure 5.2.a, we present a scenario when user applications and RTOS kernel services are executed on a single processor. The white spaces indicate processor idle periods, denoted as *slack*, which occur when the RTOS finishes earlier than in its WCET. Figure 5.2.b illustrates a case when at least one of the RTOS services that has a highly variable in execution time is transferred to

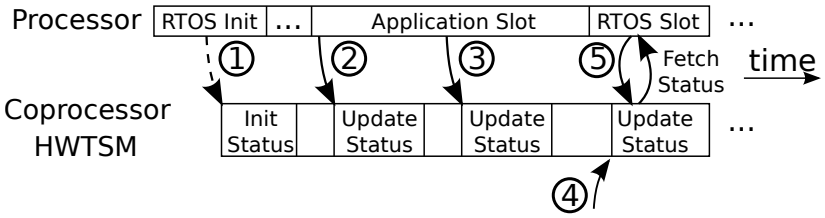


Figure 5.3: HWTSM execution profile

hardware for acceleration. As a result, the WCET of the RTOS is minimized, which preserves predictability and composability, and improves performance.

In Figure 5.2.c, the RTOS time slot is made smaller due to the new reduced WCET of the RTOS. As a result of the short RTOS time slot, the overall system performance is improved. Note, that in all three scenarios, the size of the application slots is left unchanged.

We aim at minimizing the WCET of the RTOS. Two main procedures execute during the RTOS time slot: 1. context switching and 2. scheduling. The minimization of the context switching time is already solved in fine-grained simultaneous multithreading architectures, by having a dedicated register file for each one of the hardware contexts [55]. Thus we aim to improve only the RTOS scheduling procedure running on each of the cores of an MPSoC platform. The scheduling procedure is composed of two parts - computing the status of application tasks and the scheduling policy itself. In this chapter, we aim to accelerate the computing task-status procedure in hardware, and preserve the flexibility to implement any scheduling policy in software.

5.4 Proposed Solution for the HWTSM

In Figure 5.3, we detail the execution profile of the proposed HWTSM for one processor core. After that, since the *parallel non-blocking* processor–coprocessor model (see Figure 3.1.V) is employed, tracking and computing the status of tasks is performed in parallel to the software execution. In our example, at instances ② and ③, user tasks read/write from/into their input/output FIFOs. This results in updated read/write counters of the FIFOs, as well as indirectly triggers the calculation of a new task-status by the HWTSM, as indicated by arrows in Figure 5.3. The HWTSM computational intervals are marked by *Update Status*, and are overlapped with the execution of the user tasks. The status update may also be triggered by remote read/write into a

FIFO from a task mapped on another core, as exemplified at instance ④. Later, at instance ⑤, the RTOS scheduler fetches the task-status from HWTSM. Note that, while the RTOS fetches the status of a task, the HWTSM may update this status, if that is triggered by tasks on other cores.

The status returned by the HWTSM leads always to correct application execution or, in other words, task-status update is conservative. It is not possible that the RTOS sees a task as eligible whereas in fact the task is ineligible for execution, for the following reasons. Producer and cosumer tasks (see Chpater 2) can make each other eligible for execution. For example, when a producer task writes data into a FIFO (the FIFO free space is reduced), a cosumer task becomes eligible for execution. Furthermore, when a consumer task reads data from a FIFO, it creates free space for the producer task to write into. When a task is eligible for execution, i.e., there is enough free space to write or enough data to read from the FIFO, no other task can change its eligibility. A task can only change its status from eligible to ineligible by cosuming or/and producing data. And this can only be done whe the task is executed on a processor. Once a task is updated to ineligible by the HWTSM, the RTOS will see it as such. Thus the only corner case that may lead to incorrect execution is when a task was eligible, it changes to ineligible, but the HWTSM does not detect that fast enough. Once a task is eligible it can change its status only while it is actually executed; its status cannot be changed to ineligible by another task executed on another core. For example, if a consumer task has enough data to read, i.e., is eligible, when the corresponding producer tasks writes more data into the FIFO, the consumer will still be eligible. Hence problems may only occur when the task itself does FIFO reads/writes that renders it ineligible, then it is immediately preempted and the RTOS fetches its status. As the HWTSM is next to the core where the task executes, the task-status update use only local information. Hence the HWTSM is quicker than the time required to switch the context to the RTOS. Thus it is not possible that the task-status update is invisible to the RTOS. Nevertheless, it is possible that the RTOS scheduler fetches the task-status just before the coprocessor updates it, i.e., during the *Update Status* interval. As a result, the RTOS scheduler may get an 'ineligible' status for a task that is in fact eligible for execution. Such a case is application safe, because a different task may be scheduled, and the 'eligible' status will be read in the next RTOS slot.

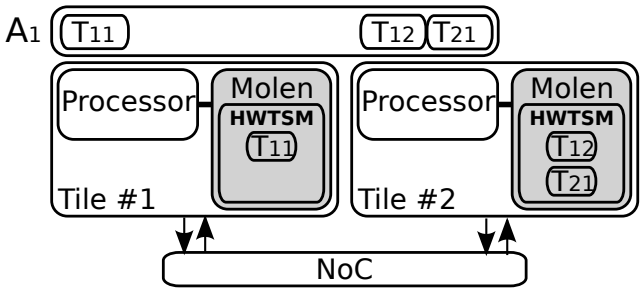


Figure 5.4: Conceptual model of an MPSoC with HWTSM

5.5 Base Hardware Platform and System Implementation

In this section, we describe the tiled-platform and implementation details of the HWTSM and the RTOS modifications.

In this section, we augment the CompSoC platform with hardware acceleration modules using a Molen-style processor–coprocesor design [110]. More specifically, we employ the microarchitecture from [121] used to solve a different, yet related problem, namely management of multiple threads on reconfigurable hardware. We chose the Molen paradigm, because it provides architectural means to efficiently accommodate any software computationally intensive kernel in hardware. Each kernel might be accelerated by one or multiple Molen-style CCUs. In the current chapter, the HWTSM is implemented as a single Molen-style CCU. The coprocessor is controlled through a fixed set of additional instructions [111]. The data transfers to and from the coprocessor might be performed through dedicated exchange registers (XREGs) or through shared data memory.

5.5.1 System Implementation Overview

In Figure 5.4, we present a conceptual model of the CompSoC platform extended with one HWTSM per processor tile. The applications executed on the platform in Figure 5.4, are those considered in the example of Figure 5.2. Application A_1 is mapped on both tiles, while A_2 is mapped on the second tile only. We implement each one of the task-status units, denoted as T_{11}, T_{12}, and T_{21} as separate hardware blocks. These blocks are responsible for computing and updating the status of the assigned software tasks. As a result of this de-

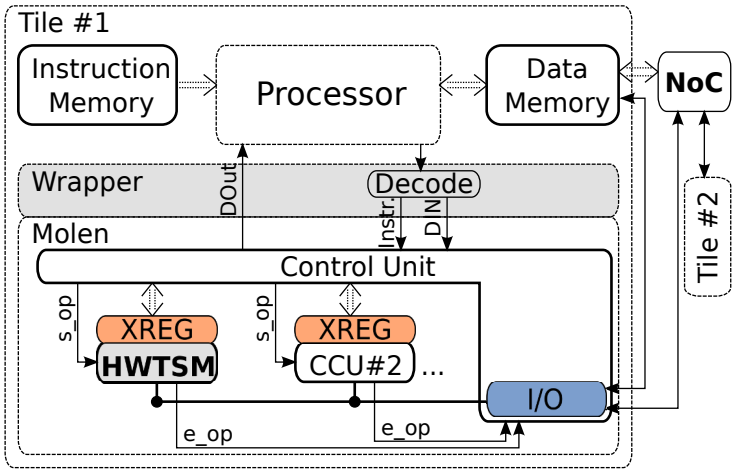


Figure 5.5: The processor microarchitecture with HWTSM

sign choice the HWTSM preserves the composability and predictability of the platform, as explained in what follows.

First, the hardware task-status units do not need to exchange information among each other, thus are completely independent. Plus concurrent computation of the statuses of all tasks is possible. These tasks may belong to the same application or to different applications. Thus, the HWTSM does not create inter-application interference, hence it preserves composability.

Second, as detailed below, the RTOS fetches the task-status by simply reading the registers of the HWTSM. Hence the RTOS perceives the response time of the HWTSM as constant. Moreover, this time is shorter than in a software implementation that would involve a sequential calculation of the available data/space in each FIFO, in turn. Thus the predictability of the RTOS is preserved, and its WCET is reduced when compared to a software solution. Furthermore, the response time of the HWTSM is independent of the task-status and of the number tasks. Therefore, we can scale the number of the hardware task-status units, while the WCET of the RTOS remains constant, which means that the worst-case bounds on the RTOS are lower than in a software implementation.

In addition, it is worth to mention that the HWTSM does not require any modifications to the existing NoC.

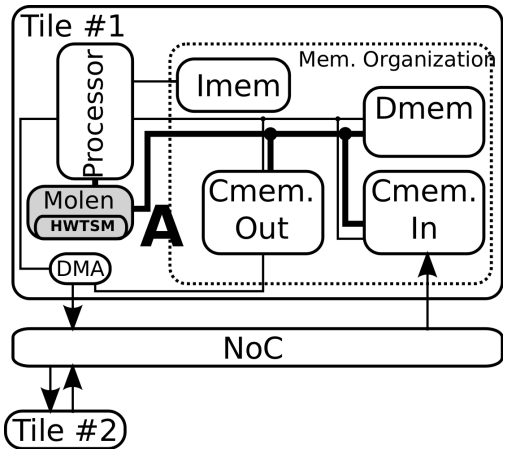


Figure 5.6: HWTSM ntegration to CompSoC – option A

5.5.2 Tile Microarchitecture Modifications

In Figure 5.5, we present the microarchitecture of one of the processing tiles in the CompSoc platform, augmented with a Molen-style coprocessor. The Molen coprocessor is attached to a processor interface using a custom wrapper, marked by a shaded block. As a result, all Molen instructions [111] and input/output data are transferred through that processor interface with the help of a custom software library. With respect to the Thread Interrupt State Controller presented in Chapter 4, we perform the following modifications:

- We propose a new processor–coprocessor *parallel non-blocking* execution model;
- The HWTSM CCU does not initiate interrupts to the processor;
- We attach an *I/O module*, which provides an abstraction interface between the *CCU*s and the external communication protocols, for the NoC interface and for the Local Memory Bus;
- We alleviate the communication bottlenecks between XREG [110] and the *CCUs* by allocating an XREG for each *CCU*;

The shaded blocks in Figure 5.6 and Figure 5.7 present Molen-style coprocessors attached to the baseline CompSoC multicore platform. Based on the platform characteristics, we distinguish three possible options for the HWTSM

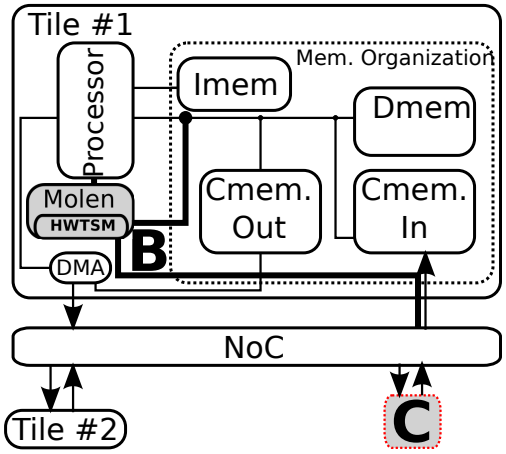


Figure 5.7: HWTSM integration to CompSoC – options B, C

integration. In the first option, denoted by A in Figure 5.6, the HWTSM is attached through a dedicated memory port to each one of the three Data Memory types, i.e., HWTSM reads memories without affecting the processor. As a result, the dual-port *Dmem*, *Cmem.In*, and *Cmem.Out* need to be substituted by three-port memories. We do not consider this option as a viable solution in our design, because multiport memories are expensive in terms of hardware resources. By multiport memories, we denoted those with more than two ports. In the second option, denoted by B in Figure 5.7, the HWTSM is attached to the Data Memory and to the NoC Input buses. Such configuration enables the HWTSM to be executed in the *parallel non-blocking* model, which allows it to track FIFO updates (i.e., snoop the communication) at the exact moment they occur on the corresponding buses. Therefore, the integration of the HWTSM in the existing CompSoC platform is accomplished with a minimum number of modifications. The third option, denoted by C in Figure 5.7, combines all HWTSMs into one dedicated tile. We ignore option C, because updating and obtaining the task-status over the NoC causes immense delays amounting to hundreds of cycles. These delays can substantially increase the execution of the RTOS.

Summarizing, we attach the HWTSM to the CompoSoC multicore platform following option B in Figure 5.7.

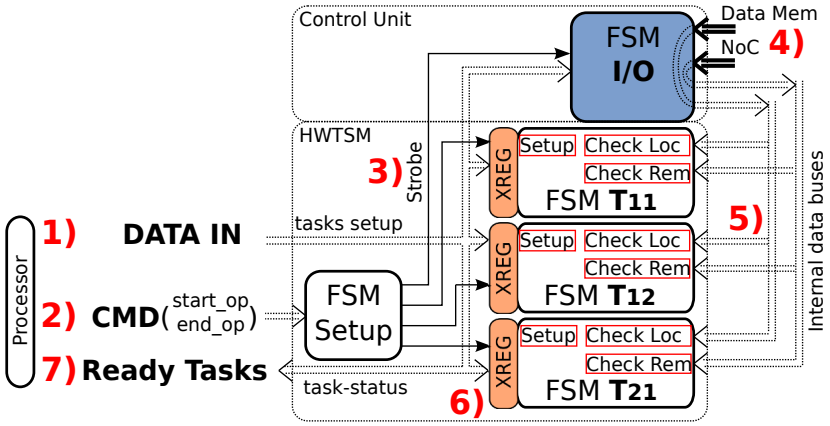


Figure 5.8: HWTSM internal organization

5.5.3 Hardware Task-Status Manager Design

In Figure 5.8, we present the internal organization of the HWTSM for the example of Figure 5.2. For the operation of the HWTSM, there are three basic types of FSMs involved – *FSM T**, *FSM I/O*, and *FSM Setup*. The *FSM T** contains the core functionality of the Hardware Task-Status Manager. By *FSM T**, we refer to any of the *FSM* T_{11}, *FSM* T_{12}, and *FSM* T_{21} in Figure 5.8. Each of the *FSM T** computes and stores the task-status information of T_{11}, T_{12}, and T_{21}, respectively. The number of *FSM T** might vary from one design to another, but it should be equal to the maximum number of tasks among all running applications assigned to the current HWTSM tile. The *FSM T** has an internal memory which preserves the *rc*/*wc* memory addresses, the size of the FIFOs and the number of data elements (tokens) available in each of the FIFOs, associated to a particular task. The *FSM I/O* translates from the external bus protocols, such as NoC interface protocols and Local Memory Bus, to internal buses shared among all Molen-style coprocessors. The *FSM Setup*, is responsible for starting and terminating the execution of *FSM T**.

At run-time, the following events occur, in order as presented in Figure 5.8:

1. A new configuration is transmitted to the *XREGs*;
2. The processor emits the Molen *exec* instruction [111], initiated by the *start_op* signal, indicating that there is a new *CCU* setup available in *XREG*;
3. The *FSM Setup* state machine redirects the *start_op* signal to the corre-

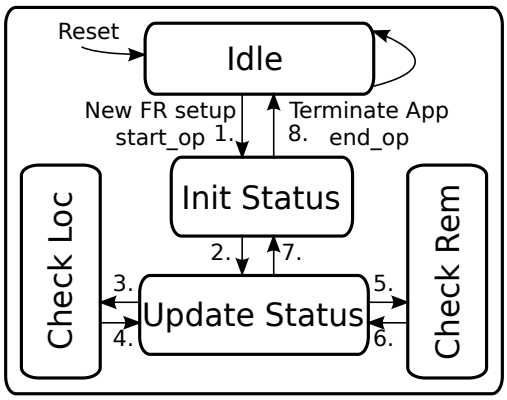


Figure 5.9: *FSM T** states

sponding *FSM T** or *FSM I/O* state machines. The configuration information for the *FSM I/O* contains memory address locations of task's FIFOs, which are part of the local Data Memory space. Once the *FSM T** state machine is started, it continuously tracks the changes to all FIFOs connected to *T** and the new setup FIFO configuration;

4. The *FSM I/O* snoops on NoC interface and Local Memory Bus and detects write operations to the dedicated *rc*/*wc* memory ranges. The *FSM I/O* is a slave on the bus and it does not affect applications bus access timings;

5. If there is a write operation to the tracked memory ranges, then the *FSM I/O* generates a strobe signal and transmits the address and data values to the internal bus;

6. The updated task-status by *FSM T** state machines is preserved in *XREG*;

7. During the RTOS time slot, the RTOS task scheduler fetches task-status information from the *XREG*;

In Figure 5.9, we present the internal states of *FSM T** state machines. During the RTOS Initialization phase, the task FIFO configuration is loaded at *Init Status* state in *FSM T**. During *Update Status*, the task-status is updated in *XREG*. Next, during *Check Loc* and *Check Rem* states, the FIFO reads/writes are tracked on the local memory bus and on the network interface, respectively.

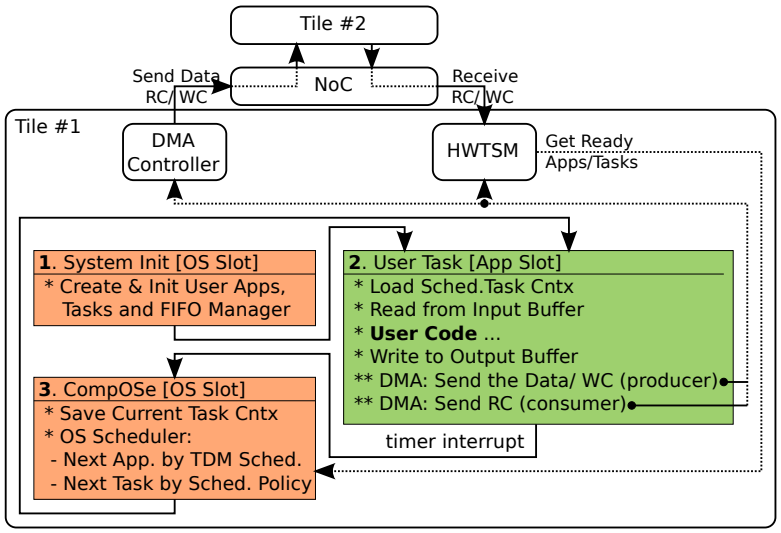


Figure 5.10: CompOSe – application and RTOS time slots

5.5.4 RTOS Extensions

In Figure 5.10, we present the integration of the HWTSM to the CompOSe RTOS [36]. The HWTSM management is performed through a tiny driver that virtualizes the low-level interface. The following stages occur during the execution of the RTOS. In stage 1, during system initialization, all coprocessors are initialized and user applications are created. After this stage, the processor starts the first application time slot. In stage 2, the context of the scheduled task is loaded. The task reads input data from input FIFOs, performs some computations and writes back the results to the output FIFOs. In case an application is partitioned and distributed on multiple tiles, a DMA module could be used to send data and FIFO information over the NoC to the remote tile memory. When a FIFO is updated, the HWTSM instantly detects the *rc* and *wc* changes and updates task-status. The HWTSM operates identically for intra- and inter-tile communication. If a user task finishes earlier than its slot, the processor goes to an idle state. After an interrupt from the hardware timer is raised, in stage 3, the processor starts the RTOS time slot. The context of the running task is saved. The RTOS scheduler gets task-status information from the HWTSM. Based on it, the RTOS scheduler chooses one of the ready tasks according to its task scheduling policy. Note that stages 2 and 3 are repetitive.

5.6 Conclusions

In this chapter, we proposed for an implementation of a time-consuming RTOS part in hardware, as a coprocessor. This coprocessor, called HWTSM, computes the task-status in a real-time MPSoC embedded system running streaming applications. For the HWTSM, we applied processor–coprocessor *parallel non-blocking* execution model, which allows overlapping of the coprocessor operation with the processor operation. As a result, we achieved shorter WCET of the RTOS while preserving the predictability and composability of the original MPSoC. Our proposal is integrated into the existing CompSoC platform and this entire system is prototyped on FPGA chip. In Chapter 7, we report the experimental results.

6

Remote Slack Distribution

Note. The content of this chapter is based on the the following paper:

P. G. Zaykov and G.K. Kuzmanov and A. M. Molnos and K. G. W. Goossens, ***Run-time slack distribution for real-time data-flow applications on embedded MPSoC****, Proc. 16th Euromicro Conference on Digital System Design (DSD), 2013, pp. 39–47*

Low energy consumption is crucial for embedded systems, including the ones that employ tiled Multiprocessor Systems-on-Chip (MPSoC). Such systems often execute real-time applications consisting of several tasks synchronized in a data-flow manner and mapped over different MPSoC tiles. Energy can be saved by lowering the processor voltage and frequency, hence extending the application execution over periods of time otherwise left idle, i.e., exploiting slack. In this chapter we propose a framework to distribute slack information at run-time, intra- and inter-tile, to enable accurate and conservative slack calculation within each tile. The slack is transferred along with the existing inter-task synchronization and as a result it is distributed across the MPSoC with low cost. In each tile, we add a hardware block that calculates the slack received during inter-tile communication and a software library to program this hardware. We integrate this framework into an existing MPSoC platform and we prototype an entire system with two tiles on an Xilinx ML605 FPGA board. We demonstrate the effectiveness of our proposal with a simple, conservative, DVFS management policy applied to an H.264 decoder application. We report the experimental results in Chapter 7.

6.1 Introduction

Nowadays, many modern real-time embedded systems execute computationally intensive, streaming applications. To achieve the desired performance, application designers exploit available concurrency by encapsulating each of the computationally intensive kernels into tasks that may communicate. Many such applications are mapped on a tile-based MPSoC, where tiles are connected through a Network on Chip (NoC). At run-time, in each tile, the application tasks are scheduled by a Real-Time Operating System (RTOS). Tasks are executed for one or more time slots, denoted as application slots. Each application slot is followed by an RTOS slot in which the RTOS stores the current task context, schedules, and loads the next task.

Real-time embedded systems have to be predictable and often should operate within a limited energy budget. A system is predictable if it is possible to accurately characterize its performance. An important performance metric for streaming applications is throughput, i.e., number of output data items produced per unit of time. Typically throughput is guaranteed by analysing, at design time, the critical execution paths of the application under worst-case assumptions [30, 95]. At run-time, each task is invoked repeatedly for an undetermined number of iterations. Such systems may have two basic types of slack: static and dynamic [67]. Static slack may be intrinsically present in an application when not all tasks are on the critical paths that limit the application throughput. Dynamic slack is present when the actual case execution time of a task iteration is shorter than its worst case execution time.

In this context, many approaches aim to reduce energy consumption without affecting the application guarantees. A way to save energy is to conservatively lower the processor operating points, e.g., by dynamic voltage-frequency scaling (DVFS) for each application task [23, 42, 67, 116]. However, existing methods cannot observe the entire static and dynamic slack that is present in an application at run-time, leading to energy waste.

In this chapter, we address the problem of accurate slack observation in real-time, data-flow applications mapped on MPSoCs. We propose a framework for conservative slack accounting and distribution between tiles. We perform slack accounting with the help of timestamps. Furthermore, the main features of our proposal are, as follows: 1. slack is distributed between tasks, potentially mapped to different tiles, along with the existing inter-task synchronization, 2. slack can be distributed in two directions, i.e., from producer tasks to consumer tasks and vice versa, and 3. static and dynamic slack is addressed.

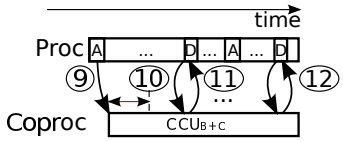


Figure 6.1: Processor–coprocessor parallel non-blocking execution model considered for hardware coprocessor in the slack distribution framework

Our framework consists of a hardware coprocessor and a software library to provide support for it. The coprocessor runs in parallel non-blocking model (see Section 3.1). In Figure 6.1, we briefly introduce the operation of the parallel non-blocking model. Furthermore, we integrate the coprocessor and the software library into an existing MPSoC platform and we prototype the entire system on a Xilinx ML605 FPGA board.

The remainder of the chapter is organized as follows. In Section 6.2, we compare our proposal with the related state of the art. In Section 6.3, we introduce the application and platform models. In Section 6.4, we present the concepts behind our solution and an example for intra-tile and inter-tile task communication. In Section 6.5, we provide the implementation details. The chapter concludes with Section 6.6.

6.2 Related Work

In this section, we discuss approaches in 1. slack management, distribution, and power-aware scheduling for data-flow applications, and two timestamps.

Nelson et al. [71] propose to reduce the energy consumption using static slack, when an application is mapped on multiple tiles. The analysis employs a custom design time tool calculating the Maximum Cycle Mean (MCM) [95] (intuitively, the length of the critical path divided by the number of tokens on the path) of the application graph. The tool finds lower clock frequencies for the tasks that do not belong to the MCM. One of the limitations of this approach is that compile-time tools may not observe all available slack. Other compile-time (static) tasks mapping and DVFS schemes for MPSoCs are based on the application data-flow graph [42, 85, 116]. Compared to these approaches, we propose a run-time technique that has the potential to improve on design time solutions.

Other approaches [44, 67] propose run-time methods to observe the dynamic slack locally in each MPSoC tile. If a task finishes early, DVFS is conservatively performed for the next executed task of the application. Those approaches accurately observe all dynamic slack within a tile, however early completion in one tile may also result in early start in other tiles. In such a case, observing the slack locally in each MPSoC tile might not be efficient enough. Compared to these approaches, we distribute the slack information between the tiles which overcomes this limitation. Moreover, we address the static slack as well.

Carta et al. propose to minimize energy consumption in a pipelined MPSoC architecture [23] by using linear and non-linear feedback control schemes. The considered pipeline architecture resembles the execution of a streaming application. Furthermore, Zamora et al. [120] utilize stochastic automata networks for system-level performance/power analysis and trade-offs in designing of multimedia, streaming applications. However these approaches target the soft real-time domain, hence the throughput guarantees are not hard.

Several examples of systems that utilize timestamps to share timing information exists [21,77].The first proposes a fully synchronous MPSoC. Timestamps are utilized to synchronize heterogeneous IP blocks which might be operating at various frequencies. In this way real-time guarantees are offered to applications. We use the timestamps in a different context, to compute the slack information that is sent from one tile to another. The approach in [21] assigns timestamps to the arrival of external events in a hard real-time system. This information is used to schedule ready tasks. Similarly, we register the arrival time of the synchronization information. However, unlike existing work, we utilize this information for slack calculation to enable energy management.

In summary, to the best of our knowledge, our proposal to distribute information among tiles together with synchronization, for the purpose of enabling accurate, conservative slack observation in data-flow applications is novel. Moreover, it can augment existing state of the art policies to further save energy.

6.3 Prerequisites

This section introduces the application and the platform models that are useful to understand the rest of this chapter.

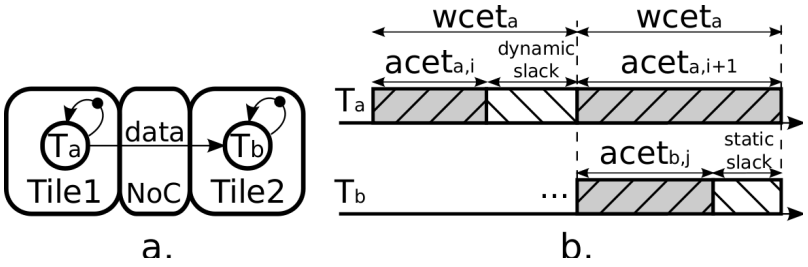


Figure 6.2: Producer-consumer example: a) considered application; b) static and dynamic slack.

6.3.1 Application Model

A data-flow application, A, consists of a set of tasks that communicate via tokens through first-in-first-out (FIFO) channels of bounded-size. Task execute indefinitely, iteratively, processing tokens from input FIFOs and producing tokens into output FIFOs. As we target the real-time domain, we consider that each task T_a has a worst case execution time, $wcet_a$, known design time. Each task iteration i, also denoted as $T_{a,i}$, has an actual execution time, $acet_{a,i}$, unknown at design time. If a task T_a produces tokens that a task T_b consumes, T_a is denoted as the predecessor of T_b and T_b the successor of T_a. A task is eligible for execution, or ready, if it has sufficient tokens in the input FIFOs and space in the output FIFOs. The eligibility of a task T_a is preserved in its state, referred to as $state_a$.

The maximum throughput of an application is given by the Maximum Cycle Mean (MCM) [30, 95] Intuitively, this is the length of the critical path in the task graph devided by the number of tokens. Static throughput analysis utilizes the worst-case execution time of tasks. For simplicity, hereafter the explanations consider single-rate data-flow applications.

In Figure 6.2.a, we consider a data-flow application with two tasks, a producer and a consumer. In the example two basic types of slack are distinguished upon: static and dynamic, as graphically presented in Figure 6.2.b. Intuitively, the static slack occurs because not all task are on the critical path. Static slack is also denoted as the maximum deadline extension in the literature [68, 95]. When the two tasks are executed on different tiles, their iterations may overlap. Consider that $wcet_a > wcet_b$ and there is enough space in the FIFO. We can delay the finish of any iteration j of T_b until the finish of another iteration $i+1$ of T_a and the throughput of the application will remain the same. Or in other words, in this example T_b has $wcet_a - wcet_b$ static slack. Dynamic slack occurs

when the actual case execution time of a task iteration is shorter than its worst case execution time, e.g., $wcet_a - acet_{a,i}$ for T_a in the example in Figure 6.2.

6.3.2 Platform Model

The targeted MPSoC consists of a number of processor tiles, hereafter shortly denoted as tiles, communicating via a Network on Chip (NoC). A tile typically comprises of a processor core, a set of memory blocks, and Direct Memory Access (DMA) modules. As we address real-time systems, we assume that the NoC offers guaranteed service, e.g., maximum throughput and minimum latency. We assume that each tile can be scaled independently, i.e., the tiles and the NoC are in own clock domains, and a time translation between different clock domains is possible (the clocks are mesochronous or there is a slower, reference clock). For the purpose of distributing slack among the tiles, we need to have time-translation functions to the referenced clock.

Application tasks may be mapped to different tiles. One tile may be shared between several applications. An RTOS executes on each processor core. Processor scheduling is at two levels, as follows. At the first level, the RTOS allocates fixed time quanta denoted as slots to each application, in a time-division multiplexing (TDM) fashion. Hence the inter-application level scheduler is preemptive. As applications are completely isolated, an application perceives its time as continuous, although in practice it may be preempted. We can consider that each application has a virtual-time consisting of the set of slots allocated to it. By knowing the TDM allocation, one can translate the application virtual-time in the physical-time of the tile and vice versa.

The two time-line translation functions are as follows:

$$\begin{aligned} time_{phy} &= f_{v\_p}(time_{vir}, A, tile), \\ time_{vir} &= f_{p\_v}(time_{phy}, A, tile). \end{aligned} \tag{6.1}$$

where $time_{phy}$ is the number of cycles in the tile-physical time, $time_{vir}$ is the number of cycles in task-virtual time, (both measured at the maximum tile frequency level), A and *tile* are the considered application and tile, respectively.

At the second level, tasks are scheduled within an application. Typically, for data-flow applications the intra-application scheduler is non-preemptive, in the sense that the task scheduler is called only after a task iteration has finished. Tasks are scheduled only if they are eligible. Hence, once a task iteration starts it is guaranteed that it finishes without blocking. This means that idle time can occur only between iterations.

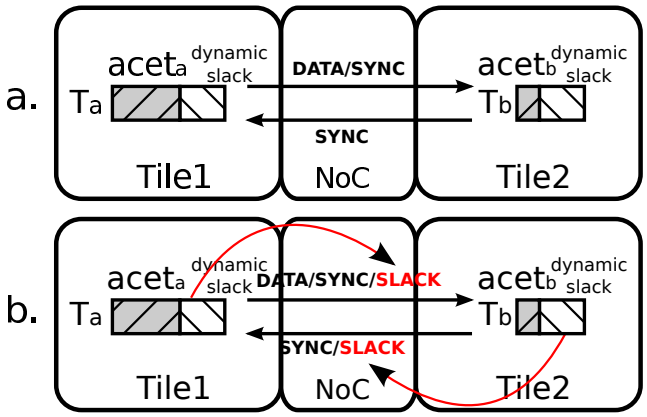


Figure 6.3: A conceptual model for slack information distribution by: a) Intra-tile technique [67]; b) Our inter-tile technique with dynamic slack.

Inter-task communication is implemented by memory mapped software FIFOs employing the C-HEAP protocol [73], where each FIFO stores a limited number of data elements (tokens). The number of the available tokens is determined by the values of the read counter (*rc*) and write counter (*wc*) of the FIFO. For each FIFO, the consumer and the producer side are responsible for updating the write counter and the read counter, respectively. Further details are provided in Chapter 2.2.

When two communicating tasks are mapped on different tiles, for each token, the NoC travel time is bounded by a worst-case traveling time *wctt*. This time depends on, e.g., the parameters of the NoC, the token size, and it is known at design time, for each FIFO [38]. As the focus of this chapter is tile slack, to simplify the notation, in the rest of this chapter we will use *wctt* to denote the worst-case traveling time in general. This notation can be detailed per FIFO and per connection.

6.4 Proposed Solution

In this section, we first outline the concept of our framework for inter-tile slack distribution and second we introduce the equations behind.

6.4.1 Conceptual Solution

In Figure 6.3, we compare the conceptual models for slack information distribution employing intra-tile technique [67] and our inter-tile technique. In FIFOs, data is transferred from T_a to T_b, while the synchronization information is transferred in both directions (see Figure 2.4). In Figure 6.3.a, we present a case in which the slack is computed locally, within the tile, e.g., [67]. In Figure 6.3.b, we present a case when synchronization is augmented with slack information for each one of the communicating tasks. As a result, we can distribute slack in both directions - from producer to consumer task and vice versa.

The distributed slack among the tiles can be static and dynamic. We model static slack as the time at which a task has to finish its next iteration. Note, this reference is not the static slack, but rather the information required to calculate it. Furthermore, we model dynamic slack as the duration of time that the current iteration has ran ahead, i.e., the difference between worst-case execution time (wcet) and actual-case execution time (acet). At run-time, we transfer static and dynamic slack in a single combined value.

Two ways to represent the slack are possible – either a relative or an absolute value. We choose a relative value representation because we target distributed systems where the global notion of time is often missing. Furthermore, a relative time value can be measured in different time-domains. In the first one, the relative time is measured in the time-domain of the application (application virtual-time). In the second one, the relative value is measured in the time-domain of the tile, taking into account the RTOS costs and the time when other applications might be running (tile physical-time). A slack measured in application virtual time-domain in one tile and transferred to another tile can be interpreted wrongly. To avoid this, we transform the slack from the virtual to physical time-domain, after which it can be transferred safely to the other tile.

We extend the static and dynamic slack classification with two new types of slack, *tile* (S_{tile}) and *remote* (S_{remote}), depending on the location of the slack in the system from the perspective of the tile. The S_{tile} is shared among all tasks of an application in a tile. Once it is distributed to another tile, we refer to it as remote on the tile that receives it. Eventually, in the remote tile, S_{tile} is updated with the received S_{remote}. The S_{tile} and S_{remote} contain static and/or dynamic slack. If multiple applications are running on the tile, then each application will have its own S_{tile} and S_{remote}. Slack values are updated at each task iteration start and finish; nevertheless, for brevity and readability we omit the indexes

of each iteration.

Summarizing, our proposal has the following features:

- We distribute slack from one tile to another only during existing inter-task synchronization; this reduces costs compared to dedicated slack communication;
- We distribute slack in both directions, i.e., from the producer to the consumer task and vice versa; in this manner we distribute the slack in an unified way during any inter-task synchronization;
- We distribute static and dynamic slack in one combined value; this aims to increase the potential for energy saving.

Our slack distribution framework is applicable for all three mapping possibilities for multi-tasking applications running on a tiled MPSoC:

1. Application is mapped as such (e.g. all tasks are on one tile only) that there is only intra-tile communication;
2. Application is mapped on two or more tiles such that the tasks involved in inter-tile communication do not have any intra-tile communication;
3. Application is mapped on two or more tiles such that the tasks involved in inter-tile communication do have intra-tile communication;

We introduce Case 1 in Section 6.4.2 and Case 3 in Section 6.4.3.

6.4.2 Intra-tile Slack Distribution

For intra-tile slack distribution, we explain our framework by means of an example presented in Figure 6.4. Figure 6.4 illustrates the moments that are relevant to slack computation, allocation, and distribution during two consecutive task iterations, $T_{a,i}$ and $T_{a,i+1}$. The application is the same as in Figure 6.2. Since all tasks are mapped in one tile only, we detail the transmission and reception of S_{tile}.

Transmission of S_{tile}: In Figure 6.4, the first event that we consider is the start of $T_{a,i}$, at instance ①. Note that at this point slack may already exist on the tile and a management policy may have allocated a part of it to $T_{a,i}$. We refer to the allocated slack as $S_{allocated}$.

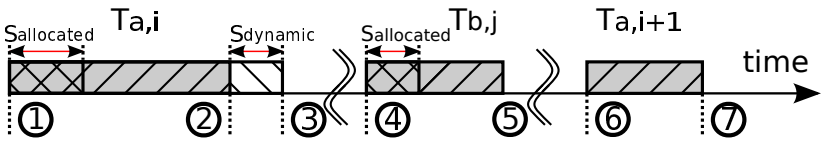


Figure 6.4: Slack computation, allocation, and distribution for intra-tile task communication

Second, at instance ② the $T_{a,i}$ finishes earlier than at worst case, marked at instance ③. Therefore, the dynamic slack for $T_{a,i}$, in application virtual-time, is computed as:

$$S_{dynamic_{vir}} = wcet_a - acet_{a,i} \tag{6.2}$$

As a result, at instance ②, the tile slack is updated with the newly computed dynamic slack:

$$S_{tile_{vir}} = S_{tile_{vir}} + S_{dynamic_{vir}} \tag{6.3}$$

Reception of S_{tile}:

Later, at instance ④ the consumer task $T_{b,j}$ starts its j-th iteration. For intra-tile communication, application tasks share the S_{tile} based on dynamic slack only (no static slack is considered), while remote slack is equal to zero. The slack policy (f_{slack}) allocates slack for $T_{b,j}$, and potentially scales its operating point:

$$S_{allocated_{vir}} = f_{slack}(S_{task_{vir}}) \tag{6.4}$$

If not all slack is allocated, the remaining tile slack before starting the scheduled task is updated, as follows:

$$S_{tile_{vir}} = S_{tile_{vir}} - S_{allocated_{vir}} \tag{6.5}$$

Let assume that slack policy allocates all of the S_{tile} to $T_{b,j}$. At instance ⑤, the $T_{b,j}$ finishes with its $WCET_{b,j}$. Next iteration of the producer task $T_{a,i+1}$ starts with S_{tile}=0 at instances ⑥ and finishes with $WCET_{a,i+1}$ at ⑦.

In Section 6.4.3, we extend the example towards inter-tile slack distribution.

6.4.3 Inter-tile Slack Distribution

Many of the steps in this section are the same as for the intra-tile slack communication. For sake of clarity, we repeat them again. Figure 6.5 illustrates the moments that are relevant to slack computation, allocation, and distribution

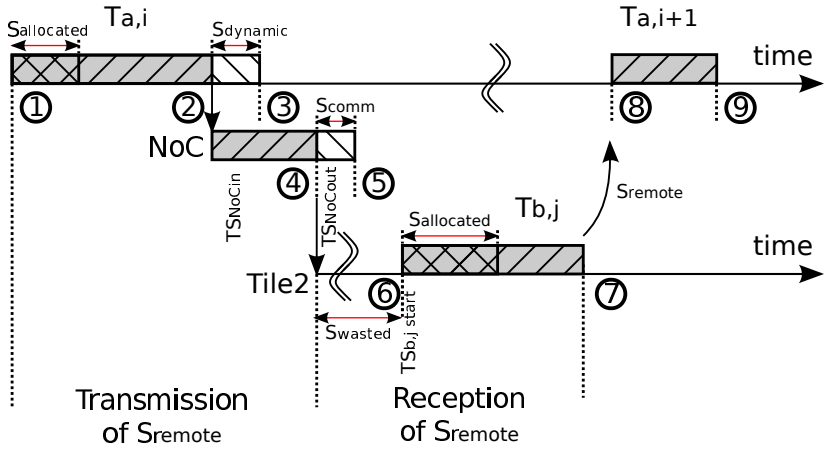


Figure 6.5: Slack computation, allocation, and distribution for inter-tile task communication

during two consecutive task iterations, $T_{a,i}$ and $T_{a,i+1}$. The application is the same as in Figure 6.2 and each of the tasks is mapped on a different tile. Next, we detail the transmission and reception of S_{remote}.

Transmission of S_{remote}:

The first event that we consider in Figure 6.5, is the start of $T_{a,i}$, at instance ①. Note that at this point slack may already exist on the tile and a management policy may have allocated a part of it to $T_{a,i}$.

Second, the $T_{a,i}$ finishes earlier than at worst case, at instance ②. Therefore, the dynamic slack for $T_{a,i}$, in application virtual-time, is computed as:

$$S_{dynamic_{vir}} = wcet_a - acet_{a,i} \tag{6.6}$$

As a result, at instance ②, the tile slack is updated with the newly computed dynamic slack:

$$S_{tile_{vir}} = S_{tile_{vir}} + S_{dynamic_{vir}} \tag{6.7}$$

Since the tasks are mapped on different tiles, in Figure 6.5 at instance ②, we compute the S_{static} as follows:

$$S_{static} = stdelay_{a,i+1} + \begin{cases} wcet_a & \text{if } state_a = Ready, \\ 0 & \text{otherwise.} \end{cases} \tag{6.8}$$

where $stdelay_{a,i+1}$ is the starting delay, i.e., the duration between the current moment and the earliest starting time of the next iteration, $i+1$, of T_a. A starting delay may exists, because, in general, the processor may be shared among

multiple tasks of the application, hence several other tasks may be scheduled before $T_{a,i+1}$. S_{static} is the latest moment in time (relative to instance ②) until when the predecessors and successors of T_a have to finish their next iterations in order to respect the throughput constraint. If the starting delay cannot be calculated (for example, because the scheduling policy is dynamic) it is conservatively set to zero. If the next candidate for execution is the currently finishing task, then the starting delay is also zero. In case iteration *i+1* of T_a is already eligible for execution ($state_a = Ready$) at the time when the current iteration completes and the execution of the $T_{b,j}$ finishes by $wcet_a$, then the application throughput is met. If T_b is on the critical path, then the transferred $wcet_a$ to T_b results in zero cycles slack (see Equation 6.15).

Furthermore, at instance ②, we compute the remote slack as the sum of application slack and static slack, as follows:

$$S_{remote_{vir}} = S_{tile_{vir}} + S_{static_{vir}} \tag{6.9}$$

As introduced in Section 6.4.1, we distinguish two types of remote slack:

$$S^{type}_{remote} = \begin{cases} Dyn & \text{if } S_{static_{vir}} = 0, \\ StaDyn & \text{otherwise.} \end{cases} \tag{6.10}$$

where *Dyn* models that remote slack includes only dynamic slack and *StaDyn* models that remote slack includes static and dynamic.

As a last step, we translate the remote slack value from task-virtual to tile-physical time-domain:

$$S_{remote_{phy}} = < f_{v\_p}(S_{remote_{vir}}, A, tile), S^{type}_{remote} >, \tag{6.11}$$

where *A* and *tile* are the target application and tile, respectively. Note that the $S_{remote_{phy}}$ also contains S^{type}_{remote}.

Reception of S_{remote}:

After the remote slack is calculated, it is sent to the predecessors and successors of T_a, along with the synchronization information, over the NoC. Because at run-time the actual NoC latency might be smaller that the worst-case, the NoC can generate a type of dynamic slack, which we refer as *communication slack* (S_{comm}). For example, at instance ④, the data arrives to the destination tile earlier that the worst-case which is illustrated by ⑤. We compute the communication slack as follows:

$$S_{comm_{phy}} = wctt - (t_{NoCout} - t_{NoCin}), \tag{6.12}$$

where t_{NoCout} is the timestamp when $S_{remote_{phy}}$ arrives on the destination tile. The t_{NoCin} is the timestamp when $S_{remote_{phy}}$ initially enters the NoC. t_{NoCout} and t_{NoCin} are measured in the NoC time-domain.

After the data has arrived, the consumer task ($T_{b,j}$) is started, at instance ⑥. We define the time between instances ④ and ⑥ as wasted slack, as follows:

$$S_{wasted_{phy}} = st_{b,j} - t_{NoCout}, \tag{6.13}$$

where $st_{b,j}$ is the tile physical starting time of iteration j of T_b. For applications with more tasks running on a tile, we may employ the starting time of any of the application tasks (st_A) instead of $st_{b,j}$, if we associate the slack information with application and not with a task. In Section 7.4, we conduct experimental study for the two cases, i.e., remote slack associated with a task and with an application, that trade accuracy of the computed slack for computation cost. If the NoC does not share a common source for clock frequency with the tiles, then t_{NoCout} in Equation 6.13 should be transferred from the NoC physical time-domain to the tile physical time-domain.

In Figure 6.5 at instance ⑥, we calculate, what has left from the remote and communication slacks after the wasted slack is subtracted, i.e., X, as follows:

$$\begin{aligned} X &= S_{remote_{phy}} + S_{comm_{phy}} - S_{wasted_{phy}}, \\ Y &= \begin{cases} X & \text{if } X > 0, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{6.14}$$

Furthermore, based on the S^{type}_{remote} value, the received remote slack is calculated as follows:

$$Z = \begin{cases} f_{p\_v}(Y - 2 * wctt, A, tile) - wcet_b & \text{if } S^{type}_{remote} = StaDyn, \\ f_{p\_v}(Y, A, tile) & \text{otherwise.} \end{cases}$$

$$S_{remote_{vir}} = \begin{cases} Z & \text{if } Z > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{6.15}$$

If the S^{type}_{remote} is *StaDyn*, then remote slack contains a non-zero reference for static slack calculation. This means that, to be conservative, the current task iteration should finish until this reference in time. At worst case, finishing would take the communication cost and $wcet_b$ for the current task iteration. The communication cost equals to *2*wctt*, because we need to count the cost of

two synchronizations, the one after $T_{a,i}$ finished and the one after $T_{b,j}$ finishes. Once we obtain the remote slack, we perform the time-domain translation. If the remote slack is equal to zero, it means that all of it is lost.

Next, the available slack for $T_{b,j}$ ($S_{task_{vir}}$) is computed as a max of the remote slack and tile slack:

$$S_{task_{vir}} = \max_{tid=1}^{N\,tasks}(S^{tid}_{remote_{vir}}, S_{tile_{vir}}), \tag{6.16}$$

where $S^{tid}_{remote_{vir}}$ is the remote slack for each remote predecessor or successor task (*tid*). The $S_{tile_{vir}}$ is the slack generated by the tasks executed on the local tile. In Equation 6.16, we achieve accurate and conservative calculation of the slack, even when we apply max instead of min function. To explain such a choice, we consider a case, which multiple tasks are mapped on the same tile and and all of them receive remote slack values. Lets assume, we do not employ Equation 6.16 while we schedule the task with the highest value of remote slack. Therefore, even if the task iteration finishes with *wcet*, the remote slack of the task is distributed to the other tasks on the tile. As a result, max function in Equation 6.16 preserves the conservative calculation of the slack.

Existing work typically computes and allocates slack per task, and not per application. However, we differentiate two implementations of Equation 6.16, depending on the value of N, i.e., either the number of remote tasks for $T_{b,j}$ or the total number of remote tasks in the application. In Section 7.4 we investigate the quantitative difference between these two approaches.

Finally, a slack policy (f_{slack}), may be applied to allocate slack for $T_{b,j}$, and potentially scale its operating point:

$$S_{allocated_{vir}} = f_{slack}(S_{task_{vir}}) \tag{6.17}$$

If not all slack is allocated, the remained tile slack before starting the scheduled task should be updated, as follows:

$$S_{tile_{vir}} = S_{tile_{vir}} - S_{allocated_{vir}} \tag{6.18}$$

At instance ⑦, task $T_{b,j}$ completes its execution and updates the rc-counter of the FIFO (see Figure 2.4). During the rc counter update, the consumer task sends back to the producer task its S_{remote} and t_{NoCout} (both values are recomputed when $T_{b,j}$ finishes). The exact list of steps are already listed in the 'transmission of S_{remote}' paragraphs. Later, at instance ⑧, the *i+1* iteration of the producer task is started and finishes at instance ⑨.

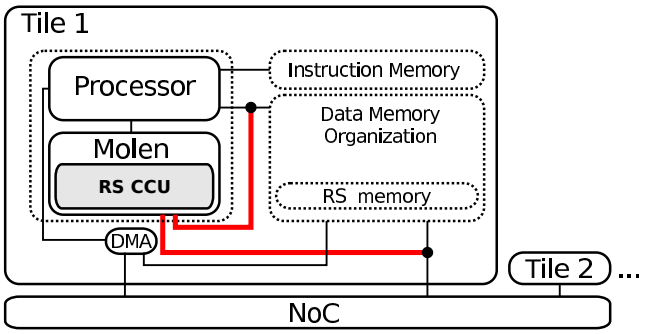


Figure 6.6: CompSoC processor tile augmented with Molen-style RS CCU

Summarizing, we propose a run-time framework that: 1. accurately computes and distributes static and dynamic slack between tasks mapped on different tiles of an MPSoC, and 2. enables the utilization of any management policy that can allocate slack and scale the tile operating point to reduce energy consumption.

6.5 System Implementation

In this section we describe the new software and hardware components added to an existing RTOS and MPSoC.

The tiled CompSoC platform [37] is the template for our implementation. Each tile execute the CompOSe RTOS [66]. To implement slack distribution, we augment the CompSoC tile with a Custom Computing Unit (CCU) using Molen-style processor–coprocessor design [111], more specifically, the coprocessor microarchitecture of the TISC CCU (see Chapter 4). We refer to the software part of our framework as Remote Slack (RS) library. Respectively, we call our CCU - RS CCU.

In Figure 6.6, we present the architecture of a CompSoC tile augmented with Molen-style coprocessor. This CCU accesses the NoC and data memory buses. The RS CCU receives the remote slack from the NoC bus and it registers the time of its arrival in t_{NoCin}. The RS CCU stores the remote slack, the t_{NoCout}, and the communication slack for each FIFO in its internal memory, which is part of the data memory organization of the tile, hence it is accessible by the tile processor through the data memory bus.

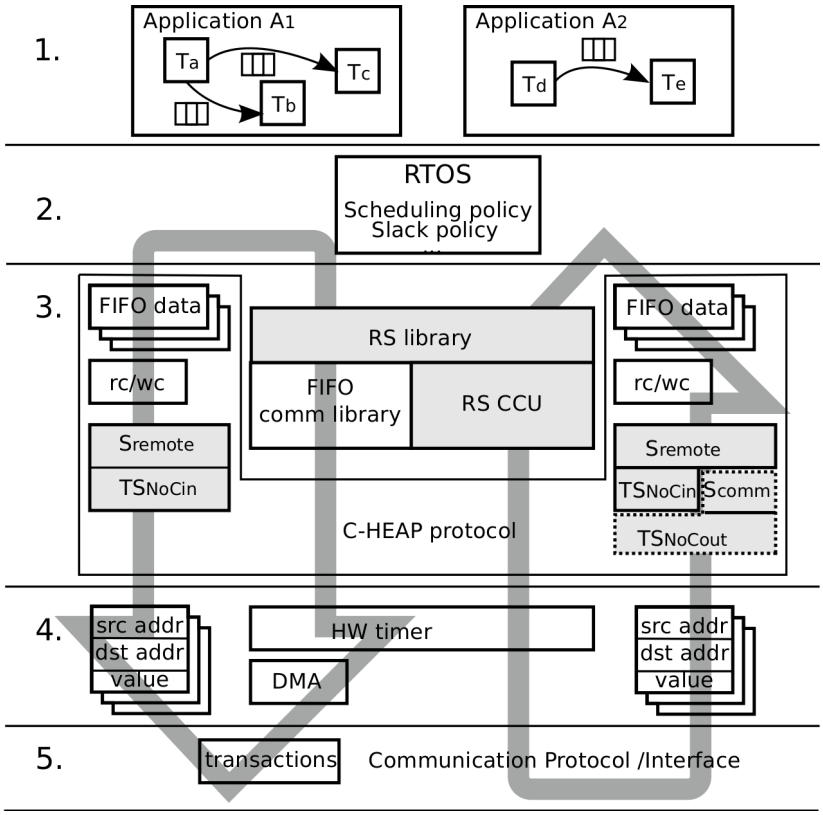


Figure 6.7: RS library and RS CCU integration to the CompOSe RTOS

6.5.1 Design Tradeoffs

We consider three abstraction levels for the partitioning of the RS library and the RS CCU: at application, at task, and at FIFO level. The most intuitive implementation is to associate the RS CCU at application level, because the slack itself is per application. At this abstraction level, the RS CCU should compute the $S_{tile_{vir}}$ for each user application. If a task communicates with multiple remote tasks, then the RS CCU should be able to register the t_{NoCout}, and process multiple $S_{remote_{phy}}$. By processing, we understand that the RS CCU should implement Equation 6.16, i.e., compare the $S_{remote_{phy}}$ and $S_{tile_{vir}}$ values. If the $S_{remote_{phy}}$ is the same type as $S_{tile_{vir}}$, e.g., static or dynamic, or one of them is equal to zero, then the comparison can be done by a standard hardware comparator. Otherwise, if the $S_{tile_{vir}}$ and $S_{remote_{phy}}$ types are different, then the RS CCU should be able to convert them to the same slack type. The conver-

sion is possible if the RS CCU has access to Equation 6.15. Such an access can be granted by either implementing Equation 6.15 in hardware or having an extra synchronization between the processor and the RS CCU to exchange the parameters in Equation 6.15. Therefore, we consider this approach as inappropriate for our implementation.

At the second abstraction level, the RS CCU is associated with a task. Again, there can be a task which communicates with multiple remote tasks. Similarly to the previous abstraction level, the RS CCU needs to have access to Equation 6.15. Therefore, we consider this approach also as inappropriate for our implementation.

At the third abstraction level, the RS CCU is associated with a FIFO. Because of the CSDF application model that we employ, there is always one producer and one consumer task associated with a FIFO. Therefore, the RS CCU does not need to compare explicitly the $S_{remote_{phy}}$ values. The RS CCU can be kept as simple as possible, preserving the $S_{remote_{phy}}$, registering the t_{NoCout}, and computing the S_{comm}. The S_{comm} is computed by Equation 6.12. The rest of the computations can be left in software, as a part of the RS library. This is the design choice that we consider for the software/hardware partitioning.

In Figure 6.7, we present the integration of the the RS library and the RS CCU to the CompSoC platform. The shaded blocks represent our contribution to the existing platform. The two large arrows illustrate the transmission and reception of FIFO tokens. If a task consumes tokens from a FIFO, then it receives the data and the *wc*, and it is responsible with updating the *rc*. If a task produces tokens into a FIFO, then it receives the *rc* and it sends data, and updates the *wc*. We augment the synchronization information, namely *rc* and *wc*, with two extra fields: slack and timestamp (t_{NoCin}). In this way, remote slack is distributed in both directions: from the producer task to the consumer task and vice versa.

Figure 6.8 details the sequence of steps involved in slack distribution. First, after a task iteration finishes the RTOS invokes the RS library to update the $S_{remote_{vir}}$, as represented with ①. Internally, the RS library calls the f_{v_p} function that uses the application scheduling information (at RTOS level) to calculate $S_{remote_{phy}}$. This is represented with ② in Figure 6.8. After that, at instance ③, the RTOS invokes the FIFO communication library to transfer the tokens (if the task is a producer), the synchronization, and the slack to remote tiles. Hence first, at instance ④, the FIFO communication library transfers, via the DMA, to the remote tile, either the DATA+*wc* (producer task), or *rc* (consumer task). At instance ⑤, $S_{remote_{vir}}$ and t_{NoCin} are transferred via the

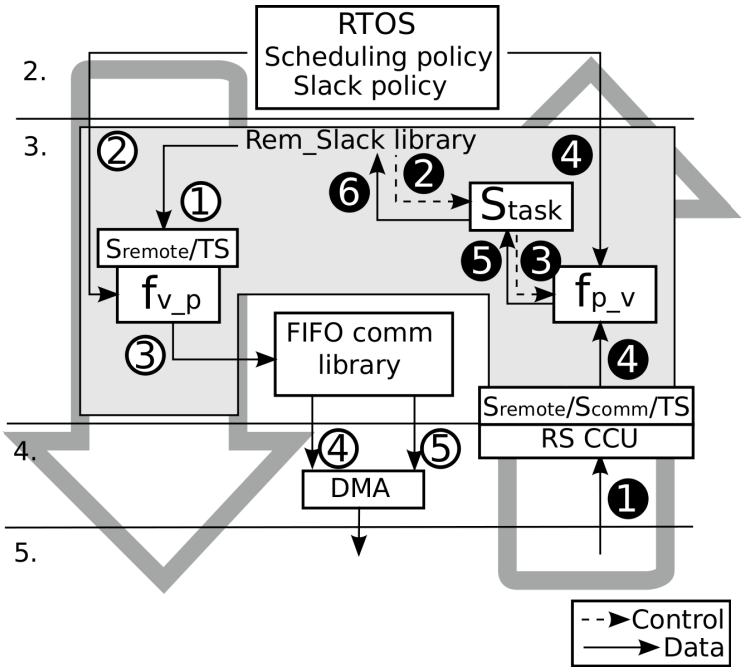


Figure 6.8: RS library and RS CCU integration to the CompOSe RTOS: detailed view

DMA.

On a remote tile, for a given FIFO, at instance ❶, synchronization and slack information is received. The RS CCU stores the $S_{remote_{phy}}$, registers its arrival time, t_{NoCout}, and computes the S_{comm}. Later, whenever a task finishes, a new one should be scheduled, for example at instance ❷. At this moment, independently whether instance ❶ occurred, the RTOS invokes the RS library to calculate the S_{task} by employing Equation 6.16. At instance ❸, the RS library invokes f_{p_v}. Internally, as labeled with ❹, the f_{p_v} employs the received remote slack and the scheduling policy information to translate the remote slack from the tile physical-time to the application virtual-time. At instances ❺ and ❻, the translated remote slack value is returned back to the RS library and the RTOS, respectively. The instances from ❸ to ❺ are repeated as many times as the task has predessesors and successors (see Equation 6.16).

6.6 Conclusions

In this chapter, we proposed a framework for slack computation, allocation, and distribution that transfers the static and dynamic slack information among the tiles in an MPSoC, executing dataflow applications. We send slack altogether with the existing inter-task synchronization. Moreover, we transferred the slack in both directions - from the producer to the consumer task and vice versa. We introduced RS CCU running in a processor–coprocessor *parallel non-blocking* execution model for transferring the slack between the tiles.

7 Experimental Results

In this chapter, we present our experimental setup and obtained results for each of the previously introduced coprocessors (CCUs). More precisely, we provide a comparison with the related state of the art projects in terms of one or multiple of the following criteria: performance–portability and performance–scalability trade-offs, average system speedup, SW/HW costs, operating processor frequency levels, and energy consumption. For our experiments, we employ synthetic and real applications.

We evaluate the Thread Interrupt State Controller (see Chapter 4) with the help of single-threaded and multithreaded benchmarks. For real applications, the suggested average system speedups are between 1.2 and 19.6. Based on single-threaded synthetic benchmark, we achieve average speedup between 8.5 and 129. For multithreaded synthetic benchmark, the achieved average speedup is between 1.3 and 7.3.

We also evaluate the Hardware Task Status Manager (see Chapter 5) through two types of applications (synthetic and real) running in software. With synthetic applications, the results indicate a WCET reduction of the RTOS between 1.1 and 3.0 times. With the real applications, i.e., JPEG and H.264 decoders, the WCET of the RTOS is reduced by 1.3 and 1.6 times, respectively. The overall system performance gain varies from 0.9% to 13.3%.

We evaluate the concept of the Remote Slack distribution framework (see Chapter 6) by comparing the processor frequency levels, and the energy consumption in four cases: a no-slack management, an existing intra-tile slack management, and two implementations of our inter-tile slack management. The experimental results suggest that our inter-tile technique reduces the average processors frequency down to 56% and the energy consumption down to

53%. In very few worse cases, our technique introduce negligible increase of the average processors frequency with 0.3%, and the energy consumption with 0.03%. Our proposal introduces only trivial software cost of up to 4% over the application execution time and negligible additional chip utilization of 0.002% from the considered FPGA board (XC6VLX240T).

7.1 Introduction

In this section, we provide a brief overview of the performed experiments for each one of the CCU exercised in the current chapter. We experiment with the CCUs as follows:

Thread Interrupt State Controller: We consider various experimental scenarios to quantify the performance gains due to our approach. Experimental results with real applications suggest average system speedups between 1.2 and 19.6 compared to pure software implementations. Based on single-threaded synthetic benchmark, we achieve an average speedup between 8.5 and 129 compared to pure software implementation. For multithreaded benchmark, the average speedup is between 1.3 and 7.3. Our proposal is compared qualitatively and quantitatively with the current state of the art research projects. The results of the comparison suggest that our approach allows better performance-portability and performance-flexibility trade-off characteristics than the most recent related proposals.

Hardware Task Status Manager: To prove our concept we experiment with two synthetic applications and two real applications, i.e., JPEG and H.264, respectively. All applications are executed on a CompSoC platform instance [37], implemented on an FPGA. We investigate the RTOS cost reduction by varying the scheduling policies, because they are the ones that directly affect the number of the tasks status computations. Hence, each one of the synthetic applications is specifically designed to explore one of the two types of scheduling policies - static, e.g., time-division multiplexing, or dynamic, e.g., Round-Robin. The experimental results on synthetic benchmarks suggest a reduction in the WCET of the RTOS, compared to a pure software implementation, between 1.1 and 1.8 times for static scheduling policies. For dynamic scheduling policies, this WCET reduction is between 1.1 and 3.0 times. With the real applications, i.e., an JPEG decoder and a H.264, the reduction in the WCET of the RTOS are 1.3 and 1.6 times, respectively. Moreover, we estimate that the overall performance gain varies from 0.9% to 13.3%, when the cost of the RTOS is reduced.

Remote Slack Distribution: Our experiments are based on a data-flow implementation of the H.264 decoder. We compare a no-slack management technique, an intra-tile management technique [67] and two implementations of our inter-tile technique. In each of the cases, we employ a simple slack policy that extract as much as possible of the observed slack without jeopardizing the end-to-end application performance. As evaluation criteria, we employ the frequency levels, the consumed energy, and the introduced costs of our technique for each one of the tiles. Our experiments suggest that the average frequency level reduction of the tasks in our technique compared to existing intra-tile technique [67] is down to 56.8%. Furthermore, the average energy consumption reduction is down to 53.3%. In very few worse cases, our technique increases the average processors frequency levels with 0.3%, and the energy consumption with 0.03%. The introduced software cost in terms of extra clock cycles varies from 1% to 4% of the application execution time. The introduced hardware cost in terms of chip utilization is trivial - 0.002% from the considered FPGA board (XC6VLX240T).

In the sections to follow, we reveal more details for each one of the performed experiments.

7.2 Thread Interrupt State Controller Evaluation

We have developed an experimental platform based on the Xilinx Virtex II XC2VP30 FPGA chip using the XUPV2P Prototyping Board. In the following, we briefly discuss our experimental setup.

GPP Core: The GPP, used to obtain the experiments, has a traditional RISC architecture. It is based on the MIPS R3000 [83] implemented as a soft-core on the FPGA chip.

RTOS: The RTOS running on the GPP is a light-weight version of [83], which we named ρRTOS. It has a memory footprint of less than 20 KBytes. The RTOS has support of multithreading, memory management, synchronization, and round-robin (RR) scheduling.

To satisfy our RTOS requirements, we made the following modifications to [83]: 1. improve the ISR that serves the hardware timer - the XREG#0 value is pushed and popped to/from the program stack; 2. a new ISR is designed, managing thread semaphores; 3. a Molen programming library is used to emulate the Molen instructions.

Compiler Support: We do not address any static scheduling techniques by

the Compiler. We create the MIPS compatible binary code, by using a standard version of the GCC compiler. Instead of modifying the compiler to consider Molen polymorphic instructions, we create a Molen programming library which emulates them. The new opcodes are generated with a custom tool, which modifies the execution binary file. In case a Molen library call is issued, the tool substitutes the used instruction with the proper opcode. In our experimental implementation, the FIDs are statically generated by the system programmer, but ultimately they should be managed by an appropriately designed compiler.

7.2.1 Evaluation Methodology

We run several real streaming applications and we design our own synthetic benchmark suite to evaluate the impact of the RTOS and the TISC on the thread and task performance. Each one of the experiments includes: thread creation, thread termination, interrupt handling, and an RTOS scheduling policy algorithm. We define a scenario to be a set of application threads, where each thread has a certain number of software and hardware tasks. We evaluate the system performance in scenario k by computing the average speedup, denoted as $S_{av}(k)$:

$$S_{av}(k) = \sum_{i=1}^{n\_exp} \frac{T_{SWe}(i)}{T_{HWe}(i) * n\_exp} \tag{7.1}$$

where $T_{SWe}(i)$ corresponds to the computation time of the pure software implementation and $T_{HWe}(i)$ is the computation time in each one of the other scenarios, containing reconfigurable hardware executions. The *n_exp* variable represents the number of performed simulations in each one of the scenarios.

We employ the following nomenclature to structure the experimental scenario names (S_N): S_N= $\{S_N$,(DL)$\}$ where D=$\{1,2,3,4\}$ and L=$\{$C,SW,T$\}$. S_N is composed of multiple (DL) pairs, where *L* are interpreted as follows: *C* corresponds to a task executed on a CCU, *SW* is a software task and *T* indicates the total number of threads. For example: 4C1T denotes 4 CCUs running in parallel in 1 thread; 1C3SW4T means 4 threads and 4 tasks, one task executes on a CCU, the other three tasks in software. Note, that in the 4C1T scenario, the TISC Unit shall assert an interrupt to the GPP only after all four hardware tasks (CCUs) are finished.

The streaming applications package includes three popular real applications: *Floyd-Warshall* algorithm, *Conjugate Gradient* and *MJPEG* Encoder. We

Table 7.1: Evaluation results with Floyd-Warshall algorithm, measured in clock cycles

Application type	Scenarios						
	1 thread		4 threads				
	SW 1SW1T	HW 1C1T	SW 4SW4T	SW + Reconfig. HW 1C3SW4T	2C2SW4T	3C1SW4T	HW 4C4T
FL25	47 650	46 520	120 483	119 382	116 981	116 497	112 079
FL400	158 082	50 830	568 505	454 976	346 849	240 635	126 787
FL1600	914 349	65 470	3 591 126	2 739 194	1 886 204	1 032 139	187 630
S_{av}(Var. Floyd)	**1**	**6.1**	**1**	**1.2**	**1.5**	**2.3**	**8.2**

choose them, because they present three different application domains: graph analysis, linear equation systems and multimedia. The results, in terms of clock cycles, are presented in Table 7.1. Note, that in all scenario, which have CCU invocations, the CCU design is identical. In the scenarios with four tasks, e.g., 4C4T and 4SW1T, each task is processing a size of dataset equal to the data size used in 1C1T/1SW1T. Thus, the four tasks in 4C4T, process four times more data than the task in 1C1T.

Real Benchmarks: The *Floyd-Warshall* algorithm (FL) finds all shortest paths in a weighted graph. In Table 7.1, it is marked as FL25, FL400, and FL1600, where the numbers indicate the count of nodes in the graph. When working with small data-sets, e.g., FL25, the execution time of the system in all scenarios is almost equal to the pure software execution time - 1SW1T and 4SW4T. The reason for such a behaviour is caused by the RTOS cost in terms of thread_creation and scheduling routines. For larger datasets, see FL400 and FL1600, the execution time of 1C1T/4C4T remains relatively constant compared to pure software. The experiments composed of software functions and hardware CCU threads such as 2C2SW4T, mimics the behaviour of 4SW4T due to the software tasks. The *Floyd-Warshall* CCU is designed according to the implementation details for a dedicated *Floyd-Warshal* coprocessor, presented in [20].

The second experimental application is based on the *Conjugate Gradient* (CG) benchmark, part of the *NAS* Parallel Benchmark Suite [15]. The most computation intensive parts of this application are the floating-point arithmetic operations. The results suggest that even with a small number of trails - 14, running such applications on a simple RISC core without floating-point unit and using software math library only, requires a tremendous amount of time. This is the reason why we do not perform any experiments with larger datasets and we do not run more than one thread. The purpose of this benchmark is to indi-

Table 7.2: Evaluation results with CG and MJPEG applications, measured in clock cycles

Application type	Scenarios	
	Pure SW 1SW1T	Reconfig. HW 1C1T
CG14	72 251 488	3 684 817
MJPEG64	4 030 275	1 269 830
S_{av}(CG)	**1**	**19.6**
S_{av}(MJPEG64)	**1**	**3.2**

cate the potential portability of our ideas to application domains traditionally positioned outside of the embedded world. On the other side, we demonstrate that we can port such complex applications in embedded systems, that have not been considered before. The experimental results, reported in Table 7.2, suggest speedup of more than 19 times compared to the pure software implementation. The experiments are executed using a dedicated memory hierarchy which feeds the CCUs with data efficiently. The description of such hierarchy is outside the scope of this dissertation. The reason of such high speedup is the fact that more than 95% of the application computation time is spent in a simple function. The function performs multiplication on a floating-point numbers. More implementation details of the *Conjugate Gradient* CCU can be found in [84].

The most time intensive function of the *MJPEG* Encoder is the Discrete Cosine Transformation (DCT), which we implemented in a CCU. The experimental results, reported in Table 7.2, suggest that the overall application execution time drops more than three times for a tiny video stream with 64 pixels (8×8) per frame.

The achieved speedup of the *Conjugate Gradient* and *MJPEG* encoder applications is due to the CCU implementation of the software functions in hardware. In 1SW1T and 1C1T, the strength of our proposal over the baseline Molen design is that the GPP can execute a different thread while one or more CCUs are performing their computations. More specifically, the GPP will be notified only after the predefined set of CCUs have terminated successfully.

Synthetic Benchmarks: Last but not least, we have designed a synthetic benchmark suite, which covers more use-cases than the previously described real applications. In Table 7.3 and Table 7.4, we list the experimental results for two types of synthetic applications, single- and multithreaded, respectively. We use one and multiple tasks per thread for the single-threaded applications and only one task per thread for the multithread applications. We assume

Table 7.3: Evaluation results with a single-threaded synthetic benchmark suite, measured in clock cycles

Dataset	SW tasks are running 10x slower than HW implementation				SW tasks are running 100x slower than HW implementation			
	1SW1T	1C1T	4SW1T	4C1T	1SW1T	1C1T	4SW1T	4C1T
100	25 413	22 165	37 438	22 273	61 422	22 165	134 783	22 273
200	29 413	22 195	53 438	22 303	101 413	22 195	341 708	22 303
500	41 428	22 491	101 438	22 599	277 223	22 491	417 184	22 599
1000	61 422	22 987	134 783	23 095	505 627	22 987	785 440	23 095
2000	150 659	23 912	231 388	24 020	1 001 412	23 912	1 507 519	24 020
4000	227 437	25 389	351 956	25 497	1 960 469	25 389	2 951 504	25 497
8000	412 996	26 283	641 095	26 391	3 833 259	26 283	5 771 489	26 391
10000	505 627	26 991	785 440	27 099	4 843 057	26 991	7 291 585	27 099
12000	598 185	29 515	929 678	29 623	6 101 164	29 515	9 184 147	29 623
S_{av}	**1**	**8.5**	**1**	**13.6**	**1**	**86.4**	**1**	**129**

that these two types of applications are not functionally equivalent. Therefore, they cannot be compared, and we evaluate them separately. In general, a single-threaded application cannot be always transferred into a multithreaded one and vice versa. For each one of the synthetic applications, we investigated two basic scenarios, when: 1. software functions are executed 10 times slower than their corresponding hardware implementations; and 2. software functions are 100 times slower than the corresponding CCUs. The execution time of the CCU, modelled as number of iterations in a single loop, varies from 100 cycles up to 12000 clock cycles. Depending on the hardware speedup (10 times or 100 times), the number of software executions varies from 10×100 up to 10×12000 and from 100×100 up to 100×12000. All synthetic simulations are implemented with four tasks, executed in a variable number of software threads with equal priorities.

In Table 7.3, we present the experimental results for the single-threaded synthetic application. The application has two scenarios with hardware tasks, i.e., 1C1T and 4C1T, and two scenarios with software tasks, i.e., 1SW1T and 4SW1T. Based on the results for 1C1T and 4C1T, we can conclude that the RTOS cost for intra-thread parallelism is minimal, in terms of dozen of cycles for each additional CCU. In scenario 4SW1T, we perform four sequential invocations of the targeted software task/function. This explains the high speedup gains for 4C1T over 4SW1T, which are mainly due to the faster CCU execution compared to its SW equivalent.

In Table 7.4, we present the experimental results for the multithreaded synthetic application. As the results suggest, the execution time of 4C4T is almost constant, even when the dataset size increases. This is due to the overlapping of the CCU execution with the RTOS services. Because we consider synthetic

Table 7.4: Evaluation results with a multithreaded synthetic benchmark suite, measured in clock cycles

Dataset	SW tasks are running 10x slower than HW implementation					SW tasks are running 100x slower than HW implementation				
	4SW4T	1C3SW4T	2C2SW4T	3C1SW4T	4C4T	4SW4T	1C3SW4T	2C2SW4T	3C1SW4T	4C4T
100	197 852	96 532	92 538	92 449	90 951	177 229	155 597	133 891	112 035	90 951
200	167 335	101 382	97 136	93 114	90 940	297 986	241 231	314 000	439 304	90 940
500	133 243	121 646	113 940	97 461	92 469	530 231	425 868	297 342	210 234	92 469
1000	178 239	155 597	133 857	111 874	92 035	990 551	764 621	538 857	306 114	92 035
2000	297 986	234 015	314 000	436 304	93 982	1 893 149	1 122 233	1 323 123	479 348	93 982
4000	448 696	358 575	269 143	176 648	94 625	3 698 131	2 033 432	1 884 584	953 265	94 625
8000	810 119	689 630	312 540	263 322	96 541	7 223 112	3 523 423	2 382 857	1 544 811	96 541
10000	990 551	764 132	538 857	306 037	98 135	9 123 232	4 128 654	3 383 123	1 629 822	98 135
12000	1 170 848	867 546	618 788	354 556	100 112	11 488 934	4 696 858	3 996 828	1 796 858	100 112
S_{av}	**1**	**1.3**	**1.6**	**1.7**	**2.7**	**1**	**1.5**	**1.7**	**2.3**	**7.3**

benchmarks, we are only interested in the overall behaviour of the system. These benchmarks were not created with any functionality in mind. Therefore, we are not in a position to investigate local deviations in the system behaviour, but rather the trends in its performance. The experimental results in Table 7.4 suggest that the higher the number of threads executed, the faster the overall application execution is. Moreover, our approach introduces trivial time costs compared to the huge performance gains it enables.

Based on the target application, the programmer can potentially employ more complex scenarios than the ones presented. One such application might have a large set of threads, where intra- and inter-thread parallelism might be exploited at the same time.

7.2.2 Comparison with Related Work

In Table 7.5, we provide an analytical comparison of our design and two of the most relevant state of the art projects in the domain of heterogeneous systems, ReconOS [58] and Hthreads [79]. We evaluate these projects in terms of performance by measuring the RTOS cost during the synchronization procedures.

We consider four execution scenarios with respect to the location of the synchronizing entities. We inherit them from [58]. With SW→SW, we denote a scenario when two software threads are synchronized to each other. In SW→HW scenario, a software thread performs a call, i.e., synchronizes itself with a hardware kernel running on a dedicated CCU. In HW→SW, a CCU initiates a synchronization to a software thread. In HW→HW, two CCUs are synchronized between each other.

We generate the results in all four scenarios using one user application composed of two types of threads. The first thread type contains a hardware kernel call, i.e., CCU call. The second thread contains the user code, except in scenario HW→HW, where the user thread is substituted by a *delegate* thread. According to the ReconOS terminology, a *delegate* thread is a software thread containing a CCU call. Therefore, all communication and synchronization procedures in scenarios SW→HW, HW→SW and HW→HW, implicitly include the *delegate* to *user* thread synchronization cost (SW→SW). In HW→HW scenario, we execute two CCUs sequentially, where each one of the CCUs is assigned to a different *delegate* thread. Note, that we verify the system scalability by extending two of the scenarios SW→HW and HW→SW with arbitrary number of CCUs, denoted as *N*.

In column 2 of Table 7.5, we present analytically the ReconOS semaphore

Table 7.5: Analytical comparison of the RTOS semaphores

Scenarios:	ReconOS	Hthreads	This work
1*SW → 1*SW	Sem_post_{user}+Sem_pend_{user}	$hthread\_mutex$+$hthread\_add$	Sem_post_{user}+Sem_pend_{user}
1*SW → N*HW	N*(Sem_post_{user}+$Sem\_pend_{delegate}$+L)	N*($hthread\_mutex$+$hthread\_add$)	N*L
N*HW → 1*SW	N*(ISR+Sem_post_{ISR}+$Sem\_post_{delegate}$+ Sem_pend_{user})	1*SW → N*HW	$Barrier$+ISR+ Sem_post_{ISR}+Sem_pend_{user}
1*HW → 1*HW sequential	HW→SW + SW→SW + SW→HW	1*SW→1*SW	2*(SW→HW+$Barrier$)+HW→SW

costs. The 1*SW→1*SW synchronization is achieved by posting and pending operations on a semaphore by the source and destination thread, marked as Sem_post_{user}. In the 1*SW→N*HW scenario, the same 1*SW→1*SW cost is available with the difference that one of the user threads is substituted by a *delegate* thread, $Sem\_post_{delegate}$. With L, we define the time period necessary for transferring the CCU parameters to the XREGs. In the SW→N*HW scenario, the RTOS cost also includes the Interrupt Service Routine (ISR) cost. The cost in the last scenario, 1*HW→1*HW, is a combination of all previous scenarios. The ReconOS designers observe that the cost in 1*HW→1*HW scenario is too high for some streaming applications which have directly communicating CCUs. Therefore, the designers employ a dedicated hardware FIFOs between the CCUs to minimize this cost. This concept is completely compatible with our prototyping platform and can not be the source of the difference between our approach and the one from [58]. Therefore, we do not consider it as a target of further experiments.

In column 3 of Table 7.5, we study the cost of the Hthreads platform. Because of the fact that the RTOS scheduler is implemented entirely in hardware, the cost among all scenarios is due to *hthread_mutex* and *hthread_add* functions, used for synchronization purposes. In scenarios 1*SW→N*HW and N*HW→1*SW, the number of Hthreads function calls grows with the number of the communicating tasks. For a large number of threads, we expect the Hthreads synchronisation cost to be much higher than the results reported in [79], because a single bus is employed to connect all CCUs and hardware RTOS blocks. Although an evaluation study of the bus cost was not provided, we expect the system scalability of [79] to be poor beyond a certain threshold.

For the same number of scenarios, in column 4 of Table 7.5, we present our synchronization cost. In 1*SW→1*SW scenario, our design has the same performance and functionality as ReconOS. In 1*SW→N*HW, the only delays in our design are due to the CCU parameter transfers. Therefore, we expect our design to be faster than the ReconOS project. Compared to N*HW→1*SW, we substitute the delay of one of the software semaphores with the delay of the "barrier" instruction, which is equal to 4 cycles extra. Note that because of the "barrier" instruction, the software cost remains constant, independent of the number of CCUs, denoted as N. In the HW→HW scenario, similarly to ReconOS, the software cost is equal to the sum of all costs for the previous scenarios.

In Table 7.6, we present the experimental results of our comparison. All scenarios are generated using two threads only. All results, reported in Table 7.6

Table 7.6: Experimental performance comparison of OS semaphores, measured in GPP clock cycles

Semaphore Synchronization	ReconOS		Hthreads	This work
	eCos /PPC /PThreads	Our RTOS /MIPS		
SW → SW	305	332	36+hthread_add	332
SW → HW	528	505	36+hthread_add	32
HW → SW	908	814	36+hthread_add	198
HW → HW (2 sequential CCUs)	1114	1007	36+hthread_add	262

are obtained with synthetic Molen-style CCUs, where the latter are designed to generate a predetermined delay of 1000 GPP clock cycles. An immediate comparison between our design and the ReconOS project is not possible, because the latter has primary software implementation on a completely different platform, programming model and RTOS. The original ReconOS was implemented on *eCos* RTOS [5] using Power PC as a processor and POSIX PThreads. Therefore, in order to provide a fair comparison between the two approaches, we decided to reimplement the ReconOS synchronization concept on our platform. In column 2 of Table 7.6, we present the originally reported results in [58] and in column 3, the results from our redesign of ReconOS. Apparently, the results from column 2 and 3 in Table 7.6 match closely, which provides us with confidence for the correctness of our further comparisons. In Table 7.6, we also provide the experimental results obtained for our design. For the SW→SW synchronization, our proposal has the same cost as the ReconOS, because the software user threads are communicating through software semaphores. As suggested by the analytical study, for the other scenarios, we can achieve the same functionality as ReconOS for $\frac{1}{16}$ up to $\frac{1}{5}$ of the time, confirmed by the figures in column 2 and column 5 in Table 7.6.

In column 4 of Table 7.6, we present the Hthreads results. We decide not to redesign any part of it, because the Hthreads has primary hardware implementation. The results in [13] reveal that the semaphores cost remains constant among all SW to HW thread synchronizations, due to the hardware implementation. The authors achieve thread semaphore synchronization by using Hardware blocks called *Mutexes* and *Thread Manager*. The cost of 36 cycles for the *hthread_mutex* is mainly due to the cost of the multi-master bus that designers employ. We do not include any numbers for the *hthread_add* function used for communication between the *Mutexes* and the *Thread Manager*, because the authors do not evaluate this cost. Due to the hardware approach in Hthreads, we expect the cost of the *hthread_add* function to be in terms of dozens of clock cycles. Compared to Hthreads, our approach performs much

Table 7.7: Qualitative comparison of the three approaches

	ReconOS	Hthreads	This work
Portability	HIGH	MEDIUM	HIGH
Flexibility	MEDIUM	LOW	HIGH
Performance	LOW	HIGH	HIGH

slower in the SW→SW scenario. In the SW→HW scenario, our performance is comparable to Hthreads. In the HW→SW and the HW→HW scenarios, the overall performance of our design is much closer to the Hthreads than to the ReconOS.

Based on our analysis and the obtained experimental results in Table 7.7, we provide a qualitative comparison of the three different designs considered. We compare them in terms of *portability*, *flexibility*, and *performance*, all defined in Section 6.2. With our reimplementation, we have demonstrated that the ReconOS is easily portable. We rate ReconOS as medium flexible, because their design is limited to transferring Operating System services of a single-threaded RTOS into hardware. The overall performance of the ReconOS design is lower than our proposal.

In Hthreads, the portability is at medium level, but their flexibility is low, because the system does not provide an efficient mechanism to port additional RTOS specific services in hardware. Their performance is ranked high, because the selected RTOS services are entirely implemented in hardware. Due to our architectural approach and the properties of the Molen architecture, we are able to achieve high system portability. The flexibility of our approach is also ranked high since we provide a platform capable to port any RTOS service or user application into hardware. As the experimental results suggest, our performance is high and it is in the same order as the highest performance demonstrated by the Hthreads project.

7.3 Hardware Task-Status Manager Evaluation

In this section, we measure and analyze the RTOS actual-case and worst-case execution time with synthetic and real applications with and without the HWTSM CCU.

We perform all experiments with a tiled CompSoC platform employing Xilinx Microblaze as a core inside of a processor tile. The design is synthesized using Xilinx Platform Studio 12.2 and verified on a Xilinx Virtex 5

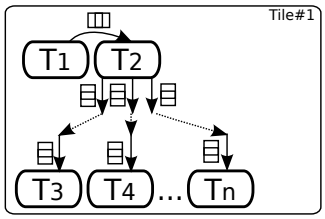


Figure 7.1: Synthetic application with StS policy

ML510 (XC5VFX130T) Evaluation Board. We employ data-flow, i.e., streaming applications to evaluate our system. The exercised streaming applications include two synthetic applications and two popular applications in the embedded systems domain – an H.264/AVC decoder [80] and an JPEG decoder. We experiment with synthetic applications mapped on one processor tile only and with real applications ona two processor tiles. However, the obtained results can be generalized for any arbitrary number of tiles. The reason for such a generalization is that the execution time of the RTOS in one processor tile depends only in the number of tasks (and the number of their FIFOs) executed locally on that tile. Furthermore, the execution time of the RTOS does not depend on the number of processor tiles to which these local tasks communicate with.

The rest of the section, we organize as follows: we first present the analytical model and the evaluation results from the synthetic applications, followed by the real applications and the overall system performance improvement. Then, we provide a qualitative comparison of the HWTSM using different processor–coprocessor execution models. We conclude the section, by evaluating the hardware costs of the HWTSM.

We develop two synthetic applications. We use each one of the applications to evaluate our proposal with two basic types of intra-application scheduling policies: Static Scheduling (StS) and Dynamic Scheduling (DyS). We explicitly investigate different scheduling policy types, because they are the ones which dictate the number of HWTSM calls. We generate our synthetic benchmark by varying the number of FIFOs per task and the total number of tasks, both in the range from 1 to 10. For each one of the synthetic and real applications, we investigate the WCET of the RTOS with and without the HWTSM included. In the synthetic applications setup, all FIFOs are configured to accommodate up to two tokens.

In the StS policies, such as time-division multiplexing (TDM), the next scheduled task is always the next one in the list of tasks. In case the task is not ready, the corresponding application time slot is left idle. In Figure 7.1, we present

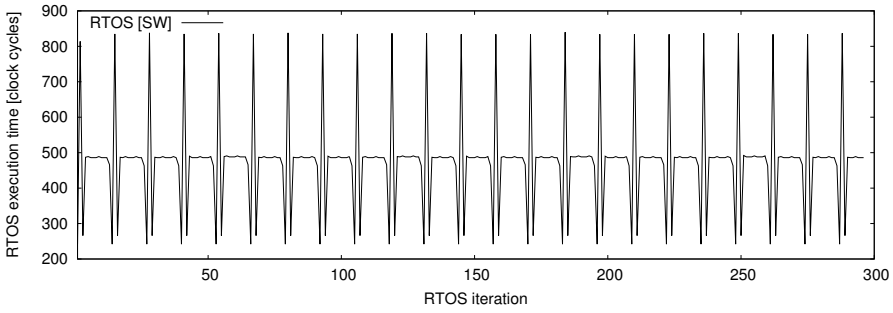


Figure 7.2: RTOS profiling with StS for 10 tasks

an inter-task communication pattern of the synthetic application that we use in a combination with StS policy. Let us assume that the currently executed task is T_1 and the next one to be scheduled is T_2. As a result, during the RTOS time slot, the StS policy only checks FIFOs associated with T_2. That's why we create the StS scenarios by scaling the number of FIFOs associated with T_2.

The WCET of the RTOS with StS policy without using HWTSM is $WCET_{sts\_sw}$:

$$WCET_{sts\_sw} = T(n_f) + Const_{sched} + Const_{cntx} \tag{7.2}$$

where $T(n_f)$ is the time to read the task *rc* and *wc* for each of the FIFOs (n_f) associated with a task and compute the status of the task. The $Const_{sched}$ is the constant time for the scheduling policy. The $Const_{cntx}$ is the context switching time. Therefore, $WCET_{sts\_sw}$ is a linear function, depending only on the number of FIFOs (n_f).

The WCET of the RTOS using the HWTSM and the StS policy is $WCET_{sts\_hw}$:

$$WCET_{sts\_hw} = Const'_{sched} + Const_{cntx} \tag{7.3}$$

where $Const'_{sched}$ is the constant time necessary to fetch the task-status from the HWTSM. Since the FIFO checking and task-status computing are performed in hardware, overlapped with the execution of the software, then the RTOS execution time is always constant.

In Figure 7.2, we present the execution time of the pure software RTOS for an application that has 10 tasks and 9 FIFOs in total. The deviations in the RTOS actual case execution time are due to the number of FIFOs in the task to be scheduled. A task with low number of FIFOs (such as T_1) requires less time to be determined whether it is eligible for execution than a task with high

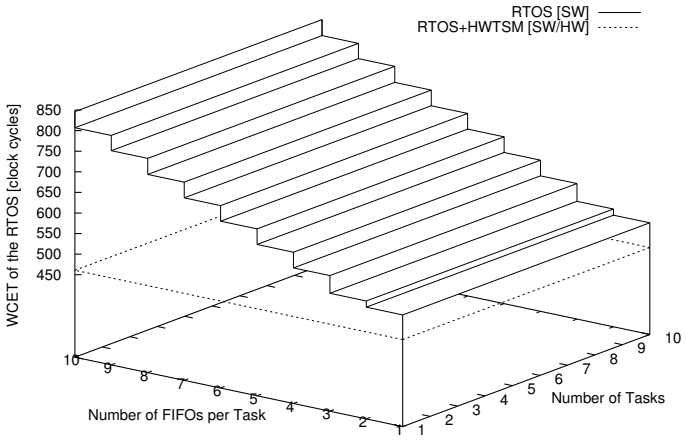


Figure 7.3: WCET of the RTOS with StS

number of FIFOs (such as T_2). In Figure 7.2, the RTOS actual case execution times mostly varies around 500 clock cycles, because the StS policy suggest to schedule a task with low number of FIFOs, which is not eligible for execution. Therefore, the RTOS StS policy loads the idle task. The WCET of the RTOS is equal to 846 clock cycles for the synthic application presented in Figure 7.1. Our analysis suggest that the WCET of the RTOS is experienced when T_2 is scheduled for execution, because T_2 is the task with the highest number of FIFOs in the application. As explained in [9], the composability of the system is ensured by leaving the processor idle for the time difference between the real execution time of the RTOS and its WCET. In Figure 7.2, the shortest execution time of the RTOS is equal to 266 clock cycles. Therefore, the idle period is equal to 620 clock cycles. Our goal is to reduce the WCET of the RTOS, as well as RTOS execution time variations, such that the idle period is minimized.

In Figure 7.3, we present the WCET of the RTOS for the StS policy as a function of the number of tasks and FIFOs. The results are obtained with and without the HWTSM for the synthetic application from Figure 7.1. As the analytical model for StS suggests, the variations of the WCET of the RTOS are close to linear, with respect to the number of FIFOs. In Figure 7.4, we present a snapshot from Figure 7.3 to visualize more clearly the dependence of the WCET of the RTOS to the number of FIFOs for an arbitrary number of tasks.

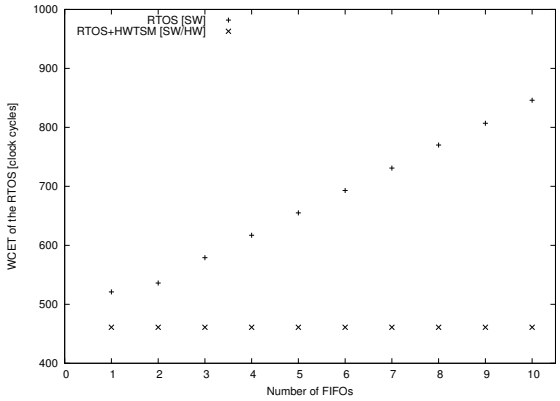


Figure 7.4: WCET of the RTOS with StS for an arbitrary number of tasks

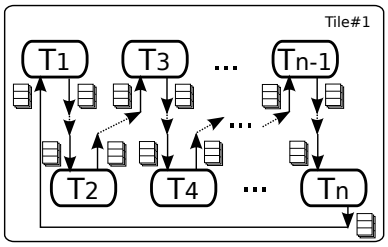


Figure 7.5: Synthetic application with DyS policy

In our experiments, the number of tasks varies from 1 to 10. The non-linear behaviour at two FIFOs is caused by the software implementation, because the processing of the input/output FIFOs is different. As the analytical model for StS suggests, the WCET of the RTOS with HWTSM is always constant, equal to 461 clock cycles. The constant execution time is due to the StS execution profile and the proposed *parallel non-blocking* execution model.

In Figure 7.5, we present the second synthetic application using DyS policy. In our experiments, we use the Round-Robin (RR) scheduling algorithm as a representative of the DyS policies. Contrary to StS, the DyS policies might check multiple tasks until a ready one is found. The WCET of the RTOS for a DyS policy occurs when all tasks belonging to the current application are checked and the next scheduled task is again the currently running one. We construct the synthetic applications for the DyS policy by varying the number of tasks and FIFOs in the application graph.

The WCET of the RTOS with DyS policy for pure software implementation is

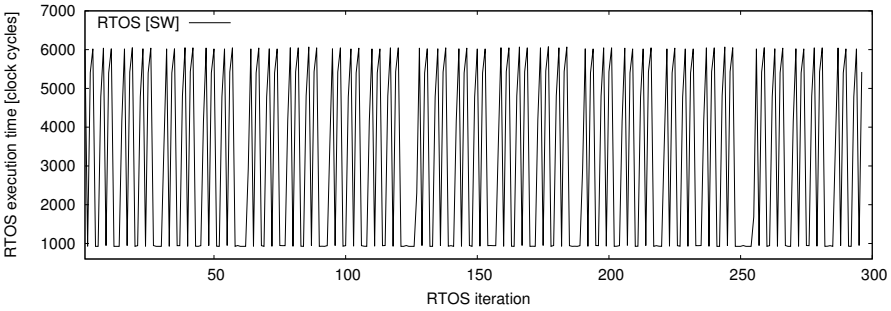


Figure 7.6: RTOS profiling with DyS for 10 tasks and 10 FIFOs per task

$WCET_{dys\_sw}$:

$$WCET_{dys\_sw} = T(n_f, m_t) + Tsched(m_t) + Const_{cntx} \tag{7.4}$$

where $T(n_f, m_t)$ is the time to read the task *rc* and *wc* for each of the FIFOs (n_f) associated with each task (m_t) and compute each task status. The $Tsched(m_t)$ is the scheduling time for the DyS policy algorithm. The $Const_{cntx}$ is the time necessary for context switching.

The WCET of the RTOS employing HWTSM and the DyS policy is $WCET_{dys\_hw}$:

$$WCET_{dys\_hw} = T'sched_{hw}(m_t) + Const_{cntx} \tag{7.5}$$

where $T'sched_{hw}(m_t)$ is time for the employed DyS policy when a HWTSM is used. Since the time for the FIFO management is removed, the deviations of the $WCET_{dys\_hw}$ are only due to $T'sched_{hw}(m_t)$. Therefore, if the DyS policy has linear complexity, the WCET of the RTOS also grows linearly.

In Figure 7.6, similarly to the StS, we present the execution time of the RTOS when a single application of 10 tasks and 10 FIFOs per task is executed. Consequently, there will be a total of 100 FIFOs in the application. As it is visible in Figure 7.6, the RTOS execution time varies from approximately 1000 up to 6000 cycles among different RTOS invocations. The deviations in the actual case execution time of the RTOS are due to the checked number of tasks until an eligible for execution is found.

In Figure 7.7, we present the WCET of the RTOS as a function of the number of tasks and FIFOs. The experimental results are obtained for six different scenarios with the number of FIFOs per task equal to 2, 4, 6, 8, and 10. As the analytical study for DyS suggests, the WCET of the RTOS for the pure software implementation grows in two dimensions defined by the number of tasks

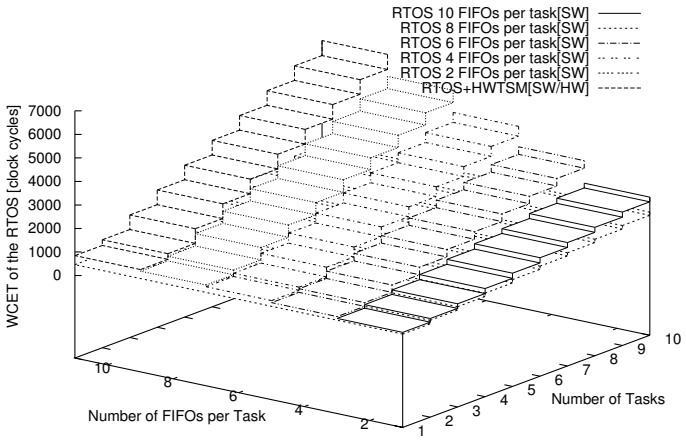


Figure 7.7: WCET of the RTOS with DyS

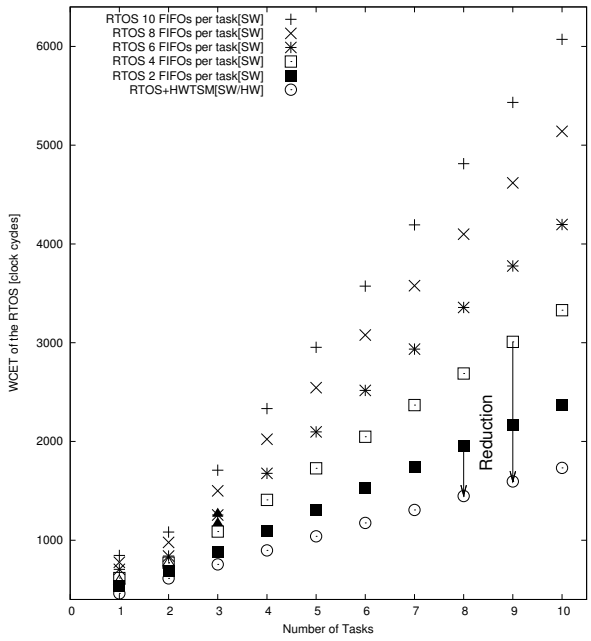


Figure 7.8: WCET of the RTOS with DyS (in detail)

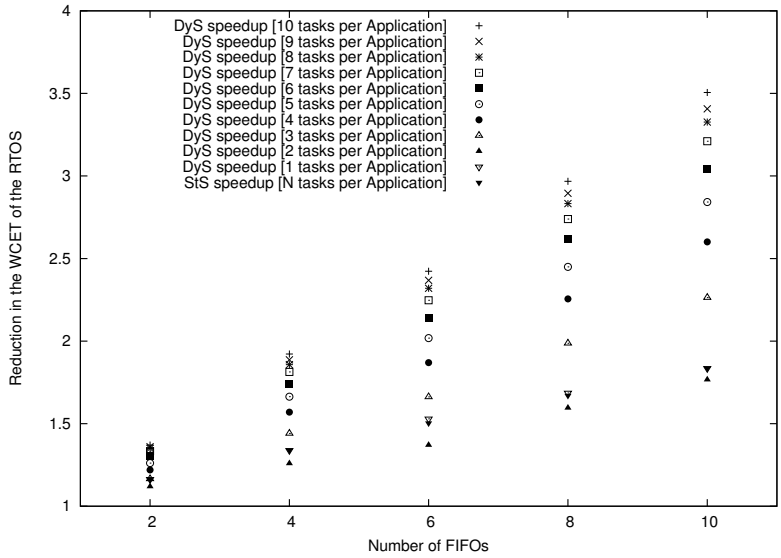


Figure 7.9: Reduction in the WCET of the RTOS against pure software implementation with DyS and StS for synthetic applications

and FIFOs. For the DyS scenario with HWTSM, denoted as *RTOS+HWTSM* in Figure 7.7, the WCET of the RTOS does not change when the number of FIFOs per task is increased. To give a better view on the execution time, we introduce Figure 7.8, which is based on the results from Figure 7.7. Keep in mind that Figure 7.8 is a projection of Figure 7.7. The execution time of the RTOS for *RTOS+HWTSM* is the same throughout all scenarios. It is because the HWTSM computes in parallel the tasks-status, which depends on the number of FIFOs per task. The non-linear change at the point of two FIFOs per task is due to our particular implementation.

In Figure 7.9, we summarize the achieved reduction in the WCET of the RTOS. The reduction is computed as a ratio between the WCET of the RTOS between pure software and HWTSM implementations. Figure 7.9 suggests that the reduction in the WCET of the RTOS for StS varies between 1.1 and 1.8 times compared to the pure software implementation. The results indicate that for the DyS scenarios, the reduction is between 1.1 and 3.0 times.

We follow the implementation approach from [72] to partition and map the JPEG and the H.264 decoders on the baseline CompSoC platform. The application inter-task communication pattern of the JPEG decoder is presented in Figure 7.10. We partition the JPEG decoder into three tasks. One of the tasks

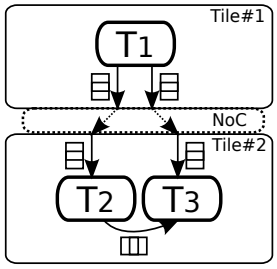


Figure 7.10: JPEG decoder

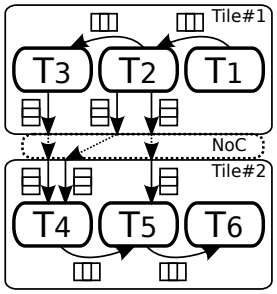


Figure 7.11: H.264 decoder

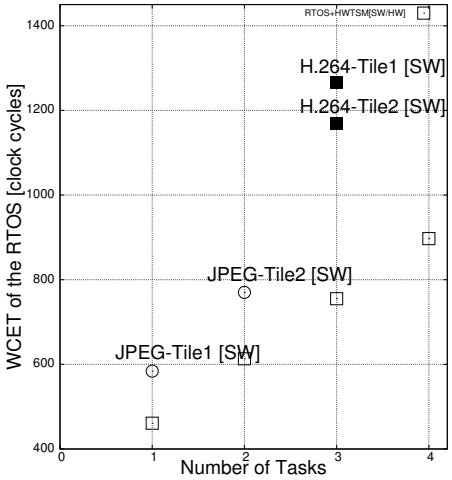


Figure 7.12: WCET of the RTOS for JPEG and H.264 decoders

Table 7.8: Overall system performance improvement

	RTOS slot : Application slot ratio	
	10% : 90%	20% : 80%
StS	0.9% − 4.5%	1.8% − 8.9%
DyS	0.9% − 6.6%	1.8% − 13.3%
JPEG	2.3%	4.6%
H.264	3.8%	7.5%

is mapped on one processor tile and the other two are running on the other processor tile. The application inter-task communication pattern of the H.264 decoder is depicted in Figure 7.11. We partition the H.264 decoder into six tasks. Each of the processor tiles executes three of the tasks.

In Figure 7.12, we present the WCET of the RTOS in each one of the tiles for the JPEG and H.264 decoders with and without the HWTSM. For the JPEG decoder, the reduction of the WCET of the RTOS with HWTSM is up to 1.3 times. Although the H.264 decoder has an equal number of tasks in each tile, we observe small deviations in the measurements, caused by the different number of the input/output FIFOs of the mapped application tasks. The reduction in the WCET of the RTOS with HWTSM is up to 1.6 times. The reason for the high WCET reduction of the RTOS even with low number of tasks is due to the *parallel non-blocking* execution model which leads to a constant, short in our case, response time of the HWTSM equal to five cycles.

Once the WCET of the RTOS is reduced, there are at least three possibilities to utilize the extra clock cycles. The first option is to improve the overall performance. The second one is to improve the system responsiveness, by increasing the rate of RTOS invocations, while preserving the application performance as in the pure software implementation. The third option is a combination of the previous two. Table 7.8 presents the overall system performance when the reduction in RTOS WCET is used to speedup the application. We investigate two cases, when the ratio of the RTOS slot to application slot size is 10%:90% and 20%:80%, respectively. As the reduction in the WCET of the RTOS slot varies between 1.1 to 3.0 times when the RTOS slot size is 10% of the total execution, the overall system performance improvement is between 0.9%-6.6%. If the RTOS slot size is 20% of the total execution time, then the overall system performance improvement is between 1.8%-13.3%. If the number of tasks and FIFOs per application are further increased, we expect to achieve even higher overall performance improvement than the current results.

One of our contributions in this dissertation is the definition of the *parallel non-blocking* execution model. To assess it, we provide a qualitative comparison

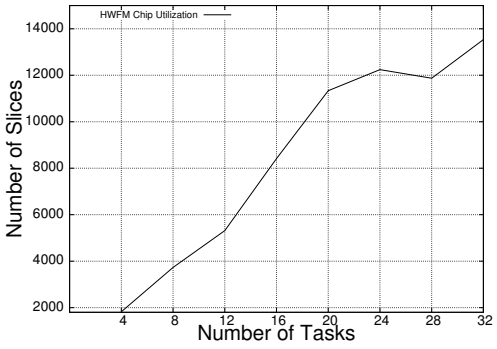
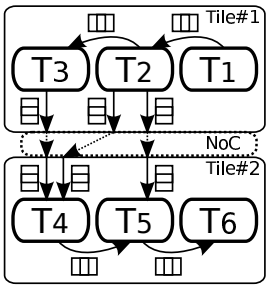


Figure 7.13: HWTSM – chip utilization [4 FIFOs per task]

on four of the execution models presented in Figure 3.1 with respect to the presented HWTSM operation. If we execute the HWTSM in *sequential* model, then the software should be stalled for some time. In this time, the HWTSM reads the *rc*/*wc* values from the data memories and computes the status of the application tasks. Therefore, the HWTSM should be sharing the data memory ports with the tile processor because the latter is not using them at the same moment in time. Such an approach enables an easy integration in the existing platform. The total execution time of the RTOS with HWTSM, running in the *sequential* model, is expected to be always longer and more variable than the HWTSM executed in the *parallel non-blocking* model.

A possible scenario to execute the HWTSM in the *parallel blocking* model is to make the HWTSM calls in advance, e.g., at the beginning of the RTOS slot. Therefore, the fetching of the *rc*/*wc* values is overlapped with the rest of the RTOS services. A way to access the *rc*/*wc* memory locations in the CompSoC platform without affecting the execution of the processor and the network interfaces is by adding an extra memory port to the data memories. The high hardware cost of implementing a three-port data memory makes the *parallel blocking* model impractical for the proposed HWTSM in the CompSoC platform.

In Figure 7.13, we present the hardware costs of the HWTSM in terms of number of slices on the FPGA chip. The results are obtained by varying the number of hardware task-status units from 4 up to 32, identified as *FSM T** in Figure 5.8. Each *FSM T** represents a separate state machine. As expected, the hardware cost scales close to linear with the number of tasks. The deviations from the linear behaviour are due to the *Place* & *Route* heuristics in the FPGA design tools.



Tile 1	wcet	Tile 2	wcet
T1	114 776	T4	174 067
T2	577 522	T5	232 294
T3	152 698	T6	20 602

Figure 7.14: H.264 tasks: mapped on CompSoC processor and WCET (clock cycles)

7.4 Remote Slack Distribution Evaluation

In this section, we first briefly describe the platform and the tools. Then, we present the target application, namely H264, and the experimental setup. In the end of this section we present the observed clock frequencies of the tiles, consumed energy and costs in software and hardware.

For our experiments, we employ a dual-tile CompSoC platform. Each tile embeds a Xilinx Microblaze core. The design is synthesized with Xilinx Platform Studio 12.3 and verified on Xilinx Virtex ML605 (xc6vlx240t) evaluation board. Moreover, we consider that each processor can operate in 15 equidistant frequency levels, varying from 50MHz downto 3.1MHz. We consider the same energy model as [71].

We exercise a streaming data-flow H.264/AVC decoder [72], to evaluate our slack distribution technique. In Figure 7.14, we present the H.264 tasks to tiles mapping and we list the worst-case execution times of tasks. The H.264 application has static slack, due to the difference in the *wcet* of the tasks, and dynamic slack, caused by the variations of the *acet*s of tasks.

We compare our inter-tile slack distribution with intra-tile slack management (Intra-tile slack mngm) [67]. Furthermore, we investigate two variants of our proposal, depending on the computation of S_{task}, i.e., the value of N in Equation 6.16. In the first case, the remote slack is the maximum value among the remote slack values from the FIFOs linked/connected with the task that is schedule to start next (Inter-tile slack mngm (MAXtask)). In the second case, the remote slack is the maximum value among the remote slack values from the FIFOs of all tasks mapped on the tile (Inter-tile slack mngm (MAXtile)). In both cases, we utilize a simple, greedy slack policy that always allocates all the available slack to the next scheduled task.

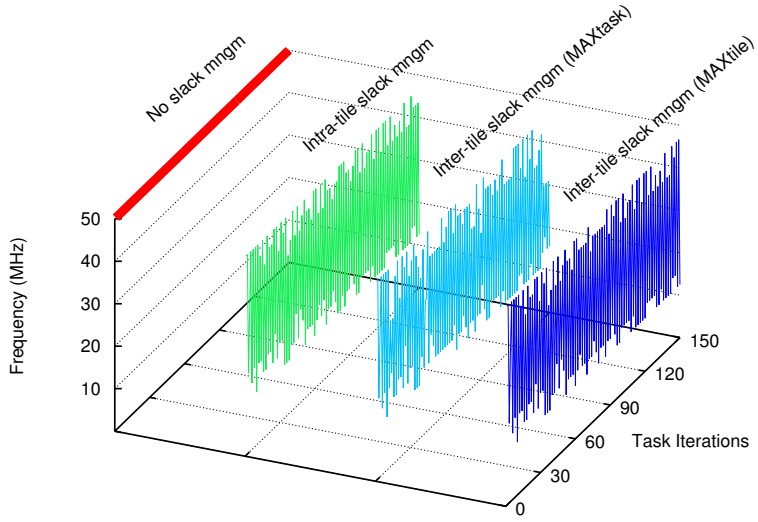


Figure 7.15: Frequency levels for the H.264 tasks running in Tile 1

In Figure 7.15 and Figure 7.16, we present the clock frequency of Tile 1 and Tile 2 for 150 task iterations. In case when no slack management is applied the core operates all the time at the maximum frequency (f_{max}). For Tile 1, the proposed inter-tile slack technique does not always achieve lower frequencies than the intra-tile technique. Compared to the existing intra-tile technique [67], the total frequency level reduction of the tasks for MAXtask is 5.7% and for MAXtile is –0.3%, respectively. There are two main reasons for it: 1. our slack policy employs more slack in a given task iteration and there is not enough slack for the policy to scale down the frequency in the following task iterations; 2. the transferred remote slack value is lower than the RS library cost, therefore, RS library introduces slow-down, instead of speed-up in the application.

For Tile 2, as the results suggest, the MAXtile case finds lower clock frequency than MAXtask. This difference is because of the amount of the S_{wasted} and the task that employs S_{remote}. For example, if the S_{wasted} is equal to zero, then the ratio between the $acet_{T_6}$ and the received remote slack is more than 15 times. In such a way, the task clock frequency can be set to the lowest possible. Compared to [67], the total frequency level reduction for MAXtask is 48.3% and for MAXtile is 56.8%.

In Figure 7.17 and Figure 7.18 we present the consumed energy for each one of

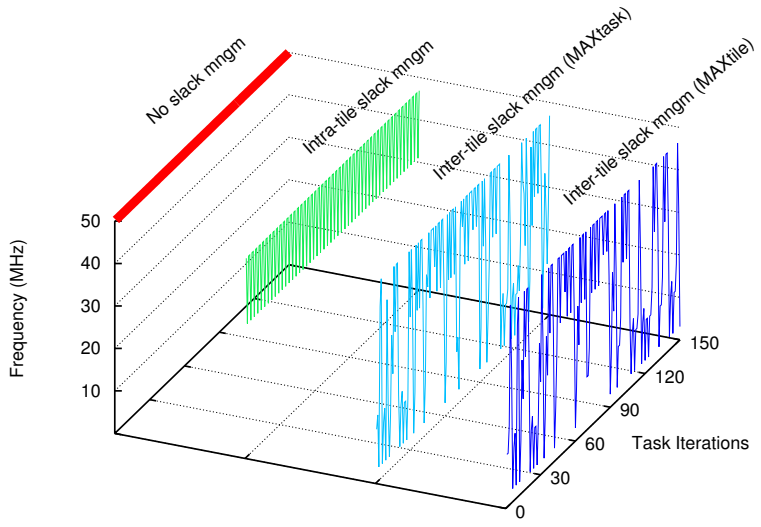


Figure 7.16: Frequency levels for the H.264 tasks running in Tile 2

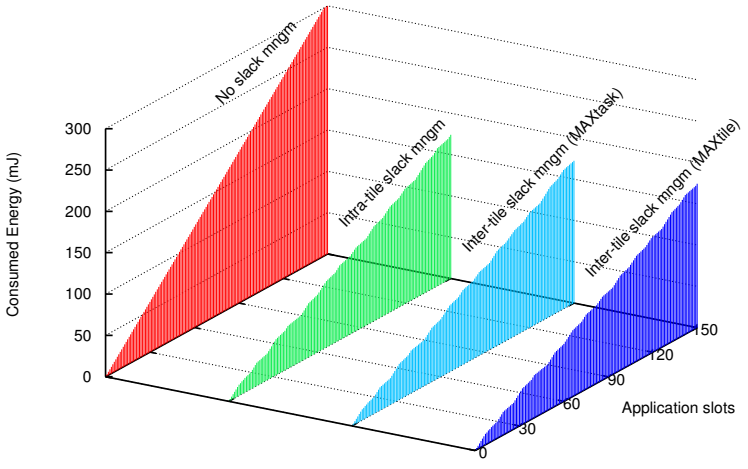


Figure 7.17: Consumed energy for the H.264 tasks running in Tile 1

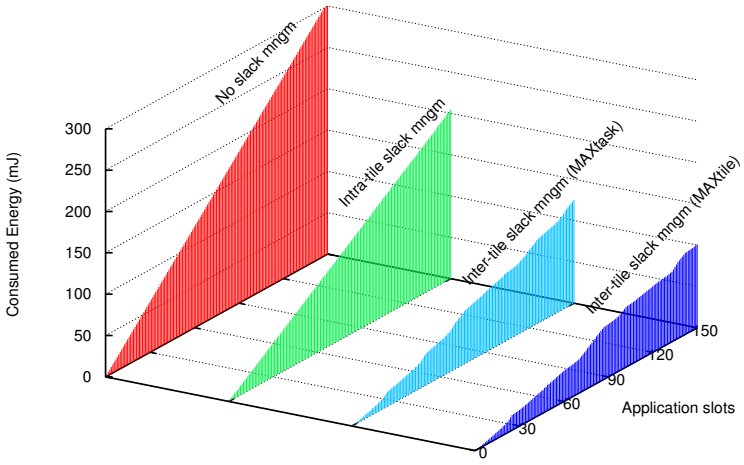


Figure 7.18: Consumed energy for the H.264 tasks running in Tile 2

the previously introduced cases, for Tile 1 and Tile 2, respectively. We run the system for 150 application slots. In Figure 7.17, the consumed energy for our proposals are almost equal to the intra-tile technique, as it was suggested by the clock frequency of Tile 1 in Figure 7.15. The total reduction of the consumed energy for MAXtask and MAXtile, compared to [67] is equal to −0.03% and 0.17%, respectively. In Figure 7.18, we observe that the consumed energy of each of MAXtask and MAXtile is lower than the intra-tile slack management, as also suggested by the frequency levels in Tile 2. During the first few application slots, the consumed energy is equal for the two implementations of our proposal. The reason is that it takes time for the inter-tile communication to be established. In Tile 2, compared to the intra-tile technique, MAXtask and MAXtile reduce energy consumption with 46.5% and 53.3%, respectively. Based on the measured energy reduction, we computed the total reduction over both tiles.

In what follows, we present the software and hardware cost of the RS library and RS CCU. The software cost is in terms of extra clock cycles. The hardware cost is in terms of chip utilization, i.e., the number of occupied FPGA slices. Depending on the number of remote FIFOs, the software cost of the RS library is as follows: for the transmitting of the remote slack varies from 1.5k up to 3.2k clock cycles. As illustrated in Figure 7.14, the number of re-

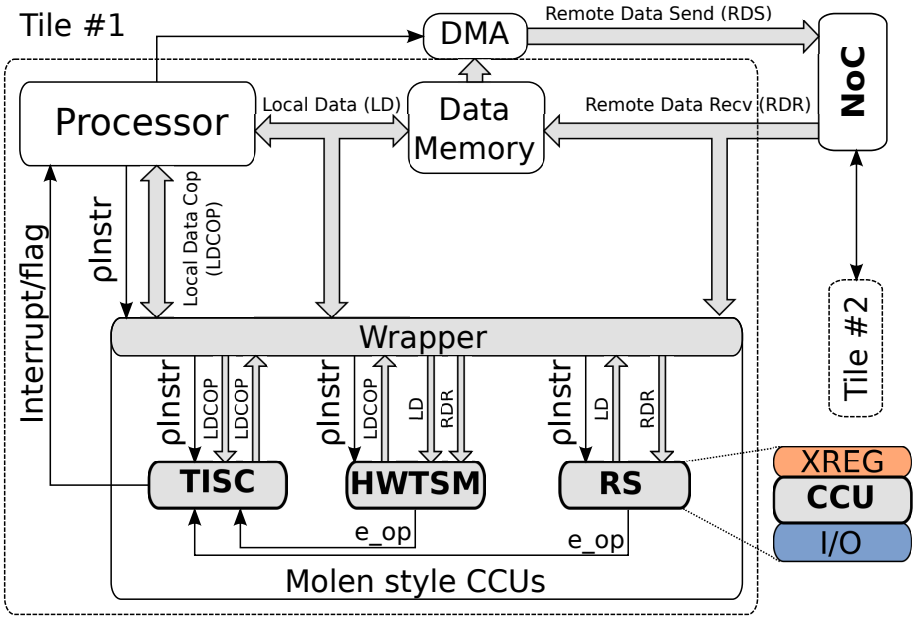


Figure 7.19: Conceptual MPSoC extended with three Molen-style CCUs – TISC, HWTSM, and RS

mote FIFOs varies between one and two. The software cost of the slack policy varies from 2.0k up to 2.5k clock cycles. Since, the RS library is invoked at the beginning and at the end of a task iteration, the introduced cost varies between 1% and 4% compared to the execution time of the H.264 task iterations directly involved in the remote slack transmission and reception.

The chip utilization of the Molen wrapper with a dedicated memory bank of 64 bytes is 585 slices. The cost of the RS CCU is 84 slices. As we expected the chip utilization of the RS CCU is negligible, less than 0.002% of the total number of slices in the considered FPGA chip. Furthermore, the hardware cost of the RS CCU does not dependent on the number of applications, tasks, and FIFOs. The only resource which scales with the number of remote FIFOs is the data memory that stores remote slack and timestamps. Although the RS CCU runs continuously due to its small footprint, we expected its energy consumption to be low as well.

7.5 Overall Results

In this section, we introduced a conceptual architecture, which illustrates that all three previously introduced CCUs, namely: TISC CCU, HWTSM CCU, and RS CCU, can be combined together. The TISC CCU operates in processor–coprocessor parallel blocking model, while HWTSM CCU and RS CCU operate in parallel non-blocking. In Figure 7.19, we present an MPSoC tile extended with Thread Interrupt State Controller, Hardware Task-Status Manager, and supporting inter-tile remote slack distribution. For the proper operation of the CCUs, we connect each one of the CCUs to the following buses: 1. Local Data Coprocessor (LDCop) bus employed for the communication between the processor and reconfigurable coprocessors; 2. Local Data (LD) bus employed for communication between the processor and the local data memory; 3. Remote Data Receive (RDR) bus employed for inter-tile communication, i.e., for receiving data from other tiles through the NoC. Note, that we run HWTSM CCU and RS CCU in processor–coprocessor parallel non-blocking programming model. Therefore, their end_op signals are employed only during their explicit termination from the software and not during their normal operation. For further details on the transferred data through the buses by each of the CCUs, we refer the interested reader to Chapters 4, 5, and 6.

Based on the available experimental results, we estimate that the improvement in the system speedup can be up to 19.6 times with the help of the Thread Interrupt State Controllers. Furthermore, we reduce RTOS cost with the help of the Hardware Task Status Manager, which results in additional application acceleration can be up to 13.3%. Last but not least, the improvement of the system energy consumption can be up to 56.7% over current state of the art with the help of inter-tile remote slack information distribution framework.

Overall, with the help of our contributions, the system performance is improved, the predictability and composability are preserved, all with reduced energy consumption.

7.6 Conclusions

For each one of the previously introduced CCUs, we have carried out extensive experimental evaluations and we outlined the following conclusions:

For the Thread Interrupt State Controller, we verify the system by means of synthetic benchmarks as well as by real applications. Comparison in the ex-

perimental results between our proposal and other state of the art proposals suggested that our approach demonstrated the best performance-portability and performance-flexibility characteristics.

The hardware complexity of the Hardware Task Status Manager grows close to linearly with the number of tasks. The experimental results are obtained with synthetic and real applications. With synthetic applications, the WCET reduction of the RTOS up to 3 times. With the real applications, i.e., JPEG and H.264 decoders, the WCET of the RTOS is reduced by 1.3 and 1.6 times, respectively. The results suggested that our technique leads to overall system performance gains up to 13.3%.

We verify our framework for slack computation, allocation, and distribution that transfers the static and dynamic slack information among the tiles in an MPSoC with a data-flow implementation of the H.264 decoder. More precisely, we studied four scenarios with the H.264 decoder: a no-slack management, an inter-tile slack management, and two variants of our inter-tile slack management. As evaluation criteria of our framework, we employed the clock frequency, the consumed energy, and the introduced cost in software and hardware.

The results suggested that our inter-tile technique reduces the total energy consumption of 27% at the cost of minor software cost of up to 4% and negligible additional FPGA chip utilization of 0.002%.

As a result of our proposal, a proof-of-concept system that includes all three previously proposed reconfigurable coprocessors has improved performance, preserved predictability, and preserved composability, all at the cost of reduced energy consumption.

8

Related Work

Note. The content of this chapter is based on the following papers:

P. G. Zaykov, G. K. Kuzmanov and G. N. Gaydadjiev, ***Reconfigurable Multithreading Architectures: A Survey****, Proc. Int'l Workshop on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS IX), 2009, pp. 263–274*

P. G. Zaykov, G. K. Kuzmanov and G. N. Gaydadjiev, ***State of the art Reconfigurable Multithreading Architectures****, Technical Report - CE-TR-2009-02, 2009*

This chapter provides a survey on the existing proposals in the field of reconfigurable multithreading (ρMT) architectures. Until the time of this survey, reconfigurable architectures have been classified according to implementation or architectural criteria, but never based on their ρMT capabilities. More specifically, we identify reconfigurable architectures that provide *implicit* support for ρMT, *explicit* support for ρMT, and *no architectural support* for ρMT. Further subdivision of these three classes is also provided by the taxonomy proposed in this chapter. For each of the referenced works, we discuss the conceptual model, the limitations and the typical application domains. We also summarize the main design problems and identify some key research questions related to highly efficient ρMT support. In addition, we discuss the application perspectives and propose possible research directions for future investigations.

8.1 Introduction

Contemporary embedded systems require high processing power and often employ varieties of different functionalities. One widely spread technique to combine multiple functionalities and improve Instruction Level Parallelism (ILP) over non-consecutive instructions is multithreading or Thread Level Parallelism (TLP). Modern appliances, such as digital cameras, mobile phones, personal media players, handheld gaming consoles and many electronic devices used in medical and automotive industries employ multithreading supported either by the Operating System (OS) or by dedicated hardware mechanisms.

Many applications running on modern embedded devices are composed of multiple threads, typically processing (exchanging) data among multiple sources. Such examples are: drivers that handle user interaction from touch-screen displays/buttons, management of wireless network protocols, multimedia (audio and video) computations, etc. During the quest of maximum performance and flexibility, the hybrid architectures combining one or more embedded General Purpose Processors (GPPs) with reconfigurable logic have emerged. There is a clear trend which shows that in the near future there will be more embedded systems integrating reconfigurable technology. The first indications of such approaches were presented in [41], [89], [110]. It is envisioned that multithreading support will become an important property of such systems.

One of the fundamental problems in multithreaded architectures is efficient system resource management. It has been successfully solved in contemporary GPPs using various implicit and explicit methods. In literature [109], the explicit techniques have been further partitioned into three main categories: Block Multithreading (BMT) - employing Operating System (OS)/compiler approaches and Interleaved/Simultaneous Multithreading (IMT/SMT) using hardware techniques. The main reason is that the reconfigurable hardware is changing its behavior per application, unlike the GPPs, which have fixed hardware organization regardless the programs running on them. Yet, current state of the art architectures do not provide efficient holistic solutions for accelerating multithreaded applications by reconfigurable hardware.

In this chapter we approach the reconfigurable multithreading (ρMT) architectural problems both from the hardware and the software perspective. The specific contributions of the chapter are as follows:

- We analyze a number of existing reconfigurable proposals with respect to their architectural support of ρMT. Based on this analysis, we propose

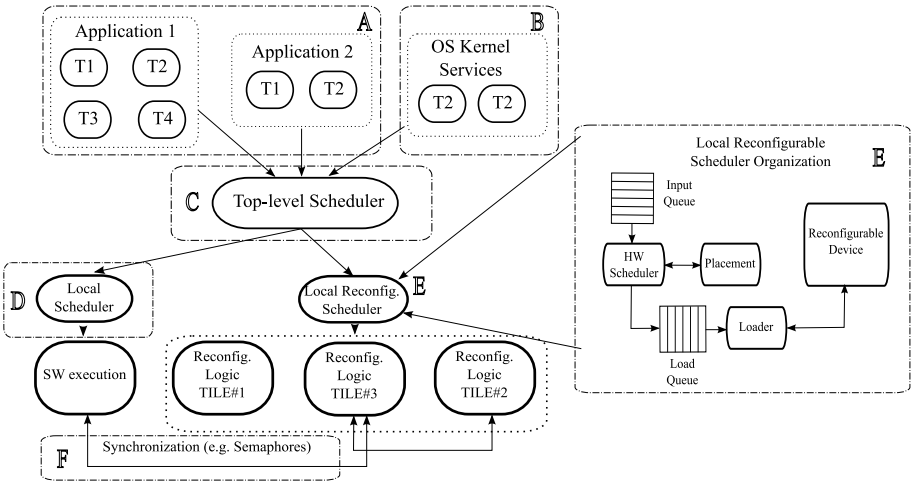


Figure 8.1: A conceptual behavioural model of an ρMT system

a taxonomy with three main classes, namely: reconfigurable architectures with *explicit*, *implicit* and *no* ρMT *support*. A further subdivision of these three classes is also provided;

- We summarize several design problems addressing performance efficient management, mapping, sharing, scheduling and execution of threads on reconfigurable hardware resources;
- We provide our vision for promising research directions and possible solutions of the identified design problems;

The chapter is organized as follows: in Section 8.2, a taxonomy covering related projects is presented. In Section 8.3, we describe in more detail the design problems. the status of the current state of the art, our vision on some possible application perspectives. Finally, the concluding remarks are presented in Section 8.4.

8.2 A Taxonomy of Embedded Reconfigurable Multithreading Architectures

A taxonomy on Custom Computing Machines (CCMs) with respect to explicit configuration instructions has been already proposed in [91]. However, that study did not consider multithreading support as a distinguishing feature. In

this section, we introduce a taxonomy of existing reconfigurable architectures with respect to the ρMT support they provide. We identify three main classes of such architectures, namely: with *explicit*, with *implicit*, and with *no architectural* ρMT support. Note, the meaning that we relate to the definitions of *explicit* and *implicit* ρMT, is different from what is used in GPP systems. In general purpose systems, the classification is based on multithreading support from algorithmic point of view [109]. In our taxonomy we use as a distinguishing feature the presence of architectural/μ-architectural extensions for creation/termination of multiple threads on reconfigurable logic. If we classify the ρMT research projects based on the GPP explicit multithreading technique, our taxonomy looks like:

- Reconfigurable Block Multithreading (ρBMT): e.g. [110], [118], [119];
- Reconfigurable Interleaved Multithreading (ρIMT): e.g. [61];
- Reconfigurable Simultaneous Multithreading (ρSMT): e.g. [107], [79];

In this chapter, we consider a different classification perspective. In architectures with *no* ρMT *support*, application threads are mapped into reconfigurable hardware using software techniques – either at the OS or at the compiler level. This software approach provides unlimited flexibility, but the performance cost too often penalizes the overall execution time especially for real-time implementations. On the other hand, architectures with *implicit* ρMT support, provide performance efficient solutions at the cost of almost no flexibility due to the fixed underlying microarchitecture (μ-architecture) facilitating multithreading. To exploit the flexibility provided at both the software level, as well as by the reconfigurable hardware at the μ-architectural level and to achieve higher system performance, a third emerging class of architectures is identified and termed as architectures with *explicit* ρMT support. Hereafter, we enlighten the proposed taxonomy through examples of existing reconfigurable architectures.

A conceptual behavioral model of an ρMT system is depictured in Figure 8.1. The picture represents the basic steps in the management and execution process of multiple threads. Initially, the programmer creates applications (tasks – Section A) or kernel service (Section B) composed of multiple threads. Later, during run-time, when an application is selected for execution, depending on the system status information, the Top-level Scheduler (Section C) passes threads to local schedulers (Section D and E). The local reconfigurable scheduler (Section E) composed of multiple units - queues, scheduling algorithm, placement

technique and loading process. The synchronization between different threads is managed by e.g. semaphores or read/write counters (Section F). The different sections of the behavior model, depicted in Figure 8.1, are implemented either at the software level, or at the architectural, or at the μ-architectural level, depending on the particular architecture. Hereafter, we shall reveal how different popular reconfigurable proposals manage the scheme from Figure 8.1 and based on their architectural support for ρMT, we shall classify them.

8.2.1 State of the art Reconfigurable Architectures

In general, reconfigurable hardware allows the designer to extend the processor functionality both statically and at run-time to speed up the application by executing its critical parts in hardware. In [28], a survey on architectural proposals targeting GPP cores extended with reconfigurable logic is presented. However, that paper has not considered ρMT as a classification criterion. In the years after, a few more reconfigurable proposals have been introduced, capable to be supported by an OS without any specific hardware modifications. We choose to briefly introduce the following two of these reconfigurable projects, uncovered by [28], because we consider them as a natural evolution of contemporary embedded systems and potentially good candidates for explicit ρMT extensions:

MOLEN: We choose The Molen Polymorphic Processor [110] proposed by CE Lab, TUDelft, The Netherlands, as an example of tightly coupled (processor–coprocessor) fine-grained reconfigurable architecture. It combines a GPP with several reconfigurable Custom Computing Units (CCUes). Further details on the Molen Polymorphic Processor are revealed in Section 2.1.1.

The Molen concept has been successfully employed in various projects in the domain of multimedia [50] and cryptography [26]. In the Molen original papers, multithreading has not been discussed, but a follow-up research towards multithreading has been reported in [107]. An overview of this enhanced MT version of Molen is examined in Section 8.2.3.

Montium TP: As an example of a Coarse Grained Reconfigurable Array (CGRA) processor core, we choose Montium TP [41], designed by RECORE Systems. It combines five identical Custom Reconfigurable Processing Units (CRPUs) in a single chip, connected through a simple 2D-mesh communication infrastructure. This architecture has the following characteristics: once configured, it does not issue any instructions (just processes the

data). It does not have a fixed instruction set architecture (ISA) - the application is encoded at microcode level and has fast reconfiguration response time, because of its coarse-grained hardware structure. In its current implementation, the Montium TP is capable to support execution of multiple threads (applications) but only at the OS level. The CRPU does not have support for context-switching - multiple threads are not capable to share the same hardware resources. The processor was originally targeting the domain of streaming applications: i.e., broadcast and multimedia.

Although in our investigations we are particularly interested in the embedded systems domain, for the completeness of our survey, we also consider briefly a few large-scale (high performance) proposals. Many contemporary mainframe/supercomputing platforms, such as Altix Family by SGI [2], ProLiant server by HP [6], Convey hybrid-core HC-1 [4], RAMP emulation platform [3], employ reconfigurable hardware to speed-up computationally intensive kernels. Though no particular attention on ρMT on these machines is paid in the literature, we believe that multithreading applications can be mapped on them using different (not trivial) software techniques [101].

Preliminary investigations indicate that an efficient ρMT processor would allow better overall system performance [107] as the thread management cost at the software level could be dramatically reduced. In addition, [107] provides clear indications that multithreading should be also addressed by the state of the art real-time reconfigurable systems at the hardware level.

The necessity for an embedded, real-time ρMT processor has been also identified in several integrated EU projects, such as MORPHEUS [104] and hArtes [1]. Currently, in those EU projects, as well as in other related reconfigurable research efforts, multithreading is supported either by the OS, by the compiler or at the application level.

Unfortunately, to our best knowledge, there is no complete study on the design perspectives of the architectural support for reconfigurable MT reported in the literature so far. Therefore, it is our ambition with this chapter to provide the results of such a study in the sections to follow.

8.2.2 Architectures with No ρMT Support

In this section, we consider all those reconfigurable proposals, which do not provide any architectural facilities to support ρMT. For all these architectures, multithreading is supported at the software level only – either by the OS, or by the compiler. The thread context is represented as a software data structure.

OS Support for ρMT

In this section, we group all known OS targeting reconfigurable devices and implementing in software - Section A, B, C, D, E and F from Figure 8.1. The first proposal, which identifies some of the necessary services, that an Operating System for reconfigurable devices should support, is presented in [118] and [31] by a research group at the University of South Australia.

BORPH [93]: The research work presented in [93] and [94] by the University of California - Berkeley, identifies that application migration from one reconfigurable computing platform to another, using conventional codesign methodologies, requires from the designer to learn a new language and APIs, to get familiar with new design environments and re-implement existing designs. Therefore, BORPH is introduced as an OS designed specifically for reconfigurable computers, sharing the same UNIX interface among hardware and software threads, which speeds up the design process. The major difference between BORPH and conventional OS-es for Field Programmable Gate Array (FPGA) architectures comes from the fact that the system reconfigurable logic are treated as a first-class computational resources instead of coprocessors. The BORPH contains three basic components: concept of hardware process and a set of universal interfaces - input/output registers (IOREG) and hardware file input/output (I/O) interface. The proposal has the following limitations - hardware executed threads are not sharing reconfigurable resources. Experimental results are produced from simple applications such as: wireless signal processing, low density parity check decoder and MPEG-2 decoding.

SHUM-uCOS [123]: Another design, tackling the problems caused by the essential differences between software and hardware-tasks is the SHUM-uCOS by the Fudan University, China [123]. The authors propose an real-time OS (RTOS) for reconfigurable systems employing uniform multi-task model. It traces and manages the utilization of reconfigurable resources, improves the utilization and the parallelism of the tasks with hardware task preconfiguration. Detailed descriptions of the abstract layers and their functionality are presented in [123] and [124]. For evaluation of the system, the authors use benchmarks and multiple voice over Internet protocol (VOIP) compression/decompression algorithms. The limitations on the current implementation of SHUM-uCOS are in the static scheduling approach (they adopted multiprocessor critical path scheduling algorithm - [51]) and the resource reusage, supported by the compiler only.

Compiler Techniques for Multithreading on Reconfigurable Platforms

The most common feature of the architectures grouped in this subcategory is the responsibility of the compiler for task partitioning, scheduling and management of the system resources. The major reason to employ multithreading in these architectures is to hide reconfiguration latencies.

MT-ADRES [119]: In MT-ADRES by IMEC, Belgium, the DRESC Compiler Framework [65] has been extended to support several threads.

A major limitation of this proposal is the inability to execute/terminate threads at run-time which is posed by the compiler static scheduling and optimization algorithms, operating with Control Data Flow Graph (CDFG). Control decisions, such as hiding the reconfiguration latencies and resource management are taken at compile time. Due to the implementation complexity and the fact that the Very Long Instruction Word (VLIW) processor and the CGRA have complete access to the register file, the DRESC compiler [65] limits the execution to only one computing resource at a time, which reduce potential performance gains. All experiments providing information about MT-ADRES performance are achieved through multimedia simulations, without any real hardware implementations.

UltraSONIC [40]: Another proposal falling in this category is the UltraSONIC project, represented by Sony Research Labs, UK [40]. It is a reconfigurable architecture optimized for video processing. It has a list of Plug-In Processing Elements (PIPEs), connected through several buses. The programmer receives an architecture abstraction through an API interface. In [117] and [74], the authors introduce multitasking to the architecture. The goal is achieved through two-phase clustering algorithm working on a Directed Acyclic Graph (DAG). The phases are: partitioning (based on Tabu Search) and list scheduling (a static technique). The algorithm places and schedules tasks, applying the following system constraints: FPGA resources, shared resource conflicts, configuration time, communication and processing cost. The system also has a Task Manager, responsible for task creation and termination procedures. Because of its static nature, the architecture has the same limitations as the MT-ADRES project [119]. The system is initially designed for the multimedia and the data encryption application domains.

8.2.3 Architectures with Implicit ρMT Support

The proposals from this category share one common feature - the detailed multithreading support on reconfigurable threads is *implicit*, i.e. hidden from the system programmer. The Instruction Set Architecture (ISA) does not have dedicated special instructions for thread creation and termination procedures. The functionality is achieved with μ-architectural extensions while preserving the architectural model. Bellow, we describe some of the existing proposals falling into the category.

Reconfigurable Extensions for the CarCore Processor: In [107], the authors combine a simultaneous multithreaded processor CarCore [108](a simulation model, architecturally compatible with Infenion TriCore 1 Processor) with a Molen style reconfigurable coprocessor [111]. To minimize the complexity of the implementation, the authors employ several constrains to the architecture. They modeled a hardware scheduler, which supports execution on reconfigurable logic of only one thread at a time, preserving the real-time capability for it. Once a thread is started for hardware execution, it could not be interrupted until it is finished (no context-switching). There is no additional ISA extensions for reconfigurable thread management. Meanwhile, other non-real-time threads can continue their execution employing the latencies of the real-time thread. The implementation includes two scheduling policies – fixed-priority and round-robin, over four executing threads. The ISA extension comprises only the Molen polymorphic ISA and no additional specific instructions for multithreaded support.

The **REDEFINE** project [88], [10] by the Indian Institute of Science, Bangalore, proposes a synthesis methodology to realize applications written in a high level language (HHL) on the coarse-grained Runtime Reconfigurable Hardware (RRH). Contrary to related projects in the field, they assume that the whole application could be represented as a set of custom instructions executed on RRH. The custom instructions are not based on occurrence statistics, but are based on co-execution (e.g. both paths of branch instruction). The transformation of the application (in HHL) to hardware proceeds in three steps: 1. The compiler transforms HHL specification to DFG; 2. The compiler partitions the DFG into various application substructures called HyperOps (equal to a thread in our classification); 3. The HyperOps are synthesized into hardware configurations. At run-time, depending on the availability of computing elements on the fabric and the data dependencies among different HyperOps, a subset of clusters (composed of HyperOps) ready for execution are scheduled based on pre-order depth first search. Each HyperOp contains information about its

data-dependent successors. Because of the lack of detailed description, we assume that this control information is provided implicitly.

Hthreads [79]: The Hthreads(Hybrid Threads) model presented by University of Kansas [79], [12] is multi-layer computational architecture which aims to bridge the gap between the programmers and complex reconfigurable devices. Some of the main system features are the migration of thread management, synchronization primitives and run-time scheduling services (Figure 8.1, Section F) for both hardware and software threads into hardware module accessed from the GPP only through an universal bus. The authors represent hardware threads with user defined component (designed by the programmer), state controller and universal interface (Register Set). Synchronization procedures are performed through semaphores. In the proposal, the CPU is only interrupted when a change in the system state requires the CPU to switch to another activity. Such changes include timers expiring, devices completing an assigned activity and generating an interrupt. The basic system components are: 1. software thread management (SWTM) which is only responsible for scheduling of software threads. It is executed in parallel with the CPU threads, which reduce the cost and context switching jitter. The SWTM scheduler manages all CPU interrupt requests, including external-device interrupt, expiring timers, terminating, blocking and unblocking threads; 2. hardware thread interface (HWTI) controller which provides management and distributed control (through command and status registers) of threads executed on reconfigurable resources. Some of the major limitations in the implementation of the Hthreads model are as follows: 1. threads executed on reconfigurable resources are not scheduled, instead they are directly loaded when it is necessary and 2. threads are not sharing the reconfigurable resources even when the thread is marked as blocked/idle. Because of the fact that the system does not have modifications at architectural and μ-architectural levels, the proposal is classified as an *implicit* ρMT. The experimental results are provided in the image processing application domain.

Reconfigurable Multithreaded Architecture Model [115], [114]: The proposal is presented by a research group in the Hamburg University of Technology, Germany. Their primary idea is to map computational threads via pipelined configuration technique into available physical reconfigurable hardware resources. The fixed resource limitations are overcome by virtualizing the computational, communication and memory resources in the reconfigurable hardware. The architecture is based on a synchronous multifunctional pipeline flow model using coarse-grained reconfigurable processing cells and reconfigurable data paths. Descriptors are used for run-time and partial reconfigu-

ration, which enables the processor cells to be configured by Time Division Multiple Access (TDMA). By itself, the descriptors represent small configuration templates in special opcodes, extending a conventional ISA. Therefore, the ISA grows proportionally with the design complexity and the number of the configuration templates. Compared to existing architectural proposals, the difference comes from the fact that, the authors do not employ the GPP to control the reconfigurable resources. Instead, a hardware approach is taken - a Microtask controller is employed. Current implementation does not support dynamic (runtime) scheduling of incoming workloads. The ideas are not applied in heterogeneous systems yet, represented by a combination of GPP and reconfigurable logic. The designer is responsible for partitioning and mapping the CDFG in microtasks (subtasks), by allocating the flow graph nodes to the system processing resources. The simulation results of streaming multimedia applications are studied.

The research group at the university of Karlsruhe [16] introduces an architecture capable to manage execution of multiple run-time threads (called Special Instructions - SI) through a 'Special Instruction Scheduler'(SI scheduler). Each Molecule is composed of one or several Atoms representing elementary data paths. Multiple Molecules (varying in resource usage & performance) compose each Special Instruction. As a result, the SI Scheduler implicitly selects (without additional control instructions) for execution a mixture of dynamically loaded data paths with conjunction with base processor instructions. The authors examines multiple run-time algorithms based on Molecule loading sequences. Because of the reduced granularity and increased possibility of resource reusage, the system achieves high system performance, tested with H.264 and CIF-video applications.

8.2.4 Architectures with Explicit ρMT Support

The basic idea of this ρMT class is to combine the flexibility of the software and the reconfigurable hardware with the potential performance efficiency of the latter and to support ρMT, both at the software level and at the μ-architectural level. There are several partial solutions in the literature which do not provide such a compete mixed model of ρMT - the software and the hardware corporate together to provide simultaneous execution of multiple threads. In such a model, the system services (e.g. scheduling, resource management) should be optimally separated between software and μ-architectural levels. Combined with efficient memory management and thread/function parameters exchange through dedicated registers, an architecture with *explicit*

ρMT support would potentially reduce the intra- and inter- thread communication costs. Similar approaches are taken in the following proposals:

OS4RS [64]: In [64], [75] and [76], a research group at IMEC, Belgium, investigates the concepts and reveals some of the open questions, raised by the run-time multithreading and interconnection networks [63] for heterogeneous reconfigurable SoC. The novelty of their approach resides in the integration of the reconfigurable hardware in a multiprocessor system completely managed by the OS for Reconfigurable Systems (OS4RS). The system maintains several threads by a two-level scheduler. The high-level scheduler is handled in software by the main GPP, which stores the running tasks as a linked list. The low-level/local scheduler can be implemented in software or hardware depending on the type of the slave computing resources (GPPs or reconfigurable logic). Note, that in the current implementation of OS4RS, hardware threads are not sharing the same reconfigurable resources. The OS has several services executed on the main GPP, responsible for monitoring the status of the heterogeneous system and distributing the workload among slave processing units. Due to the fact that a software approach is taken to solve heavily computational problems, such as real-time scheduling, resource allocation and loading, it will eventually become a system bottleneck during heavy computation periods. In their current implementation, the top-level scheduler (Figure 8.1, Section C) is implemented in software and the local-level hardware (reconfigurable) scheduler (Figure 8.1, Section E) is not implemented, yet. The authors also propose a proof-of-concept method for context-switching and migration between heterogeneous resources by saving the task state. The questions related to thread state translation between GPP register set and reconfigurable logic are still open. The OS4RS has been tested in JPEG frame decoding and experimental 3D video game. According to the project time schedule, the next generation of the system is expected to be designed between 2008 and 2010.

Reconfigurable Multithreaded processor [60], [61] by the University of Wisconsin-Madison: The authors augment SandBlaster 3000 simulator [34] with Polymorphic Hardware Accelerators (PHAs), which combine properties of functional units and reconfigurable hardware. The processor by itself has four multithreaded Digital Signal Processor (DSP) cores and an ARM processor that provides support for user interface and OS. The research study investigates potential benefits of closely coupled reconfigurable hardware to multithreaded processor. The work could be separated into two topics: the first one is to investigate architectural techniques to provide hardware-software interface between the multithreaded processor and PHAs, the second one is to evaluate the potential benefits of incorporating PHAs in a multithreaded DSP

to improve the system performance. The PHAs are implemented as a functional units at the execution stage of the processor pipeline. The multithreading is mainly employed to hide the reconfiguration time. Each one of the PHA blocks contains PHA control interface, reconfigurable block(executing user logic) and optional registers. Configuration of the PHA is done, by loading a sequence of instructions to specific register, accessed by PHA control interface. If a certain high priority task requires a PHA, it is only dedicated to it, without any interference with the other threads. In case of interrupt, thread's PHA inner state could be saved and the unit is released. Therefore, the processor supports context switching over reconfigurable resources. In case of a lack of PHAs, a realtime thread could preempt a non-realtime one. Once configured, in case of identical PHA instructions, the PHA could be reused by different threads. Some of the system model assumptions are: the PHAs are not sharing the same reconfigurable resource area, as a result there is no necessity for placement algorithms. The architecture is limited to Interleaved Multithreading called Token Triggered Threading. The authors argue the choice of such an approach instead of Simultaneous Multithreading, because of the possible power consumption reduction. The authors investigate two PHA binding techniques - static & dynamic. In case of a run-time binding, the system provides realtime constraints by restricting PHA reusage among threads.

8.2.5 Summary of the Proposed Taxonomy

Based on the criteria of the provided ρMT support, the aforementioned architectures can be briefly classified as follows:

I. *No architectural ρMT support:*

I.1. OS support for ρMT: Molen [110], Montium [41], SGI Altix [2], HP ProLiant server [6], Convey hybrid-core HC-1 [4], RAMP [33], South Australia [118], BORPH [93], SHUM-uCOS [123];

I.2. Compiler techniques for ρMT: MT-ADRES [119], UltraSONIC [40];

II. *Implicit architectural ρMT support:*

CarCore Processor extensions [107], REDEFINE [88], Hthreads [79], Reconfigurable Multithreaded Architecture Model [114], University of Karlsruhe [16];

III. *Explicit architectural ρMT support*:

III.1. μ-architecture + OS: Reconfigurable Architectures of this kind are just emerging. This approach is promising for high performance efficient schedul-

ing and execution of threads on reconfigurable hardware due to the hardware & software co-design of the ρMT managing mechanisms. OS4RS [64];

III.2. μ-architecture + compiler: Reconfigurable Multithreaded processor [61].

8.3 Design Problems

The very basic design questions related to thread scheduling on reconfigurable resources are:

- Which threads to execute, schedule or preempt at certain instance of time (e.g., when the requested reconfigurable area of prepared for execution hardware threads is higher than the available area)?
- Where to place a thread (in case of several possibilities)?
- When to reallocate the newly created threads and how to efficiently hide the reconfiguration latencies?

Depending on model assumptions, from complexity point of view, the scheduling problem on reconfigurable logic could be reduced to several well-known NP-Hard problems [97], [125], [14], [81]. Therefore, one of the ways to be solved is by reducing it to the well-known Bin-Packing problem, i.e., the scheduling problem could be solved by the introduction of an advanced heuristic algorithm. Some partially and completely solved design problems, grouped by topic, are presented in sections that follows.

8.3.1 Hiding Reconfiguration Latencies

In reconfigurable systems, the reconfiguration latency is caused by the time needed for the configuration bitstream to set the reconfigurable device for the particular operation. Typically, configuration latency is introduced during the initial task loading (tasks are composed of one or multiple threads). This is one of the major system delays and causes severe performance degradation in case of frequent reconfigurations. In literature, the most common ways to hide or minimize the reconfiguration latency are:

1. Compressing the task's bitstream. Different techniques are examined in [82];

2. Employing prefetch technique and local caching for earlier reconfiguration (overlap reconfiguration with computations). The existing prefetch technique proposals are grouped into three categories:

- **Static** – predictions are performed at design time by the compiler (e.g., The Molen compiler [78]);
- **Dynamic** – at runtime by the reconfigurable scheduler, which stores most recent configurations [56];
- **Hybrid** (combining the Static & Dynamic approaches) [27], [56]. In case of missprediction, alternative Hybrid methods [56] always pay time penalty, by delaying the reconfiguration. In [32] and [82], the authors propose inter-task placement in case of free reconfigurable area, but it is only limited to periodic hardware tasks. For aperiodic tasks, the problem has not been solved.

8.3.2 Optimized Inter-Thread Communication Scheme

The Erlangen-Nuremberg Slot Machine (ESM) [59] has target several problems common for contemporary FPGA based architectures such as: limitations of partial support on Actual FPGAs; I/O pin, intermodule communication and local memory dilemmas. The authors underline as a major advantage of the ESM platform its unique slot-based architecture which allows the slots to be used independently of each other by delivering peripheral data through a separate crossbar switch. The decision to exploit an off-chip crossbar is in order to have as many available resources on the FPGA for partially reconfigurable modules as possible. The ESM architecture is based on the flexible decoupling of the FPGA I/O-pins from a direct connection to an interface chip. This flexibility allows the independent placement of application modules in any available slot at run-time. As a result, run-time placement is not constrained by physical I/O-pin locations as the I/O-pin routing.

8.3.3 Scheduling and Placement Algorithms

In the research work presented in [99] by ETH Zurich, Switzerland, the authors propose several algorithms to manage the sharing of resources in the reconfigurable surface. Their proposal includes system services for a partial reconfiguration, which by scheduling the dynamically incoming threads solve the problems with complex allocation situations. Detailed description

of the system model could be found in [98] and [112]. The primary idea of the project is to separate threads into two groups according to their arriving times - synchronous and arbitrary. For threads with aperiodic arriving times, the authors propose two non-preemptive techniques: "horizon" and "stuffing" methods [97]. On the other side, for threads with periodic arriving times, authors propose another two preemptive scheduling algorithms: "EDF-NF" and "MSDL" [29]. Unfortunately, the preemptive methods are not adaptable for threads with arbitrary arriving times, because the system cannot guarantee that each preempted thread, previously executed for some period of time, will finish before it's relative deadline. Each one of the scheduling techniques is combined with optimized placement method named "On the Fly Partitioning" [113], based on Bazargan partitioner [17].

The ideas and algorithms are further enhanced by a research group at Fundan University [125]. They introduce an advanced heuristic algorithms based on "stuffing" technique [97]. The authors prove that the combination of a scheduling algorithm with a recognition-complete placement method does not result to a recognition-complete technique. Therefore, they enhanced the "stuffing" scheduling algorithm [97] and named the new one: "windows-based stuffing". In [126], the authors propose "Compact Reservation" (CR) scheduling algorithm which attains recognition-earliest scheduling (arrange the start time of a newly arrived thread as early as possible) by exploiting the knowledge about temporal properties of each thread. In [8] the cases of potential thread migration depending on the workload is examined – a newly arrived thread is started either in software or in hardware. Slightly different approach is proposed in [32] by a research group at the Paderborn University. They enhance a single processor algorithm (e.g., a stochastic server) with preemption support (limited only during the time of reconfiguration) for hardware tasks.

8.3.4 Context Switching

In [46], the authors clearly identify the two possible techniques for context switching of hardware threads in partially reconfigurable FPGAs. The techniques are named as follows:

1. *Thread Specific Access Structures* – when the scheduler decides to switch a thread, it's current state is saved in an external structure. The major advantages of this approach are the high data efficiency and its architecture independence. The disadvantages come from the fact that each thread is different and it is difficult to design a standard generic interface. On the other hand, the designer also needs detailed knowledge about the structure and the behavior of

the thread, therefore the method is not applicable for IP Cores represented by black-box functional blocks. In [92], the authors explore the control software required to support thread switching as well as the requirements and features of context saving and restoring in the FPGA coprocessor context. Similar approach is taken in [7] - each hardware thread is represented by one complicated Finite State Machine (FSM). In case of context switching, the scheduler saves current FSM state together with multiple data registers.

2. *Configuration Port Access* – the thread bitstream is completely downloaded from the FPGA chip and the state information is filtered. In [46], the authors design custom tools for offline bitstream processing. The advantages of the approach are: additional design efforts and information about internal thread behavior are not needed. In [7], the authors additionally compress the bitstream (bitwise XOR) to minimize the size and delay of downloaded data. The method is named "ReadBack technique".

8.3.5 Real-time Support for Reconfigurable Hardware Threads

In the literature, there are two basic approaches (described below) capable to deliver real-time support for software/hardware heterogeneous platforms:

1. *Per-case solutions using Heuristic Algorithms* – many of the proposed algorithms (see Section 8.3.3) support "Commitment Test" - each newly created hardware thread is checked for successful termination before its deadline and critical affects (e.g., delays) on other executing threads. Unfortunately the proposed ideas (heuristic algorithms) are designed only for independent hardware threads with known executing times, therefore they are not applicable for hardware threads with data, resource or communication dependencies.

2. *Complete Solutions on Conventional Reconfigurable Platforms* (e.g., BORPH [93], UltraSonic [40], Hthreads [79]) – none of them supports reconfigurable resource sharing among executing threads. In case reconfigurable area is shared, all possible resource collisions are solved at compile time.

8.3.6 Run-time Creation and Termination of Threads

Currently, all existing proposals (Section 8.3) offer partial solution for scheduling non-preemptive and periodic preemptive only tasks on reconfigurable logic. Therefore, the open question arises: "How to manage creation and termination of data, resource and communication dependent real-time threads?". The topic is still open and it is closely related to Section 8.3.5. It is one of our

Table 8.1: Design problems

Partially [PS] & Completely [CS] Solved Design Problems:
[CS] - Hiding reconfiguration latencies by prefetching, context switching and resource reusage among threads; [27], [56], [82]
[PS] - Optimized inter-thread communication scheme; [59]
[PS] - Real-time thread support by the reconfigurable architecture; [107], [93], [40], [79]
[PS] - Preemptive techniques [context switching] for threads with arbitrary arriving times. Consider inter-thread data dependencies, free reconfigurable area and communication profile; [99], [98], [125], [97]
[PS] - Thread migration between software and hardware; [92], [46], [7]
[PS] - Consider virtualization and protection; [115], [114]
[PS] - Rescheduling of threads, depending on the workload; [32]
[PS] - Run-time creation and termination of threads; [59], [93]

primary objectives to make further investigations in future research. Many of the current projects (e.g., [59], [93]) have run-time creation as a feature, but none of them provides resource sharing and real-time support for data dependent threads. In Table 8.1, we summarize the design problems discussed in this section.

8.3.7 Application Perspective

In this thesis, we consider for hardware acceleration two types of kernels that are part of the user applications and RTOS services, respectively.

As user applications, we target streaming applications such as, but not limited to, MJPEG, JPEG, and H.264. Furthermore, we consider applications and algorithms that were not initially considered as part of the embedded systems, such as Floyd-Warshall algorithm and Conjugate Gradient benchmark part of the *NAS* Parallel Benchmark Suite [15]. In general, the ideas presented in this dissertation can be potentially employed to any computationally intensive kernel in the streaming applications domain.

As part of the RTOS services, we consider three types of kernels. The first one is an advance interrupt management. The second one is the most computationally intensive and variable part of the RTOS scheduling, responsible for checking the status of the tasks. The third one is a service responsible for slack distribution among processor tiles in an MPSoC. In general, we can target

RTOS services with a behaviour similar to UNIX *daemons*. The daemons are background processes, not directly controlled by the users, that execute periodically or wait for an event such as arrival of a packet from the network [102]. We believe that if designers decide to transfer one or multiple daemons in hardware, then the proposed *parallel non-blocking* model can be an excellent candidate for an execution model. On the other hand, we consider that the *parallel non-blocking* execution model has limited applicability for hardware RTOS acceleration in reactive real-time systems [39]. By reactive real-time systems we understand systems that schedule and execute tasks, entirely relying on the presence of external events, e.g., interrupts generated by pressing a button. Both, the daemons and the reactive real-time systems rely on events. The difference between them is in the system response time requirements. The daemons usually register events only, while in reactive real-time systems, an immediate response is required.

8.4 Conclusions

In this chapter, we provided a survey and proposed a taxonomy of existing reconfigurable architectures with respect to their support of multithreading on reconfigurable resources. We identified three main classes – *explicit*, *implicit* and *no* ρMT *support*, each one of them with several sub-categories. We further summarized a number of identified design problems and several research questions, which addressed performance efficient management, mapping, sharing, scheduling and execution of threads on reconfigurable hardware resources. Finally, we marked which of the identified design problems have been partially or completely solved and which research questions remain open.

9

Conclusions and Future Directions

In this chapter, we first provide a summary of the dissertation. Then, we outline the conclusions derived from the experimental results. We conclude the chapter by a discussion on the open research questions and the potential research directions.

9.1 Conclusions

In this dissertation, we targeted the research problem of investigating the behaviour of embedded multicore reconfigurable systems with real-time requirements in a multithreading context. In achieving this goal, we have identified the following sub-problems: facilitate programmability while improving performance as well as preserve predictability.

Our general approach to solve these problems were: 1. Introduce multithreading and multitasking through architectural and microarchitectural augmentations, synchronization primitives, and execution models. 2. Hardware acceleration of the most variable and computation intensive kernels from user applications and RTOS services. Throughout the experiments we demonstrated that the newly introduced processor–coprocessor programming models and coprocessors, which exercise those models are applicable for different application programming paradigms, architectures, and problems in the domain of embedded architectures.

Bellow, we summarized the targeted research problems in this dissertation and we provided a short discussion how we addressed each one of them:

- **Improve performance** – We improved the system performance by proposing new parallel execution models for processor–coprocessor execution. With the help of the new execution models, we executed one of

the most time-consuming and time-variable parts of the RTOS in hardware. More precisely, we implemented as a reconfigurable coprocessors the following RTOS services: Thread Interrupt State Controller (TISC) and Hardware Task-Status Manager (HWTSM).

With the help of the TISC, we reduced the thread synchronization cost. More specifically, the TISC acts as a barrier that signals back the processor only after all identified application tasks executed on reconfigurable coprocessors are completed. We assumed that the reconfigurable coprocessors were executed in processor–coprocessor sequential execution model. A comparison of the experimental results to the other state of the art proposals suggested that our approach demonstrated the best performance-portability and performance-flexibility characteristics.

With the help of the HWTSM, we ported one of the most time-consuming RTOS kernel services in hardware. The HWTSM implements part of the RTOS scheduler functionality. More precisely, the HWTSM is responsible for checking the status of the application tasks. In such a way, we reduce the RTOS cost, which leaded to improved application performance.

- **Preserve predictability** – Usually the predictability is associated with guaranteeing the worst-case bounds of the application and RTOS execution. We achieved our goal by proposing new parallel processor–coprocessor execution model and implementing one of the most variable and time consuming kernels in hardware. In particular, we consider part of the RTOS services responsible for checking the status of the tasks, called HWTSM.

The proposed processor-coprocessor execution models are general solution for various problems in the real-time embedded systems. To prove generality, we apply the processor-coprocessor execution models to two common problems in the embedded domain, i.e., guarantee composability and reduce energy consumption. Composability means that the behaviour of an application, including its timing, is independent of the presence or absence of any other application. We addressed composability and energy consumption as follows:

- **Preserve composability** – We preserved the composability, i.e., application virtualization, with the help of new processor–coprocessor parallel execution model in the context of reconfigurable systems. More specifically, the coprocessor runs continuously and it does not need to

Table 9.1: Addressed design problems

	Partially [PS] & Completely [CS] Solved design problems:
	[CS] - Hiding reconfiguration latencies by prefetching, context switching and resource reusage among threads; [27], [56], [82]
✓	[PS] - Optimized inter-thread communication scheme; [59]
✓	[PS] - Real-time thread support by the reconfigurable architecture; [107], [93], [40], [79]
	[PS] - Preemptive techniques [context switching] for threads with arbitrary arriving times. Consider inter-thread data dependencies, free reconfigurable area and communication profile; [99], [98], [125], [97]
✓	[PS] - Thread migration between software and hardware; [92], [46], [7]
	[PS] - Consider virtualization and protection; [115], [114]
	[PS] - Rescheduling of threads, depending on the workload; [32]
	[PS] - Run-time creation and termination of threads; [59], [93]
✓	[PS] - Hardware scheduler agnostic to the employed embedded GPP;
✓	[PS] - System performance evaluation parameters;

be restarted every time when the processor needs the coprocessor. Furthermore, the processor can request the status of the reconfigurable coprocessor at any time. Last but not leased, independently of the current status of the reconfigurable processor, its response time is always constant. In such a way, a coprocessor can be shared among multiple applications without introducing any dependability among the applications. We examined the execution of the proposed processor–coprocessor execution model with RTOS services.

- **Reduce energy consumption** – We reduced the energy consumption by proposing a framework that distributes the slack information among the processor tiles in an MPSoC. We employ the extra slack information for Dynamic Voltage Frequency Scaling (DVFS). As the experimental results suggested, our inter-tile technique dramatically reduces average processor frequency level and energy consumption at the cost of negligible software and hardware costs.

In Table 9.1, we mark with ticks the addressed Design Problems in this dissertation. In essence, Table 9.1 is an updated version of Table 8.1 from Chapter 8. In this dissertation, we address the problem of "Optimized inter-thread communication scheme" by proposing the TISC CCU. The TISC CCU al-

lows hardware synchronization of multiple hardware tasks executed in parallel. Since one of our target problems is to preserve the predictability, we also address the problem "Real-time support by the reconfigurable architecture". The proposed programming model and execution paradigms also allow easy "Thread migration between software and hardware". Last but not least, we also partially solve two new design problems, namely: "Hardware scheduler agnostic to the employed embedded GPP" and proposal for "System performance evaluation parameters". Concerning the first new design problem, we transfer to hardware part of the RTOS scheduler (i.e., HWTSM CCU). The HWTSM CCU is responsible for checking the status of the tasks and its design is independent of the employed GPP. Concerning the second new design problem, we introduce performance-portability and performance-flexibility characteristics to compare our TISC CCU with the other state of the art projects.

9.2 Future Research Directions

One of the direct gains from employing reconfigurable multithreading architecture would be the capability for time efficient run-time creation, termination and management of multiple threads sharing the reconfigurable resources without critically affecting (delaying) each other. Possible future research could extend the functionality and overcome some limitations providing for example:

1. Real-time and runtime support of multiple hardware threads through architecture agnostic hardware scheduler. It could support run-time creation and termination of multiple threads mapped into reconfigurable logic and hardware system implementation. The compiler would be only responsible for inter-thread optimizations. The hardware scheduler would manage intra-thread optimizations;

2. More sophisticated scheduling policies capable to fairly distribute resources among multiple resource-dependent hardware threads. Introduction of a metric evaluating the resource distribution and potential thread starvation.

3. Hiding of reconfiguration latencies and efficient thread-preemption and migration model with estimation of performance costs. For periodic and sporadic threads, the migration might take place right after the end of the current iteration.

4. Consider other RTOS services for hardware acceleration. In order to achieve better results, we suggest the processor–coprocessor parallel execution models

to be employed whenever possible. Prospective experiments over various platforms and programming models will indicate further possible improvements of the overall system performance.

Bibliography

[1] http://www.hartes.org.

[2] http://www.sgi.com/products/servers/altix/, 2003.

[3] http://ramp.eecs.berkeley.edu/, 2006.

[4] The convey HC-1 computer, architecture overview (white paper)- http://www.conveycomputer.com, 2008.

[5] http://ecos.sourceware.org/, 2008.

[6] http://www.hp.com/products/servers/platforms/, 2009.

[7] AHMADINIA, A., BOBDA, C., KOCH, D., MAJER, M., AND TEICH, J. Task scheduling for heterogeneous reconfigurable computers. In *Symp. on Integrated Circuit and Systems Design (SBCCI)* (2004), pp. 22–27.

[8] AHMADINIA, A., BOBDA, C., AND TEICH, J. Online placement for dynamically reconfigurable devices. *Int'l Journal of Emerging Sciences (IJES) 1*, 3/4 (2005), 165–178.

[9] AKESSON, B., MOLNOS, A., HANSSON, A., AMBROSE ANGELO, J., AND GOOSSENS, K. Composability and predictability for independent application development, verification, and execution. In *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds., Circuits and Systems. Springer, 2010, ch. 2.

[10] ALLE, M., VARADARAJAN, K., RAMESH, R. C., NIMMY, J., FELL, A., RAO, A., NANDY, S. K., AND NARAYAN, R. Synthesis of application accelerators on runtime reconfigurable hardware. In *Proc. Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)* (2008), pp. 13–18.

[11] AMBROSE, J., MOLNOS, A., NELSON, A., GOOSSENS, K., COTOFANA, S., AND JUURLINK, B. Composable local memory organisation for streaming applications on embedded MPSoCs. In *Proc. Int'l. Conf. Computing Frontiers (CF)* (2011), ACM, pp. 23:1–23:2.

[12] ANDREWS, D., NIEHAUS, D., JIDIN, R., FINLEY, M., PECK, W., FRISBIE, M., ORTIZ, J., KOMP, E., AND ASHENDEN, P. Programming models for hybrid FPGA-CPU computational components:a missing link. *IEEE Micro 24* (2004), 42–53.

[13] ANDREWS, D., SASS, R., ANDERSON, E., AGRON, J., ANDJ. STEVENS, W. P., BAIJOT, F., AND KOMP, E. Achieving programming model abstractions for reconfigurable computing. *IEEE Transactions on VLSI systems 16* (January 2008), 34–44.

[14] ANGERMEIER, J., AND TEICH, J. Heuristics for Scheduling Reconfigurable Devices with Considerationof Reconfiguration Overheads. In *Proc. Int'l Reconfigurable Architectures Workshop (RAW)* (Miami, Florida, 2008).

[15] BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. K. The NAS parallel benchmarks. Tech. rep., The International Journal of Supercomputer Applications, 1991.

[16] BAUER, L., SHAFIQUE, M., KREUTZ, S., AND HENKEL, J. Run-time system for an extensible embedded processor with dynamic instruction set. In *Proc. Int'l Conf on Design, Automation & Test in Europe (DATE)* (2008), pp. 752–757.

[17] BAZARGAN, K., KASTNER, R., AND SARRAFZADEH, M. Fast template placement for reconfigurable computing systems. *IEEE Design & Test of Computers 17*, 1 (2000), 68–83.

[18] BIC, L., AND LEE, C. A data-driven model for a subset of logic programming. *ACM Trans. Program. Lang. Syst. 9* (October 1987), 618–645.

[19] BILSEN, G., ENGELS, M., LAUWEREINS, R., AND PEPERSTRAETE, J. Cycle-static dataflow. *Signal Processing, IEEE Transactions on 44*, 2 (1996), 397–408.

[20] BONDHUGULA, U., DEVULAPALLI, A., FERNANDO, J., WYCKOFF, P., AND SADAYAPPAN, P. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *Proc. Int'l Symp. Parallel & Distributed Processing (IPDPS)* (2006), pp. 90–100.

[21] BUI, D. N., PATEL, H. D., AND LEE, E. A. Deploying hard real-time control software on chip-multiprocessors. In *Proc. Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTSCA)* (2010), pp. 283–292.

[22] BUTTLAR, D., FARRELL, J., AND NICHOLS, B. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.

[23] CARTA, S., ALIMONDA, A., PISANO, A., ACQUAVIVA, A., AND BENINI, L. A control theoretic approach to energy-efficient pipelined computation in MPSoCs. *ACM Trans. Embedded Comput. Syst 6*, 4 (2007).

[24] CHANDRA, R., DAGUM, L., KOHR, D., MAYDAN, D., MCDONALD, J., AND MENON, R. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[25] CHANDRA, S., REGAZZONI, F., AND LAJOLO, M. Hardware/software partitioning of operating systems: a behavioral synthesis approach. In *GLSVLSI* (2006), ACM, pp. 324–329.

[26] CHAVES, R. *Secure Computing on Reconfigurable Systems*. PhD thesis, TU Delft, December 2007.

[27] CHEN, Y., AND CHEN, S. Y. Cost-driven hybrid configuration prefetching for partial reconfigurable coprocessor. In *Proc. Int'l Symp. Parallel & Distributed Processing (IPDPS)* (2007), IEEE Press, pp. 1–8.

[28] COMPTON, K., AND HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys 34*, 2 (2002), 171–210.

[29] DANNE, K., AND PLATZNER, M. A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2005), IEEE Press, pp. 568–573.

[30] DASDAN, A. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM TODAES 9*, 4 (2004), 385–418.

[31] DIESSEL, O., AND WIGLEY, G. B. Opportunities for operating systems research in reconfigurable computing. Tech. rep., 1999.

[32] DITTMANN, F. *Methods to Exploit Reconfigurable Fabrics - Making Reconfigurable Systems Mature*. PhD thesis, University of Paderborn, 2007.

[33] GIBELING, G., SCHULTZ, A., AND ASANOVIC, K. The RAMP architecture & description language. In *Proc. Int'l Workshop on Architecture Research using FPGA Platforms (WARFP)* (2006).

[34] GLOSSNER, J., SCHULTE, M., MOUDGILL, M., IANCU, D., JINTURKAR, S., RAJA, T., NACER, G., AND VASSILIADIS, S. Sandblaster low-power multithreaded SDR baseband processor. In *Proc. Workshop on Application Specific Processors (WASP)* (2004), pp. 53–58.

[35] GOOSSENS, K., AZEVEDO, A., CHANDRASEKAR, K., GOMONY, M. D., GOOSSENS, S., KOEDAM, M., LI, Y., MIRZOYAN, D., MOLNOS, A., BEYRANVAND NEJAD, A., NELSON, A., AND SINHA, S. Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow. *ACM SIGBED Rev. 10*, 3 (2013), 23–34.

[36] HANSSON, A., EKERHULT, M., MOLNOS, A., MILUTINOVIC, A., NELSON, A., AMBROSE, J., AND GOOSSENS, K. Design and implementation of an Operating System for composable processor sharing. *MICPRO 35* (2011), 246–260.

[37] HANSSON, A., GOOSSENS, K., BEKOOIJ, M., AND HUISKEN, J. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst. 14*, 1 (2009), 2:1–2:24.

[38] HANSSON, A., WIGGERS, M., MOONEN, A., GOOSSENS, K., AND BEKOOIJ, M. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques 3*, 5 (2009), 398–412.

[39] HAREL, D., AND PNUELI, A. On the development of reactive systems. Springer-Verlag New York, Inc., pp. 477–498.

[40] HAYNES, S. D., EPSOM, H. G., COOPER, R. J., AND MCALPINE, P. L. UltraSONIC: A reconfigurable architecture for video image processing. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2002), Springer-Verlag, pp. 482–491.

[41] HEYSTERS, P. M. Coarse-grained reconfigurable computing for power aware applications. In *Proc. Int'l Conf on Engineering of Reconfigurable Systems and Algorithms(ERSA)* (2006), pp. 272–280.

[42] HUANG, P., MOREIRA, O., GOOSSENS, K., AND MOLNOS, A. Throughput-constrained voltage and frequency scaling for real-time heterogeneous multiprocessors. In *Proc. of SAC* (2013), ACM, pp. 1517–1524.

[43] J.CUI, Q.DENG, HE, X., AND Z.GU. An efficient algorithm for online management of 2D area of partially reconfigurable FPGAs. In *Proc. Int'l Conf on Design, Automation & Test in Europe (DATE)* (2007), pp. 129–134.

[44] JEJURIKAR, R., AND GUPTA, R. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proc. Design Automation Conf. (DAC)* (2005), pp. 111–116.

[45] KAHN, G. The semantics of a simple language for parallel programming. In *IFIP Congress* (1974), North-Holland Publishing Co, pp. 471–475.

[46] KALTE, H., AND PORRMANN, M. Context saving and restoring for multitasking in reconfigurable systems. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2005), IEEE Press, pp. 223–228.

[47] KOHOUT, P., GANESH, B., AND JACOB, B. Hardware support for real-time operating systems. In *Proc. Int'l Conf on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2003), ACM Press, pp. 45–51.

[48] KOPETZ, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2011.

[49] KUACHAROEN, P., SHALAN, M. A., AND III, V. J. M. A configurable hardware scheduler for real-time systems. In *Proc. Int'l Conf on Engineering of Reconfigurable Systems and Algorithms (ERSA)* (2003), CSREA Press, pp. 96–101.

[50] KUZMANOV, G. K., VASSILIADIS, S., AND VAN EIJNDHOVEN, J. T. J. Hardwired MPEG-4 repetitive padding. *IEEE Transactions on Multimedia 7* (2005), 261–268.

[51] KWONG KWOK, Y., AHMAD, I., AND AHMAD, I. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 7* (1996), 506–521.

[52] KYRIACOU, C., AND EVRIPIDOU, P. Communication assist for data driven multithreading. In *PCI* (Berlin, Heidelberg, 2003), Springer-Verlag, pp. 351–367.

[53] LANGE, H., AND KOCH, A. Architectures and execution models for hardware/software compilation and their system-level realization. *IEEE Transactions on Computers 59* (2010), 1363–1377.

[54] LEE, E. A., AND PARKS, T. Dataflow process networks. In *Proceedings of the IEEE* (1995), pp. 773–799.

[55] LEVY, H. M., LO, J. L., STAMM, R. L., EGGERS, S. J., EMER, J. S., AND TULLSEN, D. M. Simultaneous multithreading: A foundation for next-generation processors. In *IEEE Micro* (1997), pp. 12–18.

[56] LI, Z., AND HAUCK, S. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proc. Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (2002), pp. 187–195.

[57] LÜBBERS, E., AND PLATZNER, M. ReconOS: An RTOS supporting hard- and software threads. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2007), pp. 441–446.

[58] LÜBBERS, E., AND PLATZNER, M. ReconOS: Multithreading programming for reconfigurable computers. *ACM Trans. on Embedded Computing Sys. 9* (2009), 1–33.

[59] MAJER, M., TEICH, J., AHMADINIA, A., AND BOBDA, C. The Erlangen Slot Machine: A dynamically reconfigurable fpga-based computer. *VLSI Signal Processing 47*, 1 (2007), 15–31.

[60] MAMADI, S. *Reconfigurable Multi Processors for Programmable Communication Systems*. PhD thesis, Wisconsin-Madison, 2006.

[61] MAMIDI, S., SCHULTE, M., IANCU, D., AND GLOSSNER, J. Architecture support for reconfigurable multithreaded processors in programmable communication systems. In *Proc. Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)* (2007), IEEE Press, pp. 320–327.

[62] MARCONI, T., BERTELS, K., LU, Y., AND GAYDADJIEV, G. Online hardware task scheduling and placement algorithm on partiallyreconfigurable devices. In *Proc. Int'l Conf on Architecture of Computing Systems (ARCS)* (2008), pp. 306–311.

[63] MARESCAUX, T., MIGNOLET, J.-Y., BARTIC, A., MOFFAT, W., VERKEST, D., VERNALDE, S., AND LAUWEREINS, R. Networks-on-chip as hardware components of an OS for reconfigurable systems. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2003), vol. 2778 of *LNCS*, Springer, pp. 595–605.

[64] MARESCAUX, T., NOLLET, V., MIGNOLET, J.-Y., BARTIC, A., MOFFAT, W., AVASARE, P., COENE, P., VERKEST, D., VERNALDE, S., AND LAUWEREINS, R.

Run-time support for heterogeneous multitasking on reconfigurable SoCs. *Integr. VLSI J. 38(1)* (2004), 107–130.

[65] MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *Proc. Int'l Conf. on Field Programmable Technology (FPT)* (2002), pp. 166–173.

[66] MOLNOS, A., BEYRANVAND NEJAD, A., NGUYEN, B. T., COTOFANA, S., AND GOOSSENS, K. Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms. In *Proc. of SCOPES* (2012), ACM, pp. 13–21.

[67] MOLNOS, A., AND GOOSSENS, K. Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors. In *Proc. Int'l Conf. on Digital System Design (DSD)* (2009), pp. 409–418.

[68] MOREIRA, O., BASTEN, T., GEILEN, M., AND STUIJK, S. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Transactions on Computers 59*, 2 (2010), 188–201.

[69] NAKANO, T., UTAMA, A., ITABASHI, M., SHIOMI, A., AND IMAI, M. Hardware implementation of a real-time operating system. In *12th TRON Project International Symposium* (1995), pp. 34–42.

[70] NEJAD, A. B., MOLNOS, A., AND GOOSSENS, K. A unified execution model for data-driven applications on a composable mpsoc. In *Proc. Int'l Conf. on Digital System Design (DSD)* (2011), pp. 818–822.

[71] NELSON, A., MOREIRA, O., MOLNOS, A., STUIJK, S., NGUYEN, B. T., AND GOOSSENS, K. Power minimisation for real-time dataflow applications. In *Proc. Int'l Conf. on Digital System Design (DSD)* (2011), pp. 117–124.

[72] NGUYEN, B. Task scheduling methods for composable and predictable MPSoCs. Master's thesis, TUD, October 2010.

[73] NIEUWLAND, A., KANG, J., GANGWAL, O. P., SETHURAMAN, R., BUSÁ, N., GOOSSENS, K., LLOPIS, R. P., AND LIPPENS, P. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems 7*, 3 (2002), 233–270.

[74] NOGUERA, J., AND BADIA, R. M. Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. *Trans. on Embedded Computing Sys. 3*, 2 (2004), 385–406.

[75] NOLLET, V., COENE, P., VERKEST, D., VERNALDE, S., AND LAUWEREINS, R. Designing an Operating System for a heterogeneous reconfigurable SoC. In *Proc. Int'l Symp. Parallel & Distributed Processing (IPDPS)* (2003), pp. 174–180.

[76] NOLLET, V., MIGNOLET, J.-Y., BARTIC, A., VERKEST, D., VERNALDE, S., AND LAUWEREINS, R. Hierarchical run-time reconfiguration managed by an Operating System for reconfigurable systems. In *Engineering of Reconfigurable Systems and Algorithms* (2003), CSREA Press, pp. 81–87.

[77] OBERMAISSER, R., SALLOUM, C. E., HUBER, B., AND KOPETZ, H. The time-triggered system-on-a-chip architecture. In *Proc. Int'l Symp. on Industrial Electronics (ISIE)* (2008), pp. 1941–1947.

[78] PANAINTE, E. M. *The Molen Compiler for Reconfigurable Architectures*. PhD thesis, TU Delft, 2007.

[79] PECK, W., ANDERSON, E., AGRON, J., STEVENS, J., BAIJOT, F., AND ANDREWS, D. HTHREADS: a computational model for reconfigurable devices. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2006), pp. 885–888.

[80] RAO, A., ALLE, M., V, S., SHAIK, R., CHOWHAN, R., SANKARAIAH, S., MANTHA, S., NANDY, S. K., AND NARAYAN, R. An input triggered polymorphic ASIC for H.264 decoding. In *Proc. Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)* (2009), IEEE Computer Society, pp. 106–113.

[81] RESANO, J., MOZOS, D., AND CATTHOOR, F. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In *Proc. Int'l Conf on Design, Automation & Test in Europe (DATE)* (2005), pp. 106–111.

[82] RESANO, J., MOZOS, D., VERKEST, D., AND CATTHOOR, F. A reconfiguration manager for dynamically reconfigurable hardware. *IEEE Design & Test of Computers 22*, 5 (2005), 452–460.

[83] RHOADS, S. http://www.opencores.org/project,plasma, 2011.

[84] ROLDAO, A., AND CONSTANTINIDES, G. A. A high throughput fpga-based floating point conjugate gradient implementation for dense matrices. *ACM Trans. Reconfigurable Technol. Syst. 3*, 1 (2010), 1–19.

[85] RUGGIERO, M., ACQUAVIVA, A., BERTOZZI, D., AND BENINI, L. Application-specific power-aware workload allocation for voltage scalable MPSoC platforms. In *Proc. of ICCD* (2005), pp. 87–93.

[86] RUPNOW, K., FU, W., AND COMPTON, K. Block, drop or roll(back): Alternative preemption methods for rh multi-tasking. *Field-Programmable Custom Computing Machines (FCCM)* (2009), 63–70.

[87] SAEZ, S., VILA, J., CRESPO, A., AND GARCIA, A. A hardware scheduler for complex real-time systems. In *Proc. Int'l Symp. on Industrial Electronics (ISIE)* (1999), vol. 1, pp. 43–48.

[88] SATRAWALA, A., VARADARAJAN, K., LIE, M., NANDY, S., AND NARAYAN, R. Redefine: Architecture of a soc fabric for runtime composition of computation structures. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2007), pp. 558–561.

[89] SENO, K., AND YAMAZAKI, M. Virtual mobile engine (VME) LSI that "changes its spots" achievies ultralow power and diverse functionality. *CX-News-42 (http://www.sony.com)* (2005).

[90] SHABBIR, A., KUMAR, A., STUIJK, S., MESMAN, B., AND CORPORAAL, H. CA-MPSoC: An automated design flow for predictable multi-processor architectures for multiple applications. *J. Syst. Archit. 56*, 7 (2010), 265–277.

[91] SIMA, M., VASSILIADIS, S., COTOFANA, S. D., VAN EIJNDHOVEN, J. T. J., AND VISSERS, K. A. Field-programmable custom computing machines - a taxonomy. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2002), pp. 79–88.

[92] SIMMLER, H., AND LEVINSON, L. Multitasking on FPGA coprocessors. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2000), Springer-Verlag, pp. 121–130.

[93] SO, H. K.-H., AND BRODERSEN, R. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems 7*, 2 (2008), 1401–1407.

[94] SO, H. K.-H., AND BRODERSEN, R. W. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, 2007.

[95] SRIRAM, S., AND BHATTACHARYYA, S. S. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[96] STEFAN, R., MOLNOS, A., AND GOOSSENS, K. dAElite: A TDM NoC supporting QoS, multicast, and fast connection set-up. *IEEE Transactions on Computers 99* (2012).

[97] STEIGER, C., WALDER, H., AND PLATZNER, M. Heuristics for online scheduling real-time tasks to partially reconfigurable devices. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2003), pp. 575–584.

[98] STEIGER, C., WALDER, H., AND PLATZNER, M. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Computers 53*, 11 (2004), 1393–1407.

[99] STEIGER, C., WALDER, H., PLATZNER, M., AND THIELE, L. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *IEEE Real-Time Systems Symposium (RTSS)* (2003), IEEE Computer Society, pp. 224–235.

[100] STUTZ, M. Get started with gawk: Awk language fundamentals. Tech. rep., IBM, 2006.

[101] TAN, Z. Multithreaded sparc v8 functional model for ramp gold. In *Presentation - http://ramp.eecs.berkeley.edu/* (2008), p. 15.

[102] TANENBAUM, A. S., AND WOODHULL, A. S. *Operating Systems Design and Implementation (3rd Edition)*. Prentice Hall, Jan. 2006.

[103] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. P. Streamit: A language for streaming applications. In *In Proc. of the Int'l Conf. on Compiler Construction (CC)* (2002), Springer-Verlag, pp. 179–196.

[104] THOMA, F., KUHNLE, M., BONNOT, P., PANAINTE, E. M., BERTELS, K., GOLLER, S., SCHNEIDER, A., GUYETANT, S., SCHULER, E., MULLER-GLASER, K., AND BECKER, J. Morpheus: Heterogeneous reconfigurable computing. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2007), pp. 409–414.

[105] TUMEO, A., BRANCA, M., CAMERINI, L., MONCHIERO, M., PALERMO, G., FERRANDI, F., AND SCIUTO, D. An interrupt controller for FPGA-based Multiprocessors. In *Proc. Int'l Conf on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS-VII)* (2007), pp. 82–87.

[106] TUMEO, A., MONCHIERO, M., PALERMO, G., FERRANDI, F., AND SCIUTO, D. Lightweight DMA management mechanisms for multiprocessors on FPGA. In *Proc. Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)* (2008), pp. 275–280.

[107] UHRIG, S., MAIER, S., KUZMANOV, G. K., AND UNGERER, T. Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. In *Proc. Int'l Reconfigurable Architectures Workshop (RAW)* (2006), pp. 209–217.

[108] UHRIG, S., MAIER, S., AND UNGERER, T. Toward a processor core for real-time capable autonomic systems. In *IEEE Symp. on Signal Processing and Information Technology (ISSPIT)* (2005), pp. 19–22.

[109] UNGERER, T., ROBIČ, B., AND ŠILC, J. A survey of processors with explicit multithreading. *ACM Comput. Surv. 35*, 1 (Mar. 2003), 29–63.

[110] VASSILIADIS, S., WONG, S., AND COTOFANA, S. D. The MOLEN $\mu\rho$-coded processor. In *Proc. Int'l Conf. Field Programmable Logic and Applications (FPL)* (2001), Springer-Verlag (LNCS) Vol. 2147, pp. 275–285.

[111] VASSILIADIS, S., WONG, S., GAYDADJIEV, G. N., BERTELS, K., KUZMANOV, G. K., AND PANAINTE, E. M. The Molen polymorphic processor. *IEEE Transactions on Computers 53* (2004), 1363–1375.

[112] WALDER, H., AND PLATZNER, M. Reconfigurable hardware Operating Systems: From design concepts to realizations. In *Proc. Int'l Conf on Engineering of Reconfigurable Systems and Algorithms (ERSA)* (2003), CSREA Press, pp. 284–287.

[113] WALDER, H., STEIGER, C., AND PLATZNER, M. Fast online task placement on FPGAs: Free space partitioning and 2D-Hashing. In *Proc. Int'l Symp. Parallel & Distributed Processing (IPDPS)* (2003), pp. 178–178.

[114] WALLNER, S. A reconfigurable multi-threaded architecture model. In *APCSAC* (2003), vol. 2823, Springer, pp. 193–207.

[115] WALLNER, S. Micro-task processing in heterogeneous reconfigurable systems. *J. Comput. Sci. Technol 20*, 5 (2005), 624–634.

[116] WANG, Y., LIU, H., LIU, D., QIN, Z., SHAO, Z., AND SHA, E. H.-M. Overhead-aware energy optimization for real-time streaming applications on multiprocessor System-on-Chip. *ACM TODAES 16*, 2 (2011), 14.

[117] WIANGTONG, T., CHEUNG, P. Y. K., AND LUK, W. Cluster-driven hardware/software partitioning and scheduling approach for a reconfigurable computer system. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2003), pp. 1071–1074.

[118] WIGLEY, G. B., AND KEARNEY, D. A. The first real operating system for reconfigurable computers. In *Proc. Annual Computer Security Applications Conf. (ACSAC)* (Jan. 2000), IEEE Computer Society Press, pp. 129–136.

[119] WU, K., KANSTEIN, A., MADSEN, J., AND BEREKOVIC, M. MT-ADRES: Multithreading on coarse-grained reconfigurable architecture. In *Proc. Int'l Symp. on Applied Reconfigurable Computing (ARC)* (2007), vol. 4419 of *LNCS*, Springer, pp. 26–38.

[120] ZAMORA, N. H., HU, X., AND MARCULESCU, R. System-level performance/power analysis for platform-based design of multimedia applications. *ACM TODAES 12*, 1 (2007), 2:1–2:29.

[121] ZAYKOV, P. G., AND KUZMANOV, G. K. Architectural support for multithreading on reconfigurable hardware. In *Proc. Int'l. Symp. on Applied Reconfigurable Computing (ARC)* (2011), pp. 363–374.

[122] ZAYKOV, P. G., KUZMANOV, G. K., AND GAYDADJIEV, G. N. Reconfigurable multithreading architectures: A survey. In *Proc. Int'l Workshop on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS-IX)* (July 2009), pp. 263–274.

[123] ZHOU, B., QUI, W., AND PENG, C.-L. An operating system framework for reconfigurable systems. In *Proc. Int'l Conf on Computer and Information Technology (CIT)* (2005), IEEE Computer Society, pp. 781–787.

[124] ZHOU, B., QUI, W., AND PENG, C.-L. Shum-ucos: A rtos using multi-task model to reduce migration cost between sw/hw tasks. In *Proc. Int'l Conf. on Computer Supported Cooperative Work in Design (CSCWD)* (2005), pp. 984–989.

[125] ZHOU, X., WANG, Y., HUANG, X.-Z., AND PENG, C.-L. On-line scheduling of real-time tasks for reconfigurable computing system. In *Proc. Int'l Conf. on Field Programmable Technology (FPT)* (2006), pp. 57–64.

[126] ZHOU, X., WANG, Y., HUANG, X.-Z., AND PENG, C.-L. Fast on-line task placement and scheduling on reconfigurable devices. In *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL)* (2007), IEEE Computer Society, pp. 132–138.

List of Publications

International Journals

1. P. G. Zaykov and G. K. Kuzmanov, **Multithreading on Reconfigurable Hardware: an Architectural Approach**, Microprocessors and Microsystems (MICPRO), Vol. 36, Issue 8, 2012, pp. 695–704

International Conference Proceedings

1. P. G. Zaykov and G. K. Kuzmanov and A. M. Molnos and K. G. W. Goossens, **Hardware Task-Status Manager for RTOS with FIFO Communication**, To appear in Proc. Int'l Conf. on ReConFigurable Computing and FPGAs (ReConFig), 2014

2. P. G. Zaykov, A. M. Molnos, G. K. Kuzmanov and K. G. W. Goossens, **Run-time slack distribution for real-time data-flow applications on embedded MPSoC**, Proc. Int'l. Euromicro Conference on Digital System Design (DSD), 2013, pp. 39–47

3. P. G. Zaykov and G. K. Kuzmanov, **Architectural Support for Multithreading on Reconfigurable Hardware**, Proc. Int'l. Symp. on Applied Reconfigurable Computing (ARC), 2011, pp. 363–374

4. P. G. Zaykov, G. K. Kuzmanov and G. N. Gaydadjiev, **Reconfigurable Multithreading Architectures: A Survey**, Proc. Int'l Workshop on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS IX), 2009, pp. 263–274

Technical Reports

1. P. G. Zaykov, G. K. Kuzmanov and G. N. Gaydadjiev, **State-of-the-art Reconfigurable Multithreading Architectures**, Technical Report - CE-TR-2009-02, 2009

Non Peer-Reviewed Conference Proceedings

1. P. G. Zaykov and G. K. Kuzmanov, **Architectural Support for Concurrency on Reconfigurable Systems**, Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), 2011, pp. 119–126

2. P. G. Zaykov, A. M. Molnos, G. K. Kuzmanov and K. G. W. Goossens, **Dynamic Communication of Slacks through Timestamps in Multiprocessor Systems**, NWO-ICT.OPEN, Poster, 2011

Other publications, not directly related to this dissertation:

International Conference Proceedings

1. P. Zaykov, **MIMD implementation with PicoBlaze microprocessor using MPI functions**, Proc. Int'l. Conf. on Computer systems and technologies, 2007, pp. 4:1–4:7

2. P. Zaykov, P. Manoilov, G. Kamburov, P. Borovska **Parallel architecture implemented in FPGA based on message passing system**, Proc. Int'l Conf. on Computer Science, 2006, Book 1, pp. 115–120

3. G. Kamburov, P. Manoilov, P. Zaykov, P. Borovska **Parallel architecture implemented in FPGA based on shared memory system**, Proc. Int'l Conf. on Computer Science, 2006, Book 1, pp. 121–126

Samenvatting

In dit proefschrift beschouwen we het probleem van het efficiënt uitvoeren (met betrekking tot performance) van meerdere threads op heterogene embedded multicore systemen. Deze systemen hebben eisen wat betreft het real-time gedrag, en bestaan uit processor tiles (tegels) met daarop een General Purpose Processor (GPP), lokaal geheugen, en één of meer co-processors uitgevoerd in herconfigureerbare logica ((e)(FPGA). We verbeteren de performance van het systeem door een combinatie van twee methoden. Ten eerste buiten we het beschikbare parallelisme uit door middel van het uitvoeren van multithreaded programma's. Ten tweede gebruiken we hardware versnellers voor de kernels die de meeste rekenkracht benodigen. Om precies te zijn is onze wetenschappelijke aanpak als volgt: we categoriseren de bestaande modellen voor uitvoering van programma's vanuit het perspectief van de processor-coprocessor synchronisatie, en introduceren nieuwe parallelle modellen voor het uitvoeren van deze programma's. Daarna beschrijven we een architecturale abstractie van deze modellen alsmede een paradigma voor het programmeren dat deze modellen omschrijft en benut. Verder stellen we een micro-architecturale ondersteuning voor de genoemde modellen voor. De functionaliteit van deze micro-architecturale uitbreidingen is ingekapseld in een nieuwe herconfigureerbare co-processor, genaamd de Thread Interrupt State Controller (TISC). Om de performance van het gehele systeem te verbeteren benutten we de voorgestelde modellen voor de overdracht van tijd-variabele en tijd-consumerende Real-Time Operating Systems (RTOS) en applicatie kernels vanuit de software, d.w.z. uitgevoerd op de GPP's, naar de hardware, d.w.z. uitgevoerd op de herconfigureerbare co-processors. We refereren aan de herconfigureerbare co-processor als de Hardware Task Status Manager (HWTSM). Vanwege de eigenschappen van de nieuw gentroduceerde modellen voor uitvoering zoals parallelle uitvoering en constante antwoordtijden behouden we de voorspelbaarheid en componeerbaarheid (composability) op het niveau van de applicatie. Tenslotte introduceren we een framework voor de distributie van informatie over slack (ongebruikte processortijd) tussen processor tiles. In het voorgestelde framework gebruiken we één van de nieuw gentroduceerde modellen voor uitvoering op parallelle processors/co-processors. We refereren aan de nieuwe herconfigureerbare co-processor als RS. We gebruiken de extra informatie over slack die wordt verkregen door ons framework voor Dynamic Voltage Frequency Scaling, wat resulteert in een reductie van het algemene energieverbruik.

Gebaseerd op de beschikbare experimentele resultaten met zowel synthetische als echte applicaties versnellen we het systeem met een factor van maximaal 19.6 met hulp van de Thread Interrupt State Controller. Verder reduceren we de kosten van het RTOS met hulp van de Hardware Task Status Manager, wat resulteert in een bijkomende versnelling van de applicatie van maximaal 13.3%. Ook reduceren we het energieverbruik van het systeem met maximaal 56.7% vergeleken met de huidige state-of-the-art, met hulp van het framework voor de distributie van informatie over slack tussen tiles.

In het geheel genomen leiden onze bijdragen tot een verbetering van de performance van het systeem terwijl voorspelbaarheid en componeerbaarheid behouden blijven, dit alles met gereduceerd energieverbruik.

Curriculum Vitae

Pavel G. Zaykov was born on November 9, 1982 in Plovdiv, Bulgaria. From 2001 to 2005 he studied at the Department of Computer Systems and Technologies at the Technical University of Sofia, Plovdiv, Bulgaria.

He received his Bachelor's degree in 2005, and continued in the same year to study at the Department of Computer Systems and Technologies, Technical University of Sofia, Sofia, Bulgaria. He received his Master's degree in 2007.

At the end of 2007, he joined the Computer Engineering laboratory at Delft University of Technology in The Netherlands, and he started his PhD study, working on architectural support for multithreading on reconfigurable logic. His research was funded by the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) and by the COmmunication-centric heterogeneous Multi-Core ArchitectureS (COMCAS) project. The results of his work were presented in the current dissertation.

Pavel's research interests include Multithreading Architectures, Reconfigurable Computing, Embedded System Design, Real-Time Operating Systems, and Multiprocessor-Systems-on-Chip.