Run-time management for future MPSoC platforms

Citation for published version (APA):
Nollet, V. (2008). *Run-time management for future MPSoC platforms*. [Phd Thesis 2 (Research NOT TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR633897

DOI:
10.6100/IR633897

Document status and date:
Published: 01/01/2008

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Run-Time management for future MPSoC platforms

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus,
prof.dr.ir. C.J. van Duijn, voor een commissie
aangewezen door het College voor Promoties in
het openbaar te verdedigen op
woensdag 23 april 2008 om 16.00 uur

door

Vincent Nollet

geboren te Veurne, België

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. H. Corporaal

Copromotor:
dr.ir. D.T.M.L. Verkest

This work was carried out at IMEC.

Cover design: Carmen Six and Vincent Nollet, cliparts from www.openclipart.org.

Printed by: Procopia N.V.

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Nollet, Vincent

Run-time management for future MPSoC platforms / by Vincent Nollet. -
Eindhoven : Technische Universiteit Eindhoven, 2008.
Proefschrift. - ISBN 978-90-386-1804-3
NUR 959
Trefw.: operating systems / multiprocessoren / computerprestaties /
ingebedde systemen.
Subject headings: operating systems, computers / multiprocessing systems /
resource allocation / embedded systems.

Acknowledgments

The destination has been reached. But a wise man once said:*"Live for the journey, not the destination"*. So, when looking back, I must say it has been a very *interesting* journey. Filled with a balance of joy and frustration, hope and despair, laughter and sadness. All these emotions have been shared with the people that I lived and worked with during this journey. I would like to express my gratitude to all of them. Particularly, I would like to thank:

Henk Corporaal, my promotor, for the many discussions, motivational talks and numerous hours spent over the years. His wife Carola for stimulating productivity on the saturday PhD meetings with envigorating coffee and cookies. Diederik Verkest, my co-promotor, for his practical guidance and for having an open door for all kinds of problems.

The PhD committee: Henk Corporaal, Diederik Verkest, Jef van Meerbergen, Francky Catthoor, Dirk Stroobandt and Johan Lukkien for investing time and energy into reading and commenting the text.

Theodore Marescaux and Prabhat Avasare for being my *brothers in arms* and *partners in crime*. Erik Brockmeyer for being an always-available, idea-sparring partner and for sharing his views on design-time application mapping. I learned a lot.

Wilfried Verachtert for providing me some *space* for writing this text. Kristof Denolf and Martin Palkovič for helping me navigate through the PhD administration jungle and Tom Ashby for giving a motivated answer to all my *English grammar* questions.

The *Gecko* team: Theodore Marescaux, Jean-Yves Mignolet, Prabhat Avasare, Paul Coene, Andrei Bartic, Will Moffat, Nam Pham Ngoc, Diederik Verkest and Serge Vernalde.

The MPSoC team: Theodore Marescaux, Erik Brockmeyer, Arnout Vandecappelle, Sven Wuytack, Eddy Degreef, Bart Durinck, Tanja Van Achteren, Rogier Baert, Martin Palkovič, Geert Vanmeerbeeck, Prabhat Avasare, Chantal Ykman-Couvreur, Stelios Mamagkakis, Tom Ashby, Zhe Ma and Maryse Wouters

The master thesis students who contributed to this work: Stefan Cosemans, Johan Leys, Jelle Welvaarts and Steven Wittens.

My family, more in particular my mother, my sister and my godmother who supported me although they often wondered *why* I was doing this. My parents-in-law, for boldy moving forward with the home renovations during the final PhD phase.

Finally, we get to the most valuable persons. My wife Carmen, who endured all the cursing, blood, sweat and tears, late evenings and early mornings, week-ends and *vacations* that were consumed in producing this thesis. I would not have reached this point without her loving support. I thank her with all my heart. While finalizing this thesis, my newborn son Jolan allowed me to put things in perspective: an unconditional smile - even when things don't go as good or as fast as *planned* - always brightens up the day.

–Vincent

Run-Time Management for Future MPSoC Platforms

Summary

In recent years, we are witnessing the dawning of the Multi-Processor System-on-Chip (MPSoC) era. In essence, this era is triggered by the need to handle more complex applications, while reducing overall cost of embedded (handheld) devices. This cost will mainly be determined by the cost of the hardware platform and the cost of designing applications for that platform.

The cost of a hardware platform will partly depend on its production volume. In turn, this means that flexible, (easily) programmable multi-purpose platforms will exhibit a lower cost. A multi-purpose platform not only requires flexibility, but should also combine a high performance with a low power consumption. To this end, MPSoC devices integrate computer architectural properties of various computing domains. Just like large-scale parallel and distributed systems, they contain multiple heterogeneous processing elements interconnected by a scalable, network-like structure. This helps in achieving scalable high performance. As in most mobile or portable embedded systems, there is a need for low-power operation and real-time behavior.

The cost of designing applications is equally important. Indeed, the actual value of future MPSoC devices is not contained within the embedded multiprocessor IC, but in their capability to provide the user of the device with an amount of services or experiences. So from an application viewpoint, MPSoCs are designed to efficiently process multimedia content in applications like video players, video conferencing, 3D gaming, augmented reality, etc. Such applications typically require a lot of processing power and a significant amount of memory. To keep up with ever evolving user needs and with new application standards appearing at a fast pace, MPSoC platforms need to be be easily programmable. Application scalability, i.e. the ability to use just enough platform resources according to the user requirements and with respect to the device capabilities is also an important factor.

Hence scalability, flexibility, real-time behavior, a high performance, a low power consumption and, finally, programmability are key components in realizing the success of MPSoC platforms.

The run-time manager is logically located between the application layer en the platform layer. It has a crucial role in realizing these MPSoC requirements. As it abstracts the platform hardware, it improves platform programmability. By deciding on resource assignment at run-time and based on the performance requirements of the user, the needs of the application and the capabilities of the platform, it contributes to flexibility, scalability and to low power operation. As it has an arbiter function between different applications, it enables real-time behavior.

This thesis details the key components of such an MPSoC run-time manager and provides a proof-of-concept implementation. These key components include application quality management algorithms linked to MPSoC resource management mechanisms and policies, adapted to the provided MPSoC platform services.

First, we describe the role, the responsibilities and the boundary conditions of an MPSoC run-time manager in a generic way. This includes a definition of the multiprocessor run-time management design space, a description of the run-time manager design trade-offs and a brief discussion on how these trade-offs affect the key MPSoC requirements. This design space definition and the trade-offs are illustrated based on ongoing research and on existing commercial and academic multiprocessor run-time management solutions.

Consequently, we introduce a fast and efficient resource allocation heuristic that considers FPGA fabric properties such as fragmentation. In addition, this thesis introduces a novel task assignment algorithm for handling soft IP cores denoted as *hierarchical configuration*. Hierarchical configuration managed by the run-time manager enables easier application design and increases the run-time spatial mapping freedom. In turn, this improves the performance of the resource assignment algorithm.

Furthermore, we introduce run-time task migration components. We detail a new run-time task migration policy closely coupled to the run-time resource assignment algorithm. In addition to detailing a design-environment supported mechanism that enables moving tasks between an ISP and fine-grained reconfigurable hardware, we also propose two novel task migration mechanisms tailored to the Network-on-Chip environment. Finally, we propose a novel mechanism for task migration initiation, based on reusing debug registers in modern embedded microprocessors.

We propose a reactive on-chip communication management mechanism. We show that by exploiting an injection rate control mechanism it is possible to provide a communication management system capable of providing a soft (reactive) QoS in a NoC.

We introduce a novel, platform independent run-time algorithm to perform quality management, i.e. to select an application quality operating point at run-time based on the user requirements and the available platform resources, as reported by the resource manager. This contribution also proposes a novel way to manage the interaction between the quality manager and the resource manager.

In order to have a the realistic, reproducible and flexible run-time manager testbench with respect to applications with multiple quality levels and implementation trade-offs, we have created an input data generation tool denoted *Pareto Surfaces For Free* (PSFF). The the PSFF tool is, to the best of our knowledge, the first tool that generates multiple realistic application operating points either based on profiling information of a real-life application or based on a designer-controlled random generator.

Finally, we provide a proof-of-concept demonstrator that combines these concepts and shows how these mechanisms and policies can operate for real-life situations. In addition, we show that the proposed solutions can be integrated into existing platform operating systems.

Contents

CHAPTER 1

Key MPSoC Requirements

In recent years, we are witnessing the dawning of the *Multi-Processor System-on-Chip* (MPSoC) era. In essence, this era is triggered by the need to handle more complex applications, while reducing the overall cost of embedded devices.

From an architecture viewpoint, these MPSoC platforms integrate computer architectural properties of various computing domains. Just like large-scale parallel and distributed systems, they contain multiple heterogeneous processing elements interconnected by a scalable, network-like structure to provide high compute performance. As in most mobile or portable embedded systems, there is a need for low-power operation and predictable behavior. To accommodate multiple applications and their varying needs, platform resources need to be allocated in a flexible way. In order to decrease time-to-market and to support software reuse, these MPSoC platforms should be easy programmable and provide hardware components that enable scalable software solutions.

From an application viewpoint, MPSoC platforms are designed to efficiently process e.g. multimedia content in applications like video players, video conferencing, 3D gaming, etc. Such applications typically require a lot of processing power and a significant amount of memory. In order to support such complex applications, MPSoC platforms need the to be easily programmable. In addition, they have to be flexible in order to keep up with ever evolving application standards. Application scalability, i.e. the ability to use just enough platform resources according to the user requirements and with respect to the platform capabilities, is equally important.

Hence, the *key requirements* with respect to both the platform hardware and application software of the MPSoC platform are: a high performance, a low power consumption, easy programmability, predictability and real-time behavior, flexibility, scalability, and reliability.

Executing a set of such challenging applications on an MPSoC platform requires matching the needs of every application with the offered MPSoC platform services. Due to changing run-time conditions with respect to e.g. user application requirements or having multiple simultaneously executing applications competing for platform resources, there is a need for a run-time decision-making and arbitration entity: *the run-time manager*.

The run-time manager is logically located between the application layer en the platform layer. As we will show, it has a crucial role in enabling or fulfilling the key MPSoC requirements. As it abstracts the platform hardware, it enables easy programmability. By deciding on resource assignment at run-time and based on the performance requirements of the user, the needs of the application and the capabilities of the platform, it contributes to flexibility, scalability and low power operation. Finally, as an arbiter, it enables predictable behavior in the presence of multiple simultaneously executing applications.

In this thesis, we detail the overall structure of the run-time manager and its role within the MPSoC context. We describe, in more detail, its components with their respective algorithms integrated into a global MPSoC application design and mapping flow. We show how the run-time manager interacts with the design-time application creation phase, how the application is managed at run-time with respect to the user requirements and the platform resources and how the run-time manager interacts with the distributed components of the platform. In addition, we provide an in-depth look at a proof-of-concept implementation of an MPSoC platform and its associated run-time manager.

The rest of this chapter is organized as follows. Section 1.1 motivates the origin of the MPSoC revolution and details the key requirements. Section 1.2 details the most prominent MPSoC platform components and their respective properties. This includes the heterogeneous processing elements, the flexible on-chip interconnect and the memory hierarchy. This section also investigates the contribution of every platform component with respect to the key requirements. Obviously, the way an MPSoC platform is composed will have an influence on the run-time manager requirements and implementation. Section 1.3 details contemporary and future applications targeted at these MPSoC platforms. These applications also have certain requirements with respect to the offered run-time management services. Section 1.4 first describes the overall role and structure of the run-time manager and details its potential contribution for tackling the key MPSoC requirements. This section also provides the thesis problem statement. Finally, Section 1.5 details the main contributions and presents the thesis outline.

1.1 Dawning of the Multi-Processor System-on-Chip Era

When taking a closer look at our everyday electronic devices, it is clear that we entered the MPSoC era. This section first explains why we entered this MPSoC era. Then, it details and motivates the key requirements of the MPSoC platform.

1.1.1 Motivating the MPSoC Revolution

Three main factors triggered the dawning of the System-on-Chip era: the enabling factor, the cost factor and the customer-pull factor.

The first factor is the *enabling factor*. As the silicon processing technology has entered the *deep sub-micron* (DSM) domain, it *enables* creating integrated circuits (IC) with billions of transistors.

The *cost* factor has multiple components. The first cost component entails the growing divergence between the silicon processing capability (i.e. the ability to put billions of transistors on a chip) and the design capability (i.e. the number of transistors a designer can design within a limited timeframe). This phenomena is known as the *design productivity gap* [3, 4] (Figure 1.1). This trend indicates that the amount of designer-months for a specific project is increasing rapidly.

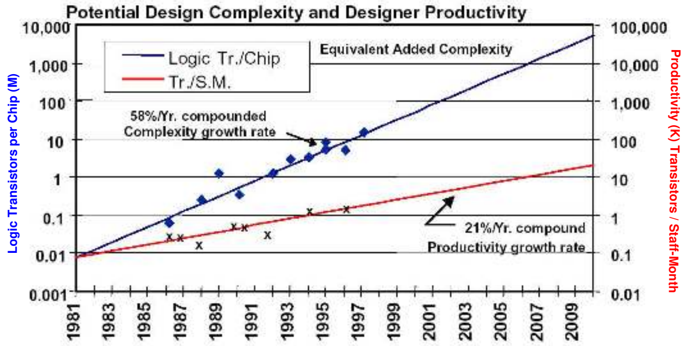


Figure 1.1: The so-called design productivity gap *indicates a growing divergence between the silicon processing capability and the design capability [3].*

A second cost component is the investment associated with creating such large ICs. On the one hand this investment entails the exponentially rising cost of the IC fabrication facilities [173]. For example, in 2005 Intel decided to build a new 300mm *Fab* in Arizona at an estimated cost of about $3 billion [149].

On the other hand, this investment entails rapidly growing *non-recurring engineering* (NRE) costs and lithography costs (i.e. the mask sets). For example, the cost of a 90nm node mask set is about $1 million, while the cost of a 45nm node mask set is well over $2 million [202]. Obviously, such investments represent a major stumbling block for creating low-volume ICs.

A solution for this cost problem is the so-called *platform-based design* paradigm [4], i.e. an extension to core-based design that uses groups of cores that already form a complete functional hardware component. These components contains programmable cores and reconfigurable logic. This way, derivative designs can easily be created. This flexibility also allows to, on the one hand, reuse the same platform for a different set of applications (i.e. increase the IC production volume) and, on the other hand, cope with ever evolving standards. This way, both the NRE and the design cost per IC will drop substantially.

The third factor is the *customer-pull* factor. MPSoC platforms serve the ever-increasing user or customer desire for more complex and integrated applications. Currently, MPSoC platforms are found in embedded devices ranging from consumer electronics to industrial equipment. This includes cell phones and PDAs, telecommunications equipment, digital televisions and set-top boxes, home video gaming equipment, etc. Pricing for these consumer devices is very competitive. A typical DVD player or a cell phone starts selling for as low as 50 Euro. This again confirms cost is an important factor in designing and manufacturing an MPSoC platform. Furthermore, these embedded devices are taking an ever more prominent role in our everyday life. The users expect them to be *always functioning* (i.e. behaving predictably) and *always-on* (i.e. low power operation).

An MPSoC platform can thus be defined as an application-focused, integrated circuit that implements most or all of the functions of a complete electronic system [103]. What components exactly are embedded in such an MPSoC heavily depends on the purpose of the MPSoC platform. In general, it contains I/O circuitry (analog or mixed signal), memory components, multiple *instruction set processors* (ISP), specialized logic and accelerators, buses, etc.

1.1.2 Definition and Motivation of the MPSoC Requirements

In this thesis, the MPSoC requirements are considered in the context of embedded platforms. The general purpose multiprocessor platform, also denoted as a *Chip Multi-Processor* (CMP), is mainly concerned with combining multiple processors on a single chip to increase overall compute performance. In contrast, the embedded platform also considers the power budget, cost constraints and application timing[1] constraints [102].

The key requirements for an embedded MPSoC platforms (with respect to both its hardware *and* its software) are *flexibility*, *scalability*, *predictability* and *real-time behavior*. In addition, MPSoC platforms should have a *low power consumption*, while providing a large amount of processing power i.e. *performance*. Finally, these MPSoC solutions should be produced at a *low cost*. As designing complex software applications in an efficient way takes an increasing amount of time, the MPSoC platforms should be *easily programmable*.

Flexibility and Programmability
Flexibility can be defined as the ability to use the same hardware resources for different purposes. Flexibility is needed to handle fast moving application standards, support new applications and amortize the high platform NRE cost. The program-

[1] The view on timing between an embedded platform and a general purpose platform is essentially different. In case of an embedded platform, timing is a constraint, while for a general purpose platform, it is a component of the cost function which needs to be minimized.

ming effort can be defined as the design effort required to get a wide variety of applications executing within specifications on the MPSoC platform. Indeed, custom design for a single product or application is no longer feasible. In this context, *Software Defined Radio* (SDR) shows the need for flexibility, as the same wireless digital baseband engine should be capable of seamlessly switching between multiple wireless communication services [224]. Combining platform flexibility with efficient application design and mapping also decreases the product time-to-market.

Scalability
Platform scalability is important, both from a design-time point of view as from a run-time point of view. Design-time scalability implies the possibility of adding more components to the platform without the need for a global platform redesign. Run-time scalability can be defined as the ability to run the same application without redesign on both a high-end, resource-rich platform and a low-end platform with minimal resources.

Predictability and Real-time behavior
Predictability refers to the degree that one can make a correct forecast of the application behavior. Real-time behavior refers to the degree at which the expected application or system output is produced within the predicted time-frame. Predictability and real-time behavior are important to ensure the *always functioning* user requirement of an embedded system.

Performance
Multiprocessor compute performance is important because contemporary multimedia and wireless applications require more processing power than a single flexible, programmable processing element can deliver. Furthermore, the user often wants to run multiple applications at the same time or some applications like e.g. video conferencing will require multiple platform services simultaneously. Pollack's Rule [182] states that for a single processor, increase in performance is roughly proportional to the square root of increase in complexity (i.e. additional transistors). A multiprocessor architecture, however, has the potential to provide near-linear performance improvement [27].

Power consumption
Power consumption is definitively an issue for battery-operated devices. A recent survey [232] conducted by market research group TNS revealed that *two days of battery-life during active use* was on top of the user's wish list for future mobile devices. The study also indicates a large consumer appetite for new multimedia and entertainment applications. However, users refrain from using these kinds of applications because of accelerated battery depletion. The continuously growing divergence between battery-capacity and the required power makes energy conservation of utmost importance. Even in non-battery operated devices we still care about power consumption. Limiting the power consumption is good for the lifetime of the chip, potentially avoids the need for a more expensive package and, reduces the power leakage, which is becoming quite dominant for deep sub-micron-and-beyond technology nodes. With respect to single processor platforms, multiprocessor platforms enable a reduced power consumption. Indeed, one can reduce the power consumption by executing an application on multiple processing elements operating at a lower voltage and frequency.

Reliability
Reliability can be defined as *the ability of a system or component to perform its required functions under stated conditions for a specified period of time* [80]. In other words, a reliable hardware or software component consistently performs according to its specifications and, in theory, is totally free of technical errors. In practice, a reliable system has often been designed to cope with technical errors. In the deep sub-micron-and-beyond technology nodes, reliability issues will play an increasingly important role [61] (see Section 8.2.5).

Obviously, these requirements will have an effect on the components and the composition of the MPSoC platform. In turn, this will have an effect on how such future embedded multiprocessor platforms are managed at run-time.

1.2 The MPSoC Platform

The main components of the *System-on-Chip* (SoC) are: the processing elements, responsible for executing the applications, the on-chip memories, responsible for storing both the application data and instructions, I/O components, responsible for communicating with the outside world and, finally, the on-chip interconnect structure, responsible for linking the processing elements with the memories and the I/O components.

Figure 1.2 details the MPSoC platform template used throughout this thesis. This template lives up to the prediction that future MPSoC platforms will consist of a mixture of tiles, containing heterogeneous computing resources and memory resources, interconnected by a flexible and configurable on-chip communications fabric [27, 123, 130]. Next to a configuration and communication engine, every processing tile also contains some local (private) memory. Furthermore, the SoC also contains some shared memory tiles. This view roughly corresponds to the SoC template presented in the 2005 ITRS roadmap [5].

The rest of this section details the rationale behind each of these SoC components and shows how they contribute to the key MPSoC requirements, i.e. increasing the compute power of the MPSoC platform while controlling both power consumption and SoC cost and while ensuring scalability, flexibility, predictability and real-time behavior.

1.2.1 Heterogeneous Processing Elements

The need for more platform performance can be fulfilled in various ways. Some companies, like ARM and MIPS are upgrading their *General Purpose Processor* (GPP) cores with signal processing capabilities like e.g. the ARM1136 and the MIPS24KEc core. Their rationale is to provide a single general purpose processing element with enough power to replace the *Digital Signal Processor* (DSP), while still retaining the original GPP functionality. However, Pollack's Rule [27,182] states that it is far more efficient to put multiple cores on a chip than to boost the performance of the single core by adding more complexity.

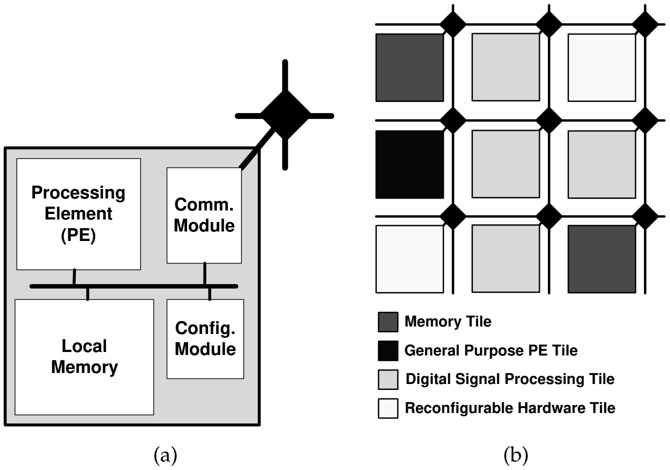


Figure 1.2: *MPSoC template: a tile of the MPSoC system can contain a PE with communication infrastructure and local memory (a). The MPSoC will be composed of a set of tiles (PE, memory or I/O) (b).*

A far better solution is to put multiple general purpose cores together on a single chip. This way, the multicore benefits can be harvested easily when multiple (legacy) applications run simultaneously. The ARM MPCore, for example, is a homogeneous architecture that contains up to four ARM11 cores. Such homogeneous MPSoC platforms are well suited to dominate the more general purpose processing domain and to handle legacy applications.

Due to the energy-flexibility and the flexibility-performance trade-off, domain-specific MPSoCs are likely to contain several specialized, programmable *processing elements* (PE), like a configurable accelerator, a DSP, fine-grain or coarse-grain reconfigurable hardware, etc. next to one or more general purpose PEs. This way it is possible to increase the compute performance while, at the same time, reduce the power consumption and remain flexible. This implies that domain-specific MPSoCs will, to a certain extent, be heterogeneous when it comes to processing elements.

Although the number of processing elements contained in such an MPSoC platform obviously depends on its purpose, there is a clear trend for an ever increasing amount of cores. Indeed, the ITRS roadmap predicts a five-fold increase of the amount of processing elements by 2010 with respect to 2005 [5].

A quick look at some contemporary MPSoC platforms shows that most of them contain application-specific PEs next to more general purpose PEs.

STI CELL

The CELL Processor [128], jointly developed by Sony, Toshiba and IBM (STI), contains up to nine PEs. One general purpose *Power Processor Element* (PPE) and eight so-called *Synergistic Processor Elements* (SPE). The PPE runs the operating system and the control tasks, while the SPEs provides the required compute performance. This MPSoC is at the heart of the *PlayStation 3* game console.

STMicroelectronics Nomadik
The STMicroelectronics Nomadik [15] contains an ARM926 RISC CPU with Java acceleration and DSP instruction extensions. In addition, the Nomadik features several *Very Long Instruction Word* (VLIW) DSP cores that act as accelerators for handling graphics and for image, video and audio processing. This MPSoC uses high end cell phones as its key driver.

Texas Instruments OMAP
The TI OMAP2 MPSoC contains an ARM11 RISC CPU, a TI C55x digital signal processor, a 2D/3D graphics accelerator and a video accelerator. The OMAP MPSoC technology can be found in e.g. high end Nokia cell phones.

Philips Nexperia
The Philips Nexperia PNX8550 (also known as the Viper2) is a media SoC mainly targeted at digital television set-top boxes. It contains three processors: one MIPS PR4450 general purpose RISC processor for control processing and balancing off-chip functions and two TriMedia TM3260 DSP processors for processing the real-time multimedia content.

Configurable Cores (Tensilica, ARC, etc.)
Another trend in dealing with the energy/flexibility and the flexibility/performance trade-off is *configurable cores*. Tensilica, for example, advocates the *sea of processors* idea, where each processor is tuned towards a specific function or application. Tensilica provides tools to generate both the optimized processor core, further denoted as *Application Specific Instruction set Processor* (ASIP), and the corresponding software programming toolset. The ASIP motivation is that tuned PEs are more efficient than general-purpose PEs, while providing more flexibility than ASICs [188].

Fine-grain Reconfigurable Hardware (Xilinx, Altera, etc.)
As Section 1.1 explains, cost, flexibility and scalability are key requirements of an MPSoC platform. That is why the *Field Programmable Gate Array* (FPGA), also denoted as fine-grain reconfigurable hardware, is becoming a viable alternative for the ASIC in many markets. As FPGAs are an off-the-shelf product, there is no NRE cost. In addition, FPGA fabric also starts to get integrated into SoCs. This way, it is possible to create high performance custom logic after manufacturing. Performance has long been the primary figure of merit with respect to FPGA technology [55]. Power consumption was never an issue, up till recently. Today, designers of next generation FPGA fabric pay greater attention to power consumption as some potential application domains have a constrained power budget. Furthermore, there is still no consensus on how this FPGA fabric should be used as a regular processing element within the MPSoC platform [103]. Additionally, most FPGA fabric vendors also provide soft IP blocks that provide GPP and DSP functionality. This enables creating custom MPSoC platforms that combine easy programmability with flexibility and scalability.

Intuitively, it is quite clear that supporting this heterogeneity with respect to processing elements will make both application development and run-time management harder with respect to the homogeneous case.

1.2.2 Flexible on-Chip Interconnect

Issues like power consumption, flexibility and scalability of the communication infrastructure and predictable platform behavior are also tackled by novel MPSoC on-chip communication architectures. Instead of interconnecting the different PEs (or IP blocks in general) through a wide variety of buses, future SoCs will use a more flexible interconnect like e.g. a *Network-on-Chip* (NoC) [22, 57].

In contrast to a bus-based interconnect, a NoC does not need to drive long (i.e. cross-chip) wires, only wire segments. Scalability is inherently present in the NoC concept [132]. In a bus-based platform, the arbitration bottleneck grows with the number of bus masters and the bus bandwidth is shared by all attached IP blocks. In a NoC-based platform, the routing and arbitration is distributed and the aggregate bandwidth scales with the network As most advanced on-chip communication fabrics, NoCs have a way to allocate a certain amount of resources for a certain application. This results in minimal inter-application interference with respect to the shared resource and, hence, predictable behavior. By interconnecting IP blocks in a flexible way and by allowing to alter both communication paths and the reserved communication bandwidth at run-time, NoCs contribute to using the MPSoC computational resources in a flexible way.

Although, the NoC communication architecture bears a close resemblance with traditional parallel and distributed systems, there are some important key differences. In contrast to off-chip networks, the on-chip networks have limited resources (e.g. memory resources) and therefore require adapted communication protocols. In addition, NoCs can have a higher bandwidth and the NoC traffic is likely to be more deterministic and periodic than in off-chip networks [119].

The term *Network-on-Chip* is used for a variety of state-of-the-art on-chip interconnect solutions [194]: ranging from multi-layer segmented buses and (partial) crossbars to on-chip networks as envisioned by Dally et al. [57]. The latter view is used throughout this thesis. Well known examples of research NoCs are Philips Æthereal [84], SPIN [7] and MANGO [25]. Bjerregaard et al. [25] provide an excellent survey of existing NoC research and NoC best practices. Salminen et al. compares a wide range of NoC solutions (including the Gecko NoC, discussed in Chapter 6). In recent years, several start-up companies like Arteris, with the *Arteris NoC*, and Sonics [235], with the *SonicsMX SMART Interconnect* have started to commercialize these NoC concepts.

This new communication architecture also introduces new run-time management challenges. For example, using a NoC will make it possible to reroute communication, i.e. to change at run-time the communication path between source and destination. In addition, resource management algorithms have to take the properties of the interconnect into account. In contrast to a bus architecture, for example, the communication latency between PEs can be different.

1.2.3 MPSoC Memory Hierarchy

A key area to handle requirements like scalability, high performance and low power consumption is the memory subsystem.

Memory access latency can cause the compute process to halt (i.e. dropping performance) and accessing memories represents an important component in the overall system energy cost. Typically, smaller memories require less time and less energy to access and require a smaller chip area. Stravers et al. [213] deduced that, in the future, an ever increasing percentage of silicon die area will be dedicated to memory functionality. Limiting the amount of required memory will reduce chip cost as die area represents a clear cost factor. This reasoning is also adopted by the ITRS roadmap [5]. Hence, for memory-intensive applications such as the upcoming multimedia applications (see Section 1.3) reducing memory needs is of utmost importance.

One way to deal with the memory latency and power consumption issue is to add a memory hierarchy: instead of using a single on-chip memory block to store all data, the hierarchy enables to store data that is often used in smaller memories closer to the processor. A memory hierarchy also relieves the contention on a central shared memory, hence improving the scalability. Unfortunately, adding a memory hierarchy also has its downsides: it increases die area and, in case of a cache-based hierarchy, it amplifies processing time variations. These processing time variations have a devastating effect on the predictability.

As a solution, scratchpad memories have been introduced in the MPSoC platform environment [181, 228]. Compared to caches, scratchpad memories require less energy per memory access, they require less on-chip area, and exhibit a predictable behavior. However, the use of scratchpad memories requires extensive design-time application analysis and exploration of the application in combination with suitable run-time management support.

In the CELL processor architecture, for example, the Power PE has 32kB instruction and data level-1 cache combined with 512kB level-2 cache. The Synergistic PEs each contain 256 kB scratchpad memory.

1.3 The Rise of Multimedia Services

The actual value of future devices containing MPSoC platforms does not lie within the embedded multiprocessor IC itself, but in its capability to provide the user of the device with an amount of services or experiences. Indeed, the industry has far larger opportunities (and incentives) to differentiate on the services it provides to its customers than on platform differentiation. This means that the applications will drive the need for and the evolution of the MPSoC platforms. So what multimedia applications and services will be created on top of these MPSoC platforms?

Current smartphones and feature phones combine mobile phone functions with PDA functionality. These devices have the ability to take low-resolution pictures, play low-quality low-resolution video and provide 2D and 3D games. However, resolution as well as framerate and video quality continuously improve.

Video decoding and encoding will undoubtedly be a major future driver application for mobile devices. These video applications will require considerable compute power. Stating that, due to device size limitations, the mobile display does not allow high-resolution and, hence, the amount of required compute power will remain limited does not hold. As mobile storage capacity keeps rising at a high rate, these embedded devices will act as a container for pictures and home video collections. These devices will also act as a *decoding* device for high resolution external displays (e.g. TV display). Furthermore, rising storage capacity and increasing phone-camera resolutions will enable using the cell phone as a video recorder. This implies that the mobile terminal must also be capable of *encoding* video, which is typically more compute intensive than video decoding.

Mobile TV, like the DVB-H standard, is also considered to be a driver application [177]. On the one hand, network providers see it as a revenue-generating service at a time where more traditional income flows (e.g. voice traffic) are stagnating. Integrated Device Manufacturers see it as an opportunity to create new upmarket handheld devices.

Wireless enabled terminals are also looking for flexible, low-power MPSoC solutions in order to support an increasingly large variety of wireless communication modes (e.g. Cellular, WiFi, WiMax, WPANs) [224]. Creating a dedicated ASIC for every mode is no longer feasible due to fast changing requirements and a decreasing time-to-market. Hence, the motivation to create a *Software Defined Radio* (SDR).

In a more distant future, handheld devices will be providing *augmented reality* services. Augmented reality enhances the user's perception of the surroundings by combining the physical world with the virtual world. Thus paving the way for media-rich, location-aware and context-aware applications [88] like e.g. attaching audio and video objects to physical objects. In addition, there is the possibility of augmented reality gaming, where 3D virtual opponents are placed in a life real-world video stream [220].

Set-top boxes, building on MPSoC platforms, will provide services like digital television, Internet access, in-home entertainment (e.g. act as a game console) and home automation.

But even beyond multimedia, in the automotive and robotics environment, there is need for the key properties that MPSoC platforms provide. Consider for example the compute power required for creating an *autonomous driverless car* or for a *personal robot servant*. Even the very limited prototypes of today already rely on an array of general purpose processors and DSPs [85].

The run-time manager has an important role in enabling these multimedia applications. In short, the run-time manager has to ensure that these driver applications receive the platform resources and services they require. In addition, the run-time manager has to provide the appropriate platform abstractions to make application design easier and faster. Section 1.4.1 provides more details.

1.4 MPSoC Run-Time Management

From an abstraction viewpoint, the run-time manager is located between the MPSoC platform and the application(s) (Figure 1.3). It is composed of a system manager and a run-time library. The system manager, containing the quality manager and the resource manager, is responsible for decision making, while the *Run-Time Library*

(RTLib) provides platform abstraction services. The run-time library collaborates with the system manager for executing the decisions, for run-time interaction with the applications and for monitoring both platform resources. Chapter 2 provides an in-depth description of these components, their interaction and their implementation options.

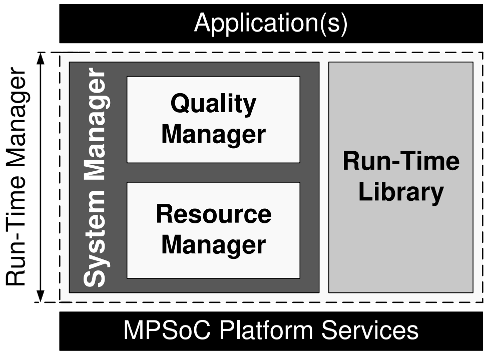


Figure 1.3: *The run-time manager contains system management components and a run-time library. In turn, the system manager contains a quality manager and a resource manager.*

This section first describes the role of the run-time manager and its components with respect to the key MPSoC requirements. Then, it details the general run-time management problem statement, followed by a short description of the specific solutions presented in this thesis.

1.4.1 Role in Addressing the MPSoC Requirements

The run-time manager plays a crucial role in enabling or fulfilling the key MPSoC requirements and in linking the needs of the application in an efficient way to the provided platform services. This section briefly describes how the different run-time management components (Figure 1.3) contribute to addressing these requirements.

- **Cost reduction**. As the run-time manager abstracts the MPSoC platform hardware details with respect to the application designer, it has an important part in providing an easily programmable platform. Meaning that the provided platform abstractions should allow the designer to create applications in a fast and efficient way (i.e. *RTLib*).

- **Scalability**. Applications can have multiple application quality levels. These quality levels provide a certain user experience given a certain amount of platform resources. By selecting the right application quality, depending on the available platform resources and the run-time user requirements, the run-time manager enables both platform and application scalability. This means that the run-time manager should provide appropriate algorithms for quality management (i.e. *Quality Manager*). Furthermore, one has to take into account that platform resource availability as well as the user requirements can change over time. Hence, the run-time manager should also provide a way to seamlessly switch between quality levels.

- **Flexibility**. Depending on the user requirements and the available platform resources, the run-time manager (i.e. *Resource Manager*) has to allocate the appropriate platform resources in a flexible way. These resource allocation decisions have to be made at run-time, so one requires a fast and efficient decision making algorithm. As the MPSoC platform will contain multiple heterogeneous processing elements, this allocation algorithm must be able to handle tasks that have support for multiple processing element types and it has to take the specific properties of the on-chip processing and communication resources into account. Also here, one has to take into account that run-time conditions change. This means that the run-time manager needs to be able to seamlessly reallocate resources at run-time, even when e.g. the source and target processing element of a certain task have a different architecture.

- **Predictability and Real-time behavior**. To achieve predictable and real-time application performance, this thesis focuses on how to use design-time application mapping and characterization information for run-time decision-making. This means that the run-time manager should provide an abstraction layer (i.e. *RTLib*) on top of the platform hardware services that allows the designer or the design tool to reason about application timing and mapping performance. At run-time, the application must also be able to depend on the availability (collaboration between *RTLib* and *Resource Manager*) and the predictable behavior of the offered platform services.

- **High compute performance and low power operation**. The cost functions incorporated into the decision making algorithms of the *Resource Manager* have to consider power and performance. This means, for example, that when assigning platform resources to the application the run-time manager can optimize for a low power solution that still satisfies the performance constraints. Furthermore, the run-time manager has to consider the trade-off between flexibility and performance. This especially holds in case of e.g. reconfigurable hardware processing elements.

- **Reliability**. The run-time manager also plays a central role in providing a reliable computing platform for the application designer. This means managing platform degradation effects caused by e.g. *chip hot-spots*. This thesis does not directly address the reliability platform requirement. In the future, however, some concepts (like e.g. task migration, discussed in Chapter 4) could be extended to handle such degradation effects.

1.4.2 Application Composability

Besides playing a role in ensuring predictable and real-time application behavior, the run-time manager is a crucial component in ensuring application composability.

Consider an application designer responsible for getting an application to execute on an MPSoC platform given a set of real-time performance constraints. In case this application designer can assume full control of the MPSoC platform resources for the target application, the amount of uncertainty is fairly low. However, in case the platform user is free to execute multiple (at design-time unknown) applications, a single designer can no longer assume to have full control of the resources. Ideally, these applications should not interfere with each other, while the real-time behavior of the individual applications with respect to the user requirements still need to be guaranteed.

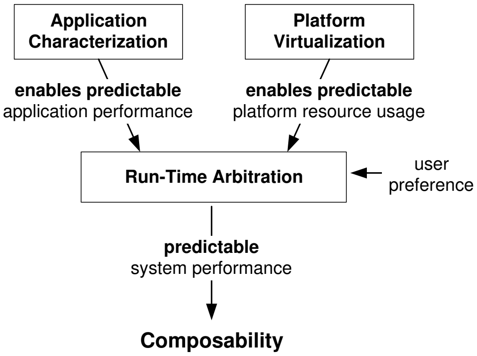

Application Characterization
Platform Virtualization
enables predictable application performance
enables predictable platform resource usage
Run-Time Arbitration
user preference
predictable system performance
Composability

Figure 1.4: System composability can be achieved through application characterization and platform resource virtualization combined with run-time arbitration given user preferences.

Hence, a *composability* issue arises when multiple applications are executing simultaneously. Composability deals with the degree to which a single application designer can create a single application in isolation, but can still reason about the application performance on the MPSoC platform when combined with other applications [118].

In this sense, the ability to provide system composability is built on *application characterization*, *platform virtualization* and the presence of a *run-time arbiter* (Figure 1.4).

Application characterization deals with estimating the platform resource requirements of an application in order to reach a certain performance goal. This characterization involves determining the required computing resources as well as the required memory resources and on-chip communication resources. Application characterization is needed for predictable application performance. This characterization information is passed to the run-time manager as part of the design-time application analysis information.

Platform virtualization involves providing the platform means to ensure that a certain application (or part of the application) is granted a guaranteed service. This can involve computing resource services but it also means access to on-chip communication services and memory services. The provided guarantees can either be hard (strictly enforced) or soft (i.e. enforced within certain boundaries). In addition, the services can be implemented in hardware or in a combination of hardware (platform services) and software (RTLib).

As multiple applications will content for resources, the run-time manager, acting as a resource arbiter, is responsible for reserving the required amount of platform resources for every application and for enforcing the resource usage bounds. This way, predictable and real-time system performance can be achieved which, in turn, leads to composability.

1.4.3 Problem Statement

In a broad sense, the main problem an MPSoC run-time manager needs to solve is: *matching of the needs and the properties of the application*[2] *in a fast, efficient and flexible way with the provided MPSoC platform services and properties in order to execute multiple applications while minimizing inter-application interference.*

Obviously, this problem domain is so vast that exploring the entire problem space easily exceeds the scope of this thesis. Hence, we create and demonstrate a quality manager, a resource manager and a RTLib and, consequently, apply them to some specific example problems. These run-time manager components have to consider the following requirements.

In order to match the needs of the application with the available platform services in a fast and efficient way, one requires a quality manager and a resource manager with their respective algorithms. The quality manager has to consider the application user requirements to derive the application resource requirements, while the resource manager takes these resource requirements and matches them with the available platform resources. These platform resources include the heterogeneous processing elements, the memory hierarchy and the flexible on-chip interconnect. It is obvious that, to find the optimal user value and to minimize inter-application interference on all levels, the quality manager and the resource manager will have to cooperate. Furthermore, both the quality manager and the resource manager should be capable of considering design-time application information, generated by the application design flow and its associated tools. This information will enable faster and more accurate (application-specific) decision making.

In order to retain the run-time flexibility with respect to the changing application requirements and the availability of platform resources, the run-time manager should be capable of changing both the application quality level as well as the application's resource allocation without causing (too much) inter-application interference.

The run-time management functionality should also provide a platform abstraction layer (i.e. RTLib) that allows a designer to create applications in an efficient way. At the same time, this abstraction layer should optimally exploit the offered platform services in order to produces as little run-time overhead as possible.

1.5 Main Contributions and Thesis Outline

This section details the contributions of this thesis with respect to the run-time management of embedded systems in general and heterogeneous multiprocessor SoCs in particular. Figure 1.5 illustrates our contributions.

The main contributions of our work can be summarized as follows.

- **Generic description of the MPSoC run-time management components and their design space (Chapter 2).** [163] Chapter 2 describes, in a generic way, the role, the responsibilities and the boundary conditions of an MPSoC run-time

[2] This includes the user needs with respect to the application.

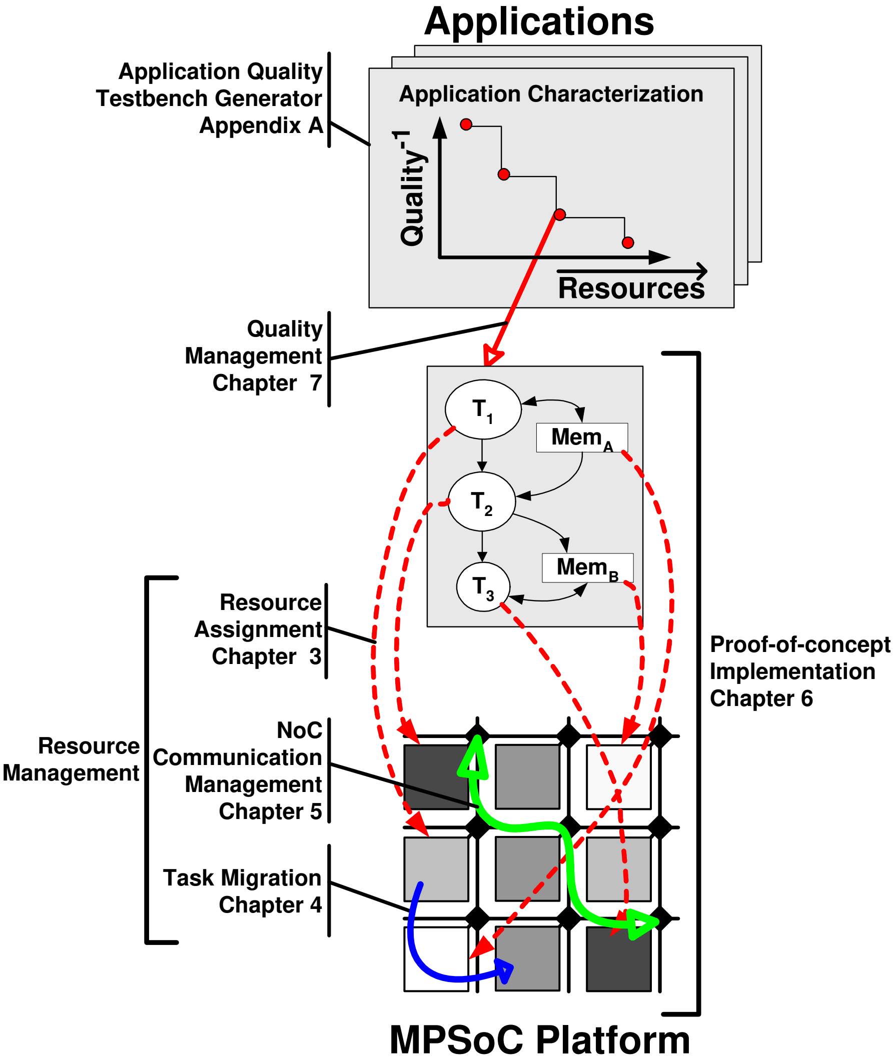


Figure 1.5: *Detailed view of the main thesis contributions. The resource management aspects are covered in Chapter 3, Chapter 4 and Chapter 5. Chapter 6 details a proof-of-concept implementation that mainly focuses on these resource management components. Chapter 7 deals with the run-time quality management and its link with both the application characterization information and the resource manager. Appendix A details a generator for testbench applications with multiple operating points.*

manager. To the best of our knowledge, we are the first to generically describe the MPSoC run-time management functionality and its implementation space. We show that the MPSoC run-time manager has to unite the embedded environment, the desktop environment and the parallel and distributed systems environment. We also provide an in-depth description of the responsibilities and the interaction of the MPSoC run-time manager components. We substantiate these run-time management functions and their implementation space by describing and positioning existing academic and commercial multiprocessor run-time management solutions.

- **Run-time resource manager algorithms (Chapter 3)** [131, 136, 157, 158, 160, 161]. Chapter 3 details a run-time resource assignment algorithm that takes the properties of fine-grain reconfigurable hardware tiles, present within the MPSoC, into account. Besides providing a fast and efficient generic heuristic for MPSoC resource assignment, we also propose a set of add-ons to improve the performance of this algorithm in the presence of fine-grain reconfigurable hardware tiles. In addition, we introduce a novel resource assignment algorithm that combines assignment of application tasks and soft IP cores. This technique is denoted as hierarchical configuration. Hierarchical configuration managed by the run-time manager enables easier application design and increases the run-time spatial mapping freedom. In turn, this results in a higher performance for the resource assignment algorithm.

- **MPSoC run-time task migration (Chapter 4)** [136,143,144,155,157]. Chapter 4 describes run-time task migration in a heterogeneous MPSoC environment. We detail a new run-time task migration policy closely coupled to the run-time resource assignment algorithm. In addition to detailing a design-environment supported mechanism, that enables moving tasks between an ISP and fine-grained reconfigurable hardware, we also propose two novel task migration mechanisms tailored to the Network-on-Chip environment. Finally, we propose a novel mechanism for task migration initiation, based on creative use of the debug registers in modern embedded microprocessors.

- **Reactive Network-on-Chip communication management (Chapter 5)** [17,162]. Chapter 5 details a communication management algorithm linked to a Network-on-Chip traffic shaping mechanism. We propose a reactive on-chip communication management mechanism with several MPSoC specific management policies. We show that by exploiting an injection rate control mechanism with global decision making, it is possible to provide a communication management system capable of providing a soft QoS in a best-effort NoC. Hence, we provide an MPSoC-specific communication management algorithm tailored to the injection rate control mechanism.

- **MPSoC proof-of-concept implementation (Chapter 6)** [136, 156]. Chapter 6 describes the *Gecko* demonstrators. This implementation-oriented contribution shows that our run-time management contributions, i.e. resource assignment and hierarchical configuration, run-time task migration and Network-on-Chip communication management can be implemented into an operating system handling the MPSoC platform resources. The MPSoC platform, denoted as *Gecko*, is emulated on a *Field Programmable Gate Array* (FPGA) linked to a

Compaq iPAQ PDA. The PDA StrongARM processor is linked to 8 FPGA slave processors by a 3-by-3 mesh Network-on-Chip. This chapter also details how the Gecko RTLib functionality is provided by an IMEC in-house 16-bit processor instantiated next to every *Gecko* processing element. The *Gecko* proof-of-concept applications consist of an edge detector, a small 3D *shoot'em up* game and a motion-JPEG video decoder.

- **Algorithms for run-time quality management (Chapter 7)** [49–52]. Chapter 7 introduces a novel, platform independent run-time algorithm to perform quality management, i.e. to select an application quality operating point at run-time based on the user requirements and the available platform resources, as reported by the resource manager. This contribution also proposes a way to manage the interaction between the quality manager and the resource manager.

- **Pareto Surfaces For Free (Appendix A).** In order to have a realistic, reproducible and flexible run-time manager testbench with respect to applications with multiple quality levels and implementation trade-offs, Appendix A details an input data generation tool denoted *Pareto Surfaces For Free* (PSFF). The PSFF tool is, to the best of our knowledge, the first tool that generates multiple realistic application operating points, either based on profiling information of a real-life application, or based on a designer-controlled random generator.

The last two appendices provide additional information with respect to the contributions. Appendix B provides the Task Graphs For Free (TGFF) configuration file used to generate task graphs and associated details for the experiments of Chapter 3. More specifically, it details the options for the resource assignment experiments (Section 3.6) and for generating the softcore library for the hierarchical configuration (Section 3.9) experiments. Appendix C details the IMEC MPSoC application mapping flow and its associated tool-chain. It shows how the run-time manager fits in this mapping flow and it details the different trade-offs a designer can make when mapping a sequential application onto an MPSoC platform.

This thesis also pays a lot of attention to related work The run-time manager components and their implementation space, detailed in Chapter 2, are substantiated with academic and industrial MPSoC run-time manager examples. In addition, every chapter provides an overview of the related work relevant to its contribution. Related work that motivates a statement or an assumption is provided when appropriate. Throughout this thesis, the reader will also find a number of example text boxes. These examples should help the reader in understanding the explained concepts. They do not contain additional information required to understand the remainder of the thesis.

1.6 Contributions Acknowledgments

We have to give credit where credit is due. The close link between run-time manager and the *Gecko* hardware was achieved by a close collaboration with Theodore Marescaux. As a result, the introduction of Chapter 5 (i.e. Sections 5.1 till 5.3) and the contribution of Chapter 6 are, shared between the author and the PhD thesis of Theo Marescaux [132]. Furthermore, the actual development of the *Gecko I* and $Gecko^2$ (read: Gecko square) demonstrators (Chapter 6) has many contributors among whom Theodore Marescaux, Jean-Yves Mignolet, Andrei Bartic, Will Moffat, Paul Coene, Prabhat Avasare, Nam Pham Ngoc and Diederik Verkest (apologies to those I may have forgotten). The contributions contained in Chapter 7 required a close collaboration between the design-time mapping flow, its associated design tools and the run-time manager. So, the base-line operating point selection algorithm (Section 7.2 of Chapter 7) was achieved in close collaboration with Chantal Ykman-Couvreur. The QSDPCM case study, to illustrate the application design trade-offs and application adaptivity (Section 7.1.2 of Chapter 7), was achieved due to the extensive discussions and enthusiastic collaboration of Chantal Ykman-Couvreur and Erik Brockmeyer. The contribution related to combining design-time application analysis with run-time decision-making (Chapter 7) was seeded by the work of the IMEC TCM PhD team (among whom Peng Yang and Zhe Ma).

CHAPTER 2

A Safari Through the MPSoC Run-Time Management Jungle

The operating system is the most important software that runs on any computing device. It has a far greater impact on how a user or programmer perceives the system than the actual hardware does [217]. In general, the operating system is responsible for a broad range of system managerial tasks such as processor management, managing computing platform peripherals and I/O devices. This includes data and file system management and maintaining an interface with the user. The operating system also provides a set of entry points for starting applications.

At the heart of the operating system is the *run-time manager*. The run-time manager is responsible for handling the critical (typically shared) resources of any computing platform: one or more potentially heterogeneous processing elements, the (on-chip) volatile memory and the platform communication infrastructure including I/O.

The run-time manager is composed of multiple interacting components. The *quality manager* is responsible for application interaction with respect to the quality needs of the user. The *resource manager* is responsible for efficiently managing the platform resources with respect to the requirements of the different executing applications. This includes minimizing the amount of inter-application interference. Furthermore, the run-time manager provides a *run-time library component* that provides the necessary platform abstraction functions used by the designer for creating the application and called by the application at run-time.

There are multiple ways of implementing the MPSoC run-time management functionality depending on the needs of the application and the services provided by the platform. We introduce an MPSoC run-time management implementation design space that contains three axis. The first axis describes the amount of distribution. The second axis details the range from pure software implementation to a pure hardware implementation. The third axis is concerned with the amount of specialization of the run-time manager with respect to the application or the offered platform services.

Besides detailing the run-time manager functionality and its implementation design space, this chapter also serves as a reference for the rest of the thesis as it puts the introduced run-time manager functionality and the implementation decisions in the correct perspective. In short, the chapter contains the following sections. Section 2.1 details the boundary conditions in which the run-time manager has to operate. Consequently, Section 2.2 details the different run-time management components and their respective roles. This section also describes the information flow that drives the run-time manager and the role of the run-time manager in guaranteeing deterministic application behavior and in minimizing inter-application interference. Section 2.3 details the run-time management design space and illustrates the implementation options with examples. In addition, this section motivates the design decisions for the run-time manager detailed throughout this thesis. Finally, Section 2.4 briefly describes some MPSoC run-time management examples and places them into the design space.

2.1 MPSoC Run-time Management Constraints

The run-time manager is squeezed between the platform hardware layer and the application layer. This means that the run-time manager has to provide the services required by the application by building on top of the available platform services.

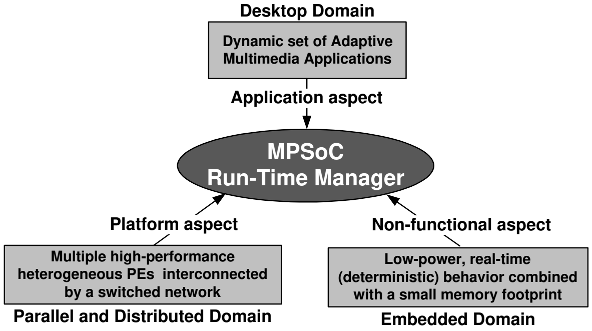

Desktop Domain
Dynamic set of Adaptive Multimedia Applications
Application aspect
MPSoC Run-Time Manager
Platform aspect
Non-functional aspect
Multiple high-performance heterogeneous PEs interconnected by a switched network
Parallel and Distributed Domain
Low-power, real-time (deterministic) behavior combined with a small memory footprint
Embedded Domain

Figure 2.1: *MPSoC run-time manager at the crossroad of* classic *computing domains.*

The run-time manager has to operate within the boundary constraints of the MPSoC environment. As Figure 2.1 illustrates, the MPSoC run-time manager sits at the crossroad of several *classic* computing domains when it comes to applications, platform hardware and non-functional constraints.

- **Platform hardware.** The MPSoC platform hardware (see Section 1.2) is in fact the on-chip equivalent of traditional parallel and distributed systems. These systems are composed of multiple, often heterogeneous, processing elements with a complex memory hierarchy interconnected by a switched network fabric.

- **Applications functionality** As Section 1.3 explains, the actual user value of the MPSoC device sits in its capability of handling a dynamic set of applications. This requirement is typically found in the desktop computing domain[1]. In this domain, it is important to maximize the overall throughput and to provide fairness between applications. However, run-time management support for soft real-time behavior, adaptivity towards the application and user interaction play an equally important role [185].

- **Non-functional application constraints.** The discussed MPSoC platforms will often end up in embedded and/or mobile devices. This means that the run-time manager will have to consider the constraints that deal with resource scarcity. This includes, for example, a limited amount of battery power. Additionally, there are constraints that deal with the (expected) behavior of the device such as e.g. real-time behavior and the users' *always-on* expectation.

Dealing with these constraints not only requires a decision on which run-time management functionality to provide with respect to the application design flow and the platform services (Section 2.2), it also requires a decision on how this functionality will be implemented (Section 2.3).

2.2 Run-Time Management Functionality

This section details the responsibilities of the run-time manager, i.e. *what* services should the run-time manager provide with respect to the application and what tasks should it take care of with respect to the platform. We first detail the system management components. Consequently, we describe the role of the run-time library.

2.2.1 System Management

As Figure 2.2 illustrates, three basic functions are contained within system management: the quality management block, a block containing the resource management policies, and a block with the resource management mechanisms.

Quality Management

As Section 2.3.3 explains more in-depth, dealing with the MPSoC run-time management requirements and constraints in an efficient way can be achieved (1) by en-

[1] Recently, multiprocessor platforms have been introduced in an accelerated way into the Desktop computing domain. This also improves the performance of the Desktop platforms with respect to the requirements of their application.

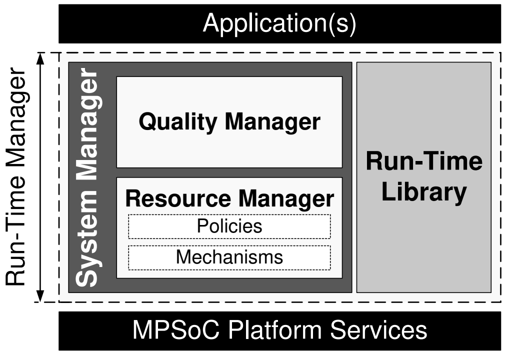


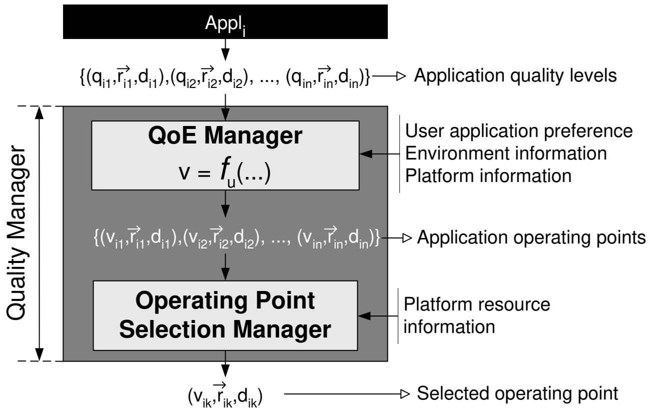
Figure 2.2: *On a high level, the run-time manager contains system management components and a run-time library.*

abling the run-time manager to exploit design-time and run-time application knowledge, and (2) by tuning the run-time manager in order to take into account the platform properties and the offered hardware services. Our run-time manager accepts application quality and implementation information that is generated at design-time. This means the run-time manager is aware about the capabilities and quality levels supported by the (adaptive) application and their respective properties.

The quality manager, sometimes referred to as *Quality of Service* (QoS) manager, is a platform independent component that interacts with the application, the user and the platform-specific resource manager. The goal of the quality manager is to find the sweet-spot between the capabilities of the application, i.e. what quality levels does the application support, the requirements of the user, i.e. what quality level provides the most value at a certain moment, and the available platform resources. In order to be generic, the quality manager should contain two specific subfunctions: a *Quality of Experience* (QoE) manager and an *operating point selection* manager (Figure 2.3).

The QoE manager deals with quality profile management, i.e. what are, for every application i, the different supported quality levels q_{ij} and how are they ranked. For example, a video application can make different trade-offs when it comes to framerate, resolution and image quality. Every application quality level q_{ij} is associated with a platform resource vector $\vec{r_{ij}} = \{r^p_{ij}, r^m_{ij}, r^c_{ij}\}$ that describes how much platform processing (r^p_{ij}), memory (r^m_{ij}) and communication (r^c_{ij}) resources are needed in order to support this quality. This resource vector can be determined either by the application mapping tools or by profiling an already mapped application. Every quality level q_{ij} also comes with its own set of implementation details d_{ij}. These implementation details are forwarded to the resource manager.

The QoE manager also has to determine the *value* every quality provides with respect to the user of the application. One could attribute a fixed value to every application quality level. However, this assumes that a certain quality level always provides the same value to the user. In reality, a certain quality level might provide a different user value or experience at different moments in time [79] depending on a range of parameters. Consider, for example, a mobile TV application. Intuitively it is easy to understand that the user quality needs (i.e. framerate, resolution, image size and image quality) will be different for viewing a commercial than for watching a sports event.In addition, the desired user video quality will also be different in case high quality implies that the battery will not last until the end of the movie or that other applications will have to be terminated. Hence, a *session utility function*[2] $f_u(\cdots)$ needs to be defined [30, 112, 121] in order to determine a relative value v_{ij} for every quality level q_{ij} at a certain moment. The user input and other parameters such as e.g. expected input data, platform information (like battery status) and time/location can be reflected in $f_u(\cdots)$. The end result is an ordered set of application quality levels q_{ij}, each with an associated value v_{ij}, resource vector $\vec{r_{ij}}$ and implementation details d_{ij}. The tuple $(v_{ij}, \vec{r_{ij}}, d_{ij})$ is further denoted as an application operating point.



Appl_i
$\{(q_{i1},\vec{r}_{i1},d_{i1}),(q_{i2},\vec{r}_{i2},d_{i2}), ..., (q_{in},\vec{r}_{in},d_{in})\}$
Application quality levels
Quality Manager
QoE Manager
$v = f_u(...)$
User application preference
Environment information
Platform information
$\{(v_{i1},\vec{r}_{i1},d_{i1}),(v_{i2},\vec{r}_{i2},d_{i2}), ..., (v_{in},\vec{r}_{in},d_{in})\}$
Application operating points
Operating Point Selection Manager
Platform resource information
$(v_{ik},\vec{r}_{ik},d_{ik})$
Selected operating point

Figure 2.3: *Run-time quality management. (1) The QoE manager receives a set of application quality levels and their associated resource needs from the design-time phase and attributes a value to each quality level according to a session utility function $f_u(\cdots)$. The result is a set of application operating points. (2) The operating point selection manager selects a good operating point according to a system utility function and with respect to the available platform resources.*

The operating point selection manager deals with selecting the best quality level or operating point for all active user applications given the current platform resource status and non-functional constraints like e.g. available energy. The goal of the operating point selection manager is to maximize the *system utility function* [30, 112, 121]. This means selecting an operating point $(v_{ij}, \vec{r_{ij}}, d_{ij})$ for every active application i such that the total system utility is maximized, while the total amount of required resources does not exceed the available platform resources. In its simplest form, the system utility function maximizes the total application value.

Quite some authors also use a quality manager on top of a resource manager to manage application and global system quality. Wust et al. [237], Bril et al. [30] and Van Raemdonck et al. [225] all provide such run-time management setups. Pastrnak et al. [171, 172] distinguish a design-time phase to estimate application resource needs and quality settings and, consequently, a run-time quality selection phase. In addition to a global quality manager, some authors also introduce a local, application-

[2] Also denoted as *job utility function* [30] or *application utility function*

specific quality manager that manages quality within the boundaries negotiated with the global quality manager.

Separating quality management from resource management and creating a set of well-defined (in the future, maybe even standardized) interfaces between them, enables reuse and interchangeability of these run-time management components. In addition, the separation supports the *platform based design* paradigm (Section 1.1.1) as it allows a global quality manager to work with a resource manager instantiated inside an application-service providing third party component.

Resource Management: Policies

Once an application operating point $(v_{ij}, \vec{r_{ij}}, d_{ij})$ has been selected, the resource requirements $\vec{r_{ij}}$ and the implementation details d_{ij} containing the application task graph (TG) are known. Some authors [112] also include in the implementation details d_{ij} a fixed allocation of platform resources. Such a fixed resource allocation greatly reduces the decision freedom of the resource manager. So, in order to remain flexible, one should rely on a run-time resource manager to map the application task graph (TG) onto the platform architecture graph (AG). In other words, to decide on the allocation of platform resources and to ensure the execution of this decision. This means that application tasks need to be assigned to processing elements, data to memory, and that communication bandwidth needs to be allocated. As Figure 2.4 illustrates, the resource management policy can be divided into a platform independent resource management algorithm and platform specific cost functions.

The different ways the task graph can be mapped onto the architecture graph define the mapping solution space. The resource management algorithm is responsible for the way this solution space is searched. Different types of algorithms can be used in this context depending on e.g. the trade-off of mapping speed versus mapping quality, or depending on the distribution of the mapping algorithm over the platform (see Section 2.3.1). Casavant et al. [221] provide a comprehensive taxonomy of those (platform independent) resource management algorithms in the domain of general purpose parallel and distributed systems. The decision making process of this algorithm relies on platform specific cost functions to determine the quality of a certain (maybe partial) mapping solution. This taxonomy is illustrated by Figure 2.5.

The first distinction of this taxonomy, static versus dynamic, is made based on *when* the task assignment is made. In both the static and dynamic domain, schemes can be divided into optimal and sub-optimal mapping solutions. In case all information regarding the problem is known and it is computationally feasible one can use an algorithm that derives the optimal solution (e.g. searching the full solution space). Within the algorithms that provide sub-optimal solutions, we can identify two categories: approximate and heuristic. The approximate solution uses the same algorithm as the optimal solution. However, instead of searching for an optimal solution, the algorithm is stopped as soon as a satisfactory solution is found. Heuristic solutions use some simple rules of thumb to determine a good (but often non-optimal) solution. All the optimal and sub-optimal/approximate techniques can be classified as based on searching the solution space by enumeration, on graph theory, on mathematical programming, or on queuing theory.

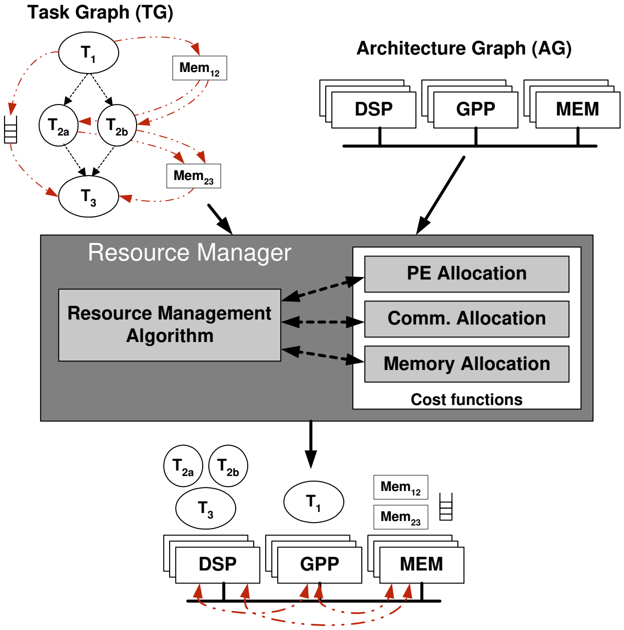


Figure 2.4: The resource manager maps a task graph onto an architecture graph. The policy for doing so contains a platform independent resource management algorithm and platform specific cost functions. The resource management algorithm determines how the mapping solution space is searched, while the cost functions allow the algorithm to assess the quality of a certain (maybe partial) mapping solution.

Besides the hierarchical scheme of algorithms, detailed in Figure 2.5, there is a flat classification that contains properties that can appear in any node of the hierarchy. The flat portion of the algorithm taxonomy handles properties like *adaptivity*, i.e. if the mapping policy takes the current and/or the previous behavior of the system into account, *load distribution* or *one-time assignment versus dynamic reassignment*, i.e. whether a resource assignment remains or can be revisited during the application execution. Tanenbaum [217] denotes these policies as *non-migratory* and *migratory* respectively.

So the platform independent part of the resource manager could be e.g. a heuristic, a branch & bound algorithm, a genetic algorithm, a full solution space search algorithm, etc. that uses the platform specific resource cost functions to determine the overall cost or quality of a certain TG to AG mapping. Intuitively, it is easy to understand that the cost of the communication resources, allocated between two processors, will be different in case these processors are interconnected by a simple bus or by a complex NoC. In a NoC, the cost of communication depends on the number of *hops* between source and destination tile, while this is not applicable in case of a bus. A similar reasoning holds for determining the cost of allocating tasks to different types of processing elements (e.g. DSP, accelerator or FPGA fabric tile) and for different types of memories (e.g. number of banks, ports, cost per memory access, etc.).

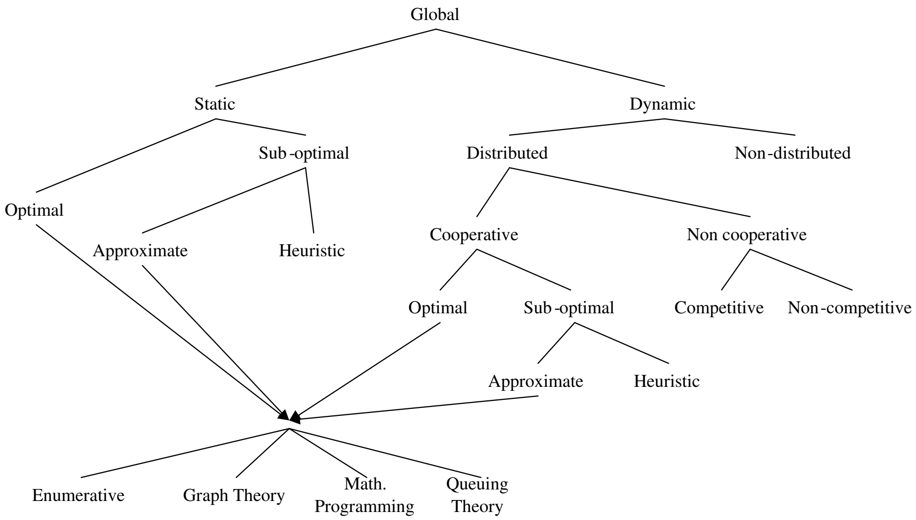


Figure 2.5: *Taxonomy of resource assignment algorithms (policies) for parallel and distributed systems [221].*

In recent years, a number of authors have considered the task graph (TG) to architecture graph (AG) mapping and resource allocation problem. Hu et al. [96] use a branch & bound algorithm to walk through the task graph TG to architecture graph AG mapping search space. Hansson et al. [91] use a greedy design-time heuristic to map an application graph of IP blocks onto an empty interconnection network graph. This design-time heuristic combines placement of IP blocks with NoC path selection and timeslot allocation. Recently, Stuijk et al. [215] employs a design-time heuristic to assign and schedule a task graph onto an architecture graph. Both the order at which tasks[3] are selected for assignment as well as the tile cost for a certain task is determined by a cost function. Several authors perform the mapping in two phases: a design-time phase and a run-time phase. Yang et al. [239,240] use a genetic algorithm for the design-time detection of the most promising task graph to architecture graph assignment and scheduling options. At run-time, they use a greedy heuristic [239] to select right options for all active task graphs. Ma et al. [127] extend this approach by introducing a run-time local search heuristic to further optimize the scheduling (not the assignment) produced by the greedy heuristic.

In Chapter 3, we introduce a heuristic to perform dynamic (run-time) assignment of resources based on cost functions specifically tuned for the on-chip interconnect and specific processor types. The processor cost function takes the specific properties of the FPGA processor tiles into account. Consequently, in Chapter 4, we turn it into a migratory algorithm by adding run-time task migration capabilities to the heuristic. In order to assess the heuristic performance, we compare it with a full search algorithm that searches for the most optimal solution.

[3] The author uses the term *actor* to refer to the task graph nodes, i.e. the tasks.

Resource Management: Mechanisms

The resource manager makes the resource allocation decisions. However, for executing these decisions, the resource manager has to rely on the underlying mechanisms. A mechanism describes a set of actions, the order in which they need to be performed and their respective preconditions or trigger events. In order to detect trigger events, a mechanism relies on one or more monitors. The action is performed by one or more actuators. The mechanisms are typically associated with the platform dependent parts of the resource management policies depicted in Figure 2.4. The resource management mechanisms closely collaborate with the run-time library in order to perform both basic allocation functions, like instantiating tasks on the processing elements, allocating memory blocks and setting up inter-task communication structures, and more complex functions like e.g. run-time task migration.

In this thesis, we introduce a set of mechanisms to support our resource management policies: two task migration mechanisms support the resource manager's task migration decisions. On-chip communication conflicts cause inter-application interference. Such situations are first detected by a monitor, then corrective decisions are taken by the resource manager according to the communication management policy. Finally, the actuator executes and enforces these decisions.

2.2.2 Associated Platform Run-Time Library

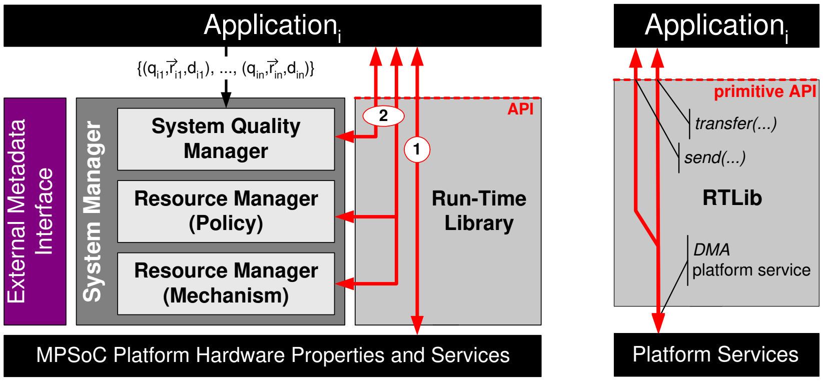
The *Run-Time Library* (RTLib), shown in Figure 2.2, essentially has two functions. First, it provides primitives to abstract the services provided by the hardware ((1) in Figure 2.6(a)). At design time, these RTLib primitives are used by the designer or the design tools. At run-time, the primitives are called by the application. This means that an RTLib implementation should be available on every processing element. Secondly, it acts as the run-time interface to the system manager ((2) in Figure 2.6(a)). In that sense, it plays an important role in enforcing the decisions of the quality manager and the resource manager.

With respect to the provided primitives, one can distinguish three types:

- **Quality management primitives.** These primitives link the application or the application-specific quality manager to the system-wide quality manager embedded in the run-time manager. This allows the application to e.g. renegotiate the selected quality level [171,172,237].

- **Data and communication management primitives.** These primitives are closely linked to the programming model. For example in a programming model where tasks communicate by passing messages, the typical primitives require *send()* and *receive()* primitives. To allocate memory blocks, either for processor-local or shared memory, one requires *malloc()* and *free()* alike primitives. Finally, the RTLib can also provide primitives to manage the memory hierarchy. In order to manage scratchpad memory a designer would need a *transfer()* function to move arrays or parts of arrays through the memory hierarchy [31]. As Figure 2.6(b) illustrates, both the *transfer()* and the *send()* RTLib primitive can rely on the platform hardware DMA service to transfer data to move data to another memory hierarchy layer or the destination task respectively. The RTLib is responsible for configuring the DMA service. In case of software controlled caches, the RTLib could provide *invalidate()* and *prefetch()* primitives [244].

- **Task management primitives.** These primitives allow a designer to create and destroy tasks (e.g. *spawn()* and *join()*) and manage their interaction (e.g. with semaphores, mutexes and barriers). The RTLib can also provide primitives to select a specific PE-local scheduling policy or to explicitly invoke the local scheduler (e.g. *yield()*). In addition, there are primitives used for task migration or operating point switching (e.g. *migrationpoint()* or *switchpoint()*). These primitives actually interface to the resource manager and the quality manager respectively ((2) in Figure 2.6(a)).


Application$_i$
$\{(q_{i1},\vec{r}_{i1},d_{i1}), \ldots, (q_{in},\vec{r}_{in},d_{in})\}$
External Metadata Interface
System Manager
System Quality Manager
Resource Manager (Policy)
Resource Manager (Mechanism)
2
1
API
Run-Time Library
MPSoC Platform Hardware Properties and Services
Application$_i$
primitive API
transfer(...)
send(...)
RTLib
DMA platform service
Platform Services

(a) (b)

Figure 2.6: Run-Time Library roles. (a) Primitives to abstract the hardware service$^{(1)}$ *and primitives that provide an interface to the system manager*$^{(2)}$*. (b) Both the* send() *and* transfer() *RTLib data management primitives rely on the DMA hardware service.*

The run-time RTLib cost will depend on its implementation: ranging from a software implementation executing on the processor that also executes the application tasks to a separate hardware engine next to the application processor. For example, Paulin et al. [175, 176] use a *hardware message passing engine* to reduce the inter-task communication cost. In the Eclipse hardware template, Rutten et al. [192] use a hardware interface block, denoted *hardware shell* between the compute engines and the communication hardware. This *shell* is instantiated at every compute engine and acts as a run-time library. The shell provides its computation engines with five generic interface primitives for task management and data management. Section 2.3.2 discusses the trade-offs between hardware and software in more detail.

In Chapter 4 we show how our RTLib plays an important role during the execution of the task migration mechanism. In our case, heterogeneous task migration happens in a cooperative way: the system management signals the RTLib that a certain task needs to migrate. Consequently, the RTLib waits until the application task signals it is ready to be migrated, i.e. calls the *migrationpoint()* function. When that happens the RTLib, in turn, informs the system management and performs a series of actions to proceed with the actual migration. In Chapter 6, we show that our RTLib is implemented as a separate 16-bit processor instantiated alongside every application processor.

2.3 Run-Time Management Implementation Space

Section 2.1 explains the boundary conditions the MPSoC run-time manager has to operate in. Section 2.2 details what functions a run-time manager needs to perform. However, there still is an implementation design space where run-time management implementation trade-offs can be made.

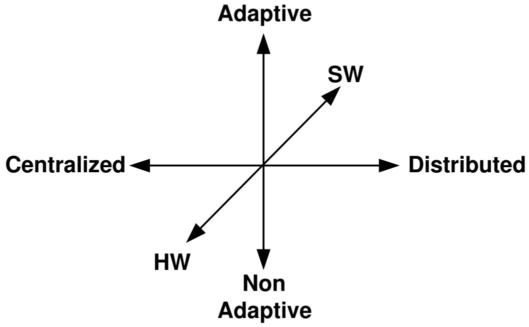


Figure 2.7: Run-time manager design space: the run-time manager can be implemented in hardware or software, distributed or centralized, generic of tuned towards the application.

In general, one can identify three run-time management implementation space axis (Figure 2.7). The first axis deals with the amount of distribution of the run-time manager. The second axis details the spectrum between a hardware and a software implementation. The third axis makes a trade-off between a generic or a flexible or domain specific implementation. By choosing the right implementation point, it should be possible to design an efficient run-time manager that meets the application needs and that satisfies the key MPSoC requirements. The rest of this section describes the run-time management implementation space and illustrates the trade-offs that can be made with research and industrial examples.

2.3.1 Centralized versus Distributed

As we are dealing with multiprocessor systems, one can distinguish a design space that deals with the amount and type of distribution of run-time management functionality. Figure 2.8 details the classic multiprocessor run-time management categories [76, 204] based on how the processing elements share run-time management responsibilities.

- **Master-Slave configuration.** In a master-slave configuration, there is a single master processor that executes the run-time manager, i.e. that monitors and

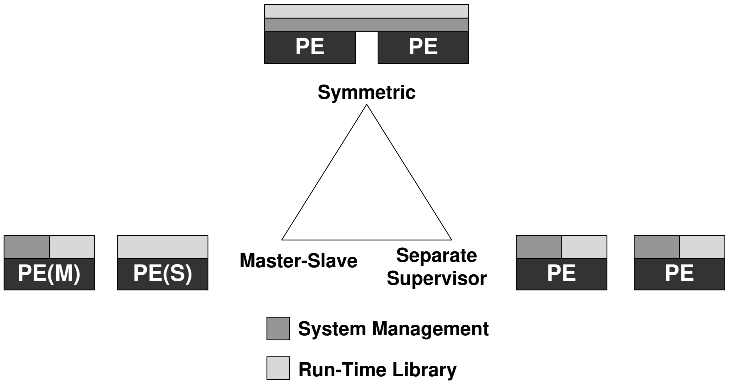


Figure 2.8: Design space for multiprocessor systems with respect to distribution of run-time management responsibilities.

assigns work to the slave processors. In addition, the master is responsible both for part of the computation and the I/O jobs. The slave processors only execute user applications code. This means that a run-time library is instantiated on the master and on every slave processor. The slave processors have to wait while the master is handling their calls to the system manager. The benefits of this type of run-time management implementation are its simplicity and efficiency when the slaves are mainly used for compute intensive jobs. As there is only one processor executing the system manager, synchronization with respect to shared resources can be implemented in a straightforward way. The single master is also the main disadvantage: it risks becoming a single point of failure or a bottleneck that fails to serve the slaves with enough work.

- **Separate Supervisor configuration.** In this case, every processor executes its own run-time management functionality and has its associated run-time library and data structures. Hence, each processor acts as an independent system. Special structures and mechanisms exist to achieve global system management, i.e. collaboration across the processor boundary. This type of system is scalable, gracefully degrades in case of processor failure, and a single processor cannot become a management bottleneck. Unfortunately, due to duplication of data structures there is a memory penalty. In addition, there is a management overhead in optimally controlling and using the system resources with respect to the user application.

- **Symmetric configuration.** There is a single run-time manager executed by all processors concurrently. Access to shared data structures needs to be handled by critical sections. Although the symmetric configuration is the most flexible of all configurations, it also has some downsides. First, in contrast to the other configurations, this configuration requires a homogeneous set of processing elements with a shared memory. Second, it is the most difficult one to implement in an efficient way. Similar to the Master-Slave configuration, the execution of the run-time manager can become a bottleneck. This can be solved by allowing multiple processors to have concurrent access to disjoint run-time management components. Hence, the scalability of this system lies between a Master-Slave configuration and the Separate Supervisor configuration.

Furthermore, it is easy to envision an MPSoC system that uses a mix of configurations. As Section 2.4 details, the next generation ST Nomadik platforms, for example, will combine a Symmetric configuration with a Master-Slave configuration [175].

The taxonomy presented by Casavant et al. [221] (Figure 2.5) also handles the distribution of resource management policies. In the realm of dynamic algorithms one can distinguish distributed and non-distributed algorithms, based on whether the resource assignment is done by a single processor (i.e. the Master-Slave configuration) or is distributed among multiple processors (i.e. Symmetric or Separate Supervisor configuration). Depending on the existence of interaction between the different processors, distributed resource management can be split into cooperative and non-cooperative. Non-cooperative resource management means individual processors make assignment decisions independent of the actions of the other processors. Cooperative resource management still means that processors make their own decisions, but they collaborate to reach a common system-wide goal. For example, one of the distributed processor management algorithms in the Cosy operating system [37] is based on the principle of leaking water. From an application entry point, an amount of water (workload) leaks and flows into the direction where the resistance is lowest (lowest processor load).

In this thesis, we focus on a Master-Slave configuration.The main motivation for choosing the Master-Slave configuration is the fact that we use a relatively small number of powerful heterogeneous slave (between 4 and 16) processors for computationally intensive tasks. In addition, the slaves only perform a very limited amount of run-time manager calls and they can communicate data with each other without master processor intervention. As Chapter 6 explains, this configuration is achieved by extending an existing single processor real-time operating system to handle a set of heterogeneous slave processing elements.

2.3.2 Hardware versus Software

System management functionality or a run-time library is typically implemented in software (e.g. software scheduler) by building on the low-level hardware services provided by the platform (e.g. timer interrupt service). In recent years, also fueled by the MPSoC revolution, quite some run-time management functions have been implemented in additional hardware.[4] Hence, one can identify a *hardware versus software* design axis that is applicable to both the system management and the run-time library. The main motivation for implementing part of the run-time manager in hardware or a separate accelerator is to avoid the *overhead* caused by executing the run-time management functionality on the application processor [174–176, 219].

Furthermore, a (more) dedicated implementation of run-time management functionality should satisfy the MPSoC run-time manager environment constraints (see Section 2.1). First, besides being significantly faster than its software counterpart

[4] This often means an additional processor or programmable IP block that provides run-time management services.

[148,175,176,195,219], a hardware implementation holds the promise of being more energy efficient [94]. Secondly, both the maximum response time and its variance decrease, which improves the real-time behavior of the system [174]. This is partly caused by the fact that there is significantly less cache space needed for the run-time management functionality. In addition, the memory footprint of the run-time manager decreases [98]. Finally, by implementing this functionality in a separate block, the run-time management complexity caused by the heterogeneity of multiple platform processing resources is mitigated [115]. Indeed, one can combine *heterogeneous* application processing with *homogeneous* run-time management. As Rutten et al. [192] explain, their *hardware shell* takes care of all system-level run-time management issues, while the compute engine designers focus on application functionality.

Example 2.1: Scheduling in hardware or in software (Figure 2.9).
Consider two tasks executing on a single application processor. In case of software scheduling, the scheduler is invoked at regular intervals by a platform timer interrupt (action (1) and (2)) or it is called when an event occurs (e.g. (3) when semaphore changes or a message arrives). This causes overhead because the scheduler needs to be executed on the application processor even when the newly selected task ends up being the same task (action (2)). When moving this functionality to a platform hardware service (Figure 2.9(b)), the run-time overhead is kept to a minimum. Instead of interrupting the executing task to attend to the management functionality, the decision making is done in parallel. This means that valuable application processor cycles are not wasted while taking management decisions. When a decision is made by the platform hardware service, the processor specific actions, like the task context switch, still need to be performed by the processor itself. In addition, decreasing the management time granularity in case of a software scheduler (i.e. time between clock interrupts) creates a proportional increasing overhead. In contrast, a platform hardware service can work at a fine granularity without causing additional application processor overhead.

In general, most of the state-of-the art [174–176, 191, 192, 219] focuses on implementing run-time library functionality in hardware. This mainly includes making scheduling decisions for the (local) application processor or hardware accelerator, providing support for memory management and handling inter-task communication and synchronization. However, one can also rely on a hardware block to perform task to processor assignment in a multiprocessor environment [98,147]. Finally, hardware support is also used for collecting run-time information. This involves non-intrusive monitoring of what is happening on the platform. In real-time systems, it is important to minimize the intrusiveness of the monitoring, i.e. it should not alter the system behavior. This can be achieved by adding the monitoring functionality in hardware [201].

There is indeed a spectrum with a full hardware implementation on the one hand, like e.g. the *Real-Time Unit* (RTU) [115], and a software implementation on the other hand. In between, one finds configurable run-time management accelerators or combined HW/SW solutions. These platform run-time management services attempt to find the sweet-spot between acceleration and the flexibility to change the policies

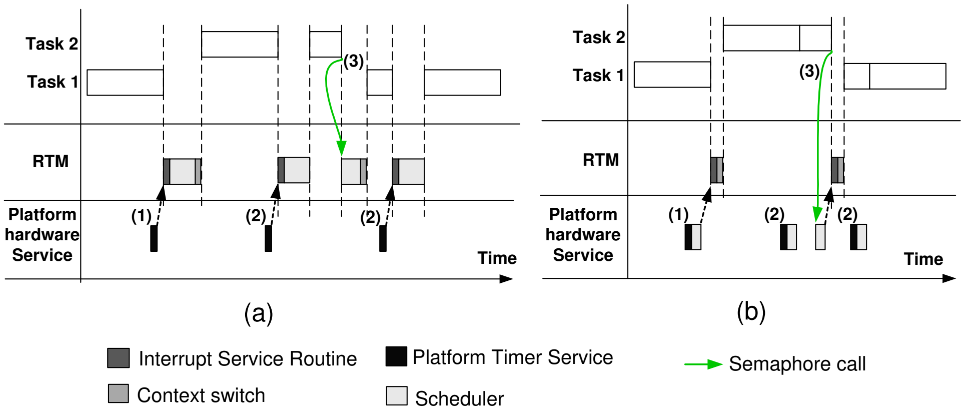


Figure 2.9: Hardware or Software? (a) A hardware timer service periodically invokes a software scheduler, which causes fewer processor cycles to be available for the actual user application. (b) The scheduler is implemented in additional hardware and only interrupts the application processor when a context switch is needed.

or to adapt to existing software run-time managers. A downside of pure hardware acceleration is the limited application scalability. For example, the RTU is limited to handling 16 tasks at 8 priority levels, 16 semaphores and 8 external interrupts. Similarly, the *Task Control Unit* (TCU), described by Theelen et al. [219], is limited to supporting 63 independent tasks and 128 communication resources (i.e. semaphore, mailbox or pipe). However, as Paulin et al. [176] point out, one has to make a trade-off between speed and deterministic execution on the one hand and flexibility/scalability on the other. Furthermore, existing software operating systems also can have limitations on the number of simultaneous user processes or file descriptors. Limitations are acceptable if tuned to the application domain.

Another way of dealing with the *inflexibility* of hardware is to use reconfigurable hardware (explained in-depth in Chapter 3). Depending on the application needs, one can reconfigure the number of supported tasks as well as the scheduling algorithm [117] of the hardware run-time management block.

In our proof-of-concept platform (Chapter 6), every tile contains a small 16-bit processor responsible for tile-local run-time library (RTLib) tasks next to the application processor. Our main motivation is to facilitate the integration of heterogeneous processing elements: combining heterogeneous processing with homogeneous run-time management. This also ensures an uniform, predictable timing for the RTLib function calls (i.e. independent of the associated application processor). In addition, it facilitates the use of FPGA fabric processor tiles as these tiles are not suited for running a traditional software run-time library.

2.3.3 Adaptive versus Non-Adaptive

A general purpose operating system for the desktop PC or workstation attempts to provide fast response time for interactive applications, high throughput for batch applications and an amount of fairness between applications [185]. It relies on generic application-independent heuristics to achieve this [29].

Dealing with the MPSoC run-time management requirements and constraints in an efficient way can also be achieved (1) by tuning the run-time manager with respect to the application or by exploiting application knowledge that is either gathered at run-time or received by the application design flow, and (2) by tuning the run-time manager as to take into account the platform properties and the offered hardware services. One can consider two categories of adaptation of the run-time manager to the application: *design-time adaptation*, i.e. designing the run-time manager in such a way that it suits the needs of the application and platform, and *run-time adaptation*, i.e. changing the behavior of the run-time manager according to the current application(s) or even deferring some system management responsibility towards the application.

As Figure 2.10 illustrates, design-time adaptation of the run-time manager relies on a library of parameterizable and configurable run-time management components [77, 78]. A run-time manager generation tool is responsible for creating the run-time manager with the right functionality. This tool takes as input, for example, a specification of the architecture, the memory and resource allocation map and a high-level (i.e. configurable) description of the application tasks. By analyzing the needs of the application tasks, like e.g. communication and synchronization needs, and by looking at the available platform services, the needed run-time management functionalities are instantiated for every processing element.

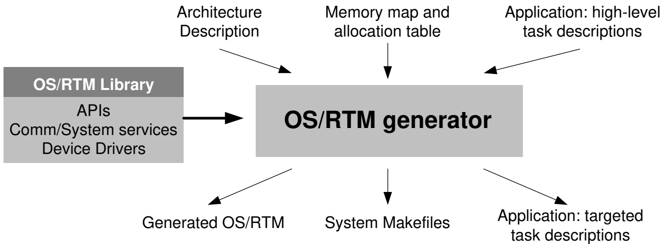


Figure 2.10: *high-level flow for automatic generation of application-specific run-time management and automatic application software targeting [77, 78].*

Run-time adaptation exploits a closer relationship between the run-time manager and the executing application(s). This has typically been the focus of run-time managers included in adaptive multimedia systems [112, 154, 184, 185] and of adaptive operating systems such as the Exokernel [72, 106]. Three types of negotiation and adaptation between run-time manager and application can be identified (Figure 2.11).

The first type (type 1) occurs when the run-time manager supports *adaptive applications* (Figure 2.11(a)). Adaptive applications can be defined as applications that support multiple modes of operation along one or more resource and/or quality dimensions [30, 48, 51, 52, 105, 112, 185]. Each application operating mode has its own resource requirements and offers some degree of value towards the user. This way, the run-time manager is able to select the most appropriate operating mode for each executing application. This approach promises higher user value than a simple

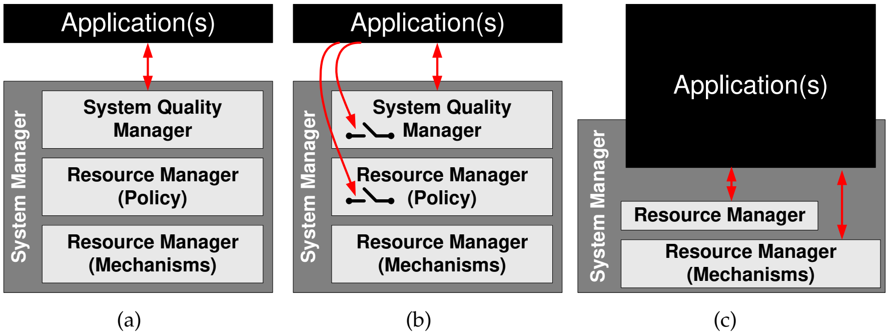


Figure 2.11: *Three types of run-time adaptation: (a) run-time manager support for adaptive applications (type 1), (b) the application also configures parts of the run-time manager policies (type 2), (c) the application takes over part of the run-time manager responsibilities (type 3).*

application accept/reject policy, but requires communication of application design-time analysis information to the run-time manager as well as a run-time manager capable of handling this information.

The second type (type 2) involves the application configuring parts of the run-time manager policies (Figure 2.11(b)). This way, generic policies can be tuned specifically towards the needs of a certain application. This results in better decision-making and a more optimal usage of platform resources. Mamagkakis et al. [129] describe a technique in which the dynamic memory allocator is configured depending on the requesting application. This does not only require the communication of application design-time analysis information towards the run-time manager (as in the first type), this also requires a run-time manager with configurable management policies.

The third type (type 3) occurs when the application takes over part of the run-time manager responsibilities (Figure 2.11(c)). This means that application-specific management is handled within the application itself. The run-time manager is responsible for allocation or multiplexing of the hardware resources. Noble et al. [154] detail a set of extensions to the NetBSD operating system. In their setup, the application requests a set of platform resources and essentially manages these resources to provide a certain quality level with respect to the user. At the extreme end resides the MIT Exokernel [72, 106]. The Exokernel's sole function is to allocate, deallocate and multiplex the physical platform resources. The application (or application library developer) is responsible for building the necessary hardware abstractions and for managing the allocated resources in an efficient way.

As Chapter 7 explains, we focus on a run-time manager of the first type. This means that the run-time manager accepts the design-time application analysis information, but the system management policies or mechanisms are not getting (re)configured according to the application. This decision is motivated by the fact that it is the solution with the largest *industrial relevance*[5] for three reasons. First, the application design flow does not need fundamental changes. If needed, the designer can continue with the traditional design flow and provide the application analysis information

[5] Industrial adoption is one of the key performance indicators within IMEC.

afterwards. The more innovative application design flows (see Appendix C) will help the designer in creating this information. Second, platform run-time manager functionality is complex. Hence, the industrial providers are more likely to be persuaded into putting an additional component or into tuning a component for a specific application domain and/or platform than into allowing an application change the behavior of the run-time manager. This distinction is critical especially when considering the need for ensuring real-time and deterministic behavior. Third, this approach does not require any platform specific knowledge from the application designer.

2.4 Multiprocessor Run-Time Management Examples

To substantiate our MPSoC run-time management design space, this section provides a selection of run-time management examples[6] for industrial and academic, large-scale, board-level and SoC multiprocessor platforms (Table 2.1 and Table 2.2).

Texas Instruments
Cumming et al. [54] detail the run-time management approach used by *Texas Instruments* (TI) to support its OMAP MPSoC platforms (Figure 2.12(a)). In essence, the run-time manager is a software implementation of a Master-Slave configuration. TI created a small, real-time embedded RTOS, denoted DSP/BIOS, as to provide a dedicated run-time library (RTLib) for its DSP processing elements. The DSP/BIOS kernel provides basic communication primitives and task scheduling functionality on top of the DSP hardware; so application developers can build modern multi-threaded applications in an easy way.

TI developed the *DSP/BIOS Link* software to support the e.g. OMAP SoC platform, where a general purpose master RISC processor is combined with one or more slave DSP processing elements on a single die. The DSP/BIOS Link software links a standard, independent operating system executing on the general purpose master to the DSP/BIOS kernel executing on the slave DSP. The purpose of the DSP/BIOS Link is to provide a control and communication API between the general purpose tasks and the DSP/BIOS tasks. In addition, it allows the master to boot the slave DSP(s) and to control which algorithms they execute for a specific application.

As Figure 2.12(a) illustrates, there is a system resource management component, linked to the operating system and executing on the general purpose RISC processor. This master resource manager communicates with its counterpart(s), denoted as *RM server*,[7] executing on top of the BIOS kernel. The resource manager is responsible for selecting and allocating the slave DSP, for task creation and for setting up the communication structures, for starting and stopping the tasks and, finally, for deallocating the resources.

[6] The comparison is based on publicly/freely available information.

[7] In the microkernel world, only the most basic functionality (BIOS) is incorporated into the RTOS kernel. The run-time management or operating system functionality responsible for more high level tasks, such as resource management, execute outside the RTOS kernel and are commonly denoted as *servers*. Hence the name RM server for the resource management functionality.

TI provides a porting kit that allows easy porting of their MPSoC run-time manager to other general purpose operating systems. Quite some (real-time) operating system vendors have also used this option: e.g. QNX Neutrino RTOS (QNX Software Systems), INTEGRITY RTOS (Green Hills Software) and Linux (MontaVista).

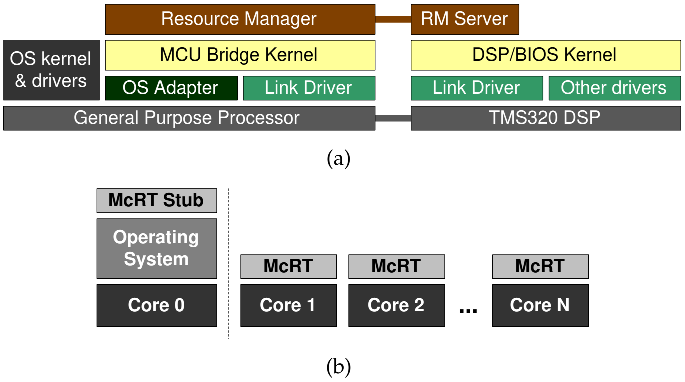


Figure 2.12: *Run-time management approach for (a) today's TI OMAP MPSoCs [54] and (b) for tomorrow's Intel Tera-scale platforms [193].*

Enea Systems

Similar to TI, *Enea Systems* presents a Master-Slave configuration solution for heterogeneous multiprocessors systems by combining the feature-rich *OSE RTOS* executing on the master with a compact DSP kernel, denoted *OSEck*, for the slaves. A *link handler* provides message passing inter-processor communication primitives.

Quadros

The Quadros RTXC RTOS provides its own solution for multiprocessor platforms (e.g. OMAP platform). This RTOS can handle MPSoCs as well as a set of DSPs on a board or a loose collection of PEs (either heterogeneous or homogeneous). Their approach is to duplicate the RTOS kernel services on every PE. A *link manager* provides an easy way for tasks on different PEs to communicate. The link manager relies on an high-level message-passing inter-processor communication service. This communication service abstracts the underlying platform hardware communication service. In that sense, the Quadros solution can be classified as a more Separate Supervisor approach. However, the provided solution mainly focuses on providing RTLib functionality for the application designer. The designer is responsible for deciding on the resource allocation, so there are no actual run-time resource manager policies present. Quadros allows design-time adaptation towards the needs of the application. The *RTXCGen* tool allows the designer to easily configure the kernel to fit the processing requirements of your application and to only include the required kernel services. There is no support for run-time adaptation.

ARM MPCore & Linux SMP

The ARM MPCore is a homogeneous embedded multiprocessor platform that relies on a general purpose operating system, like Linux, that supports a Symmetric configuration. This way, the operating system actually hides the fact that multiple processing engines are available which enables an easy speed-up of the applications:

either because the application is composed of multiple tasks that execute on different processing elements or because multiple single-task applications no longer have to share a single processing element. Linux includes a scheduler with a load balancing policy [29]. The *Native Posix Thread Library* and the C library act as RTLib [83].

RTOS vendors, like e.g. Express Logic with their *ThreadX* RTOS, focus on providing SMP support for the MPCore to increase performance [38]. In contrast to Linux, such an RTOS is faster and more deterministic. Indeed, the RTOS does not have a user-kernel space boundary, it has some simple, yet efficient and fast ways for determining the task schedule, and it provides plenty of scheduling opportunity [234]. In addition, it has a smaller memory footprint. While ThreadX also provides some degree of automatic load balancing (i.e. resource management functionality), the RTOS mainly focuses on providing RTLib functionality. ThreadX does not provide any design-time RTOS tuning tool (like the Quadros RTXC), but it does provide the full RTOS source code. This equally enables the designer to only include the required components.

NetBSD & Odyssey
Odyssey [154] extends the NetBSD operating system and, hence, supports a Symmetric configuration. Odyssey provides a collaborative partnership between the system manager and the application. The system manager is responsible for resource arbitration, i.e. for making resource allocation decisions, for enforcing these decisions, and for notifying the applications about these decisions. Then, every application independently decides on how to best adapt its provided quality given the resource constraints. Hence, the Odyssey extensions enable type 1 application adaptivity: the different application quality levels are first provided to the system manager which takes them into account when deciding on the resource allocation.

RealFast AB
RealFast AB developed a *Real-Time Unit* (RTU) [115]. The RTU is a commercial hardware IP block that provides RTLib functionality for the (heterogeneous) on-chip processors. Communication with the application and the system management happens with memory mapped registers and interrupts. The RTU is linked to a general purpose OS and manages the application processors. The *Silicon TRON* project [148] and the *Task Control Unit* (TCU) of the MμP architecture [219] provide a similar hardware solution.

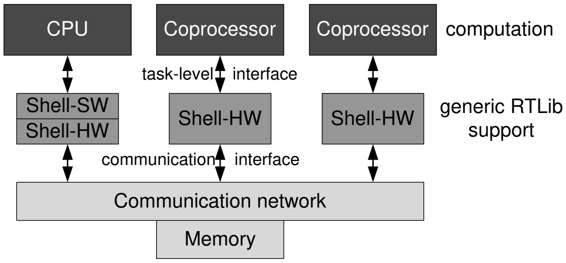


Figure 2.13: *The Eclipse architecture template [191, 192] combines a general purpose processor, denoted as CPU, with application-specific coprocessors. The* hardware shell *provides RTLib functionality and it represents the interface between computation and communication.*

Eclipse

The Eclipse architecture template (Figure 2.13) defines a heterogeneous multiprocessor to be used as a flexible and scalable subsystem for MPSoC platforms [191, 192]. Its target application domain is stream processing (e.g. video processing). Eclipse combines the application flexibility of a general purpose processing element (i.e. the CPU) with the efficiency of application-specific hardware processing, denoted as *coprocessors*. The *hardware shell* acts as an interface between processing and communication. As it provides RTLib functionality, it alleviates the coprocessor designer from having to worry about system-level issues like synchronization, data transport and scheduling. The entire system is conceived as a Master-Slave configuration: a general purpose processor is responsible for configuring the coprocessors and handling their reported events.

STMicroelectronics MultiFlex

For its next generation Nomadik platforms, ST Microelectronics has developed a dedicated approach for designing applications and performing run-time management [175, 176] denoted as *MultiFlex*. These new Nomadik platforms contain multiple general purpose processing elements executing e.g. Linux, Symbian or WinCE in a Symmetric configuration. In addition, the platforms contain multiple specialized DSPs and ASIPs for handling video, audio and 3D algorithms. These processors act as slave processing elements and receive their tasks from the general purpose processing element cluster. The DSPs and ASIPs rely on hardware schedulers and hardware message passing engines for efficient scheduling and inter-task communication respectively. This approach combines a Symmetric configuration with a Master-Slave configuration. In addition, the platform contains hardware services for providing the most critical run-time library functions. Except for the close relation between the MultiFlex design flow and the MultiFlex run-time management components, there is no adaptivity with respect to the application.

Cosy

The *Cosy* microkernel operating system was designed and optimized for board-level multiprocessor and multicomputer platforms [36, 37]. Cosy has a strong focus on providing the right platform abstractions to ease application development. This includes providing primitives for starting tasks (i.e. components of a parallel application) at run-time and for inter-task communication. Run-time resource assignment can be performed manually by the designer or automatically by Cosy. In this context, Cosy expects an application task graph that can then be mapped by several run-time mapping functions. Cosy implements a Separate Supervisor configuration and interacts with the application in a sense that it takes the task graph into account to allocate platform resources. Cosy is too heavyweight for SoC platforms. Other similar microkernel operating systems are Amoeba [64, 216], Sprite [63, 64] and Mach [26, 216].

AsyMOS

The *AsyMOS* run-time manager [146] assigns specific functionality to specific processors in a symmetric, shared memory multiprocessor system. This means that some processors are assigned to handle the application code, while others perform system management functionality. This solution is located between a Symmetric configuration and a Master-Slave configuration. It simplifies run-time management and reduces the amount of interrupts and cache contention on the application processors, which increases performance and predictability. AsyMOS also allows an application to insert its specific resource management components (i.e. type 3 adaptivity).

K42
K42, an IBM research run-time manager for 64-bit cache-coherent multiprocessor platforms, focuses on high performance, platform scalability and application adaptivity [13]. Although K42 implements a Symmetric configuration on the surface, every run-time manager resource object and associated data structures can be distributed in an efficient way over the multiprocessor in order to exploit the use of local memory and to avoid global data structures, global data locks and global management policies. Just like a Separate Supervisor approach, this approach provides near-linear scalability [13]. The system management can also be adapted to the application needs by allowing the application (designer) to select the right combination of provided system management building blocks (i.e. type 2 adaptivity). K42 provides a scheduling infrastructure that supports real-time behavior, resource time-sharing, gang scheduling, and synchronized locks.

MIT Exokernel
Although similar to K42, the *MIT Exokernel* [71, 72] takes application adaptivity to the extreme: it simply allocates, deallocates and multiplexes physical resources like e.g. memory and processor time-slices. All quality and resource management are pushed into the application-level software: either into user-level libraries or into the application itself. This is type 3 adaptivity.

Intel McRT
Intel recently published its view on the *runtime environment for Tera-scale platforms* [193]. Their *Many-Core RunTime* (McRT) environment supports heterogeneous platforms as they see future platforms containing high performance scalar cores as well as an array of high throughput cores and accelerators. The McRT essentially controls the platform resources in a more distributed and cooperative way, while it is linked to a traditional (Master) Operating System that provides all non-core functionality. The McRT RTLib functionality provides the application designer with parallelism primitives like e.g. threading and synchronization services. In addition, it provides primitives to support fine-grain atomic memory transactions (also denoted as *transactional memory* support). The fine-grain synchronization and scheduling services are partly implemented in hardware. Finally, the McRT RTLib is able to translate popular APIs like e.g. the OpenMP API and the PThreads API into the core McRT API. In order to accommodate the various application requirements, the McRT supports both design-time and run-time adaptivity. The design-time adaptivity is achieved by only including those McRT modules that are required, while run-time adaptivity is achieved by providing configurable scheduling policies that allow it to flexibly adapt to specific application needs (i.e. type 2 adaptivity). The scheduling scalability bottleneck is addressed by using cooperative scheduling.

Table 2.1 and Table 2.2 provide an overview of the discussed run-time management solutions. Table 2.1 details their available functionality, while Table 2.2 describes their implementation solution. An asterisk (*) indicates the run-time management solutions for embedded platforms.

Table 2.1: *Run-time Manager implementation space. Multiprocessor run-time manager solutions for embedded* platforms and large-scale parallel systems.*

RTM Functionality	Quality Manager	Resource Manager	Resource Management Mechanisms/RTLib
TI OMAP*	-	√	√
Linux*	-	√	√
ThreadX*	-	√	√
Quadros RTXC*	-	-	√
ST MultiFlex*	-	√	√
RealFAST RTU*	-	-	√
Eclipse*	-	√	√
Odyssey*	√	√	√
Cosy	-	√	√
AsyMOS	-		√
K42	-	√	√
MIT Exokernel	-	√	√
Intel McRT	-	√	√
IMEC RTM	√	√	√

Table 2.2: *Run-time Manager implementation space. Multiprocessor run-time manager solutions for embedded* platforms and large-scale parallel systems.*

RTM Implement.	Distribution	Adaptivity	HW/SW
TI OMAP*	Master-Slave	None	SW
Linux*	Symmetric	None	SW
ThreadX*	Symmetric	Design-Time	SW
Quadros RTXC*	Separate Supervisor	Design-Time	SW
ST MultiFlex*	Master-Slave & Symmetric	None	HW RTLib
RealFAST RTU*	Master-Slave	None	HW RTLib
Eclipse*	Master-Slave	None	HW RTLib
Odyssey*	Symmetric	Run-Time, type 1	SW
Cosy	Separate Supervisor	None	SW
AsyMOS	Master-Slave	Run-Time, type 3	SW
K42	Symmetric with Separate Supervisor properties	Run-Time, type 2	SW
MIT Exokernel	Symmetric with Separate Supervisor properties	Run-Time, type 3	SW
Intel McRT	Master-Slave with Separate Supervisor properties	Design-Time and Run-Time, type 2	SW with HW RTLib support
IMEC RTM	Master-Slave	Run-Time, type 1	HW RTLib support

Concluding Remarks
It is clear that, with respect to heterogeneous multiprocessor platforms, the contemporary industrial run-time management functionality is mainly focused on providing resource management mechanisms, e.g. support for starting and terminating an application, and RTLib functionality, i.e. on providing the application designers with an abstraction layer on top of the hardware. More academic approaches, like Odyssey and Cosy, provide a resource manager. Quality management still appears to be a research topic. With respect to the implementation space, we see a trend for moderate distribution of resource management functionality although the amount of distribution really depends on the MPSoC platform and the application domain. Hardware support for run-time management is also on the rise, with RealFAST already providing a commercial solution today. The ST MultiFlex approach also relies on hardware RTLib support to reach high performance combined with a low power and deterministic operation. Adaptivity is still in its infancy: academic embedded solutions like Odyssey provide run-time adaptivity, while the Quadros provides a commercial tool to tune their run-time manager, at design-time, to the needs of the application. Except for the ST MultiFlex approach, all current commercial solutions have not been designed for the emerging MPSoC environment, but are based on extending or linking together existing technology. Finally, the Intel view on run-time management for Many-core platforms provides a peek into the future, where distribution of run-time management, configurability and hardware support are predicted to be mainstream.

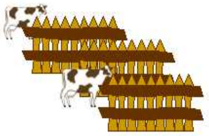
2.5 Conclusion

After describing the boundary conditions for MPSoC run-time management solutions, this chapter details the different components of the run-time manager together with the functionality they provide. At a high level, the run-time manager contains a system management component and a run-time library. In turn, the system management component contains a quality manager and a resource manager. The quality manager negotiates quality levels with the applications according to the run-time needs of the user. The resource manager makes the resource allocation decisions according to a certain policy and it orchestrates the execution of these decisions through the associated mechanisms. Through the run-time library, the run-time manager provides hardware abstraction services that are used by the application designer and called by the application at run-time.

This chapter also provides the first description of the MPSoC run-time management implementation design space. This design space contains three axes: the first axis deals with the amount of distribution, the second axis depicts the hardware versus software trade-off space and the third axis deals with the amount of run-time management adaptivity towards the application and the platform. This chapter also puts the remainder of this thesis into perspective with respect to the selected run-time management functionality and the design decisions for our proof-of-concept implementation. We provide a system manager containing a quality manager and a resource manager with an associated RTLib. The separation of quality manager and resource manager enables their re-use and better supports the platform-based design paradigm. As we consider platforms with a relatively small number of powerful heterogeneous processing elements, we selected a Master-Slave configuration. The RTLib is implemented as a separate programmable processor instantiated next to every application processor as it enables heterogeneous processing with homogeneous management. We selected type 1 adaptivity, as this form of adaptivity is (1) supported by the IMEC mapping tools and (2) most likely to be adopted by the industry.

Finally, this chapter briefly details some contemporary industrial and academic multiprocessor run-time management solutions and takes a peek into the future. It is clear that, with respect to MPSoC platforms, the industrial run-time management is mainly focused on providing RTLib functionality, i.e. on providing the application designers with a hardware abstraction layer. Although currently, industry does not provide any real resource or quality management functionality, the ST MultiFlex effort shows that this will be present for future platforms. Academia have developed experimental multiprocessor operating systems that provide advanced run-time management capabilities and that explore the run-time management design space for more optimal solutions with respect to resource management and adaptivity. However, these solutions often need re-targeting towards embedded platforms.

However, the (research) trend for MPSoC run-time managers is clear: moderate distribution of the run-time manager over the platform resources, more platform services to support the run-time manager and more configurability towards the application. Hardware run-time management components and distribution of the run-time management functionality is likely to arrive first as this can be provided by MPSoC platform and run-time management vendors. Adaptivity requires a close collaboration between run-time manager and developer and/or design tools. Such solutions probably require more effort to deploy on an industrial scale.



CHAPTER 3

Task Assignment in an MPSoC containing FPGA Tiles

In order to meet the ever-rising compute requirements while retaining platform flexibility, SoCs contain multiple heterogeneous processing elements. This means that besides general-purpose processing elements, there will be some specialized processing elements (e.g. a DSP) that only perform a limited number of tasks (e.g. signal processing tasks), but do so in a more efficient way. This is called the *flexibility-performance trade-off*.

Fine-grain reconfigurable hardware (FPGA fabric) has the ambition to deliver the same amount of flexibility as an instruction set processor (ISP) while providing a performance level close to that of an ASIC. Obviously, these reconfigurable hardware devices operate in a completely different way and, hence, exhibit radically different properties with respect to an instruction set processor architecture.

When combining ISP tiles with FPGA fabric tiles into a System-on-Chip, the run-time manager could abstract the differences between the tiles and just consider every tile as *a processing element* that can execute a task with a certain performance for a certain cost. However, by neglecting (i.e. abstracting) the specific properties of the fine grain reconfigurable hardware tiles, the performance of the task assignment algorithm is likely to be sub-optimal. This results in less efficient usage of platform resource.

In addition, FPGAs have become large enough to accommodate a large amount of soft IP cores. This enables the creation of a so-called *configuration hierarchy*. Instead of executing a dedicated hardware task, the FPGA fabric hosts a programmable softcore IP block that, in turn, executes the task functionality.

The rest of this chapter is organized as follows. Section 3.1 and Section 3.2 respectively detail the rationale of reconfigurable systems and briefly introduce fine-grain reconfigurable hardware. Section 3.3 provides the problem definition. Section 3.4 till Section 3.7 details a fast and efficient run-time resource assignment heuristic. In addition, we show that by exposing this heuristic to the specific properties of the fine grain reconfigurable hardware tiles, one can significantly improve the assignment performance. Section 3.8 and Section 3.9 introduce a novel run-time task assignment algorithm that exploits a configuration hierarchy. Hierarchical configuration enables easy time-multiplexing of FPGA fabric and it improves the spatial task assignment freedom resulting in a more efficient usage of platform resources. Section 3.10 describes the related work. Finally, Section 3.11 presents the conclusions.

3.1 The Rationale of Reconfigurable Systems

There are a number of reason for using fine-grain reconfigurable hardware instead of an ASIC or for integrating reconfigurable fabric into a System-on-Chip. Most of these reasons relate to the issues that triggered the SoC revolution and to the requirements imposed on MPSoC systems (Section 1.1).

Figure 3.1 details the computational efficiency of reconfigurable hardware with respect to ASICs and ISPs [43]. However, *Field Programmable Gate Arrays* (FPGAs) compete with ASICs and ISPs on several fronts: design cost, unit price, time-to-market, performance, flexibility, etc.

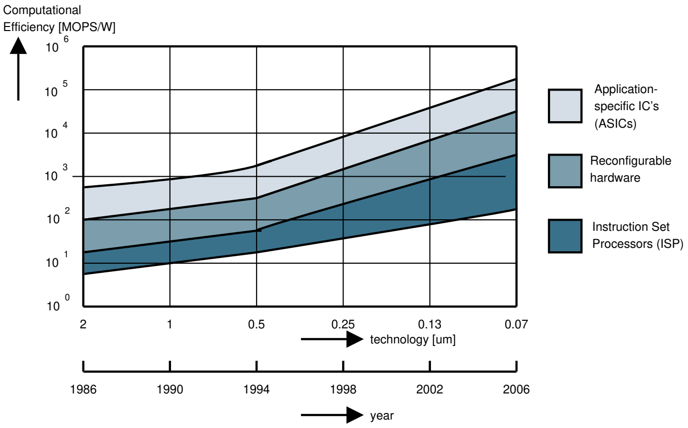


Figure 3.1: *Computational efficiency (MOPS/W) of reconfigurable hardware with respect to ASICs and general purpose ISPs.*

First of all, since FPGAs are an off-the-shelf product, they don't really have an NRE cost associated with them. This means that, for low-volume products, ASICs can no longer compete with FPGAs. Furthermore, Moore's law is favorable for FPGAs [222]. Figure 3.2 shows how the volume trade-off point between reconfigurable hardware (also denoted as Programmable Logic Devices) and ASICs shifts over time. Although both reconfigurable hardware as well as ASICs get cheaper per component (decreasing slopes), the fixed costs for ASICs are rising. Hence, the volume versus costs trade-off point rapidly evolves in favor of the reconfigurable systems. Furthermore, the cost of the basic EDA tools for FPGA design is a lot lower than what is needed for designing an ASIC.

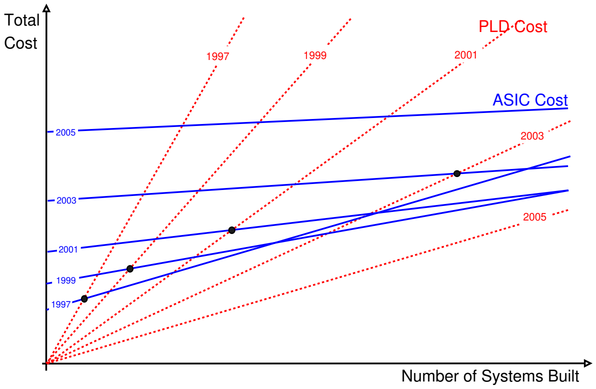


Figure 3.2: *Evolution of the volume versus cost trade-off point between ASICs and programmable logic devices (PLDs, i.e. reconfigurable hardware) [222].*

Secondly, reconfigurable hardware represents the middle ground between ASICs and ISPs. Just like ISPs, FPGAs provide (run-time) programmability and flexibility, while providing far superior performance as the ASIC does. In a world where the rate of innovation is ever-increasing and where standards are evolving fast, programmability is a key asset. This flexibility will also reduce a products time-to-market.

Finally, the applications requirements become more and more demanding. Hence, The performance-flexibility trade-off provides an excellent opportunity for using reconfigurable fabric.

Recently, the industry starts to acknowledge the virtues of using reconfigurable fabric in their SoCs. STMicroelectronics, for example, released the SPEAr™product line. The SPEAr™Lite is composed of an ARM9 RISC PE combined with 400K customizable equivalent ASIC gates with 123 dedicated general purpose I/Os. Xilinx' latest Virtex FPGAs contain one or two hardcore PowerPC microprocessor(s) that allow a designer to create a customized (multiprocessor) platform.

3.2 Fine-Grain Reconfigurable Hardware in a Nutshell

In its broadest sense, a *Field Programmable Gate Array* (FPGA) is an integrated circuit where the functionality can be configured after the chip production process. They can be considered as *general purpose ICs* applicable in a wide range of application domains.

Four different technologies are widely used for commercially available FPGAs. FPGAs based on SRAM cells (i.e. static memory to hold the FPGA configuration), on anti-fuse technology, on EPROM or EEPROM transistors. Depending on the application domain, some technologies are better suited than others. The anti-fuse technology, for example, is faster and cheaper than the SRAM technology, but can only be programmed once. Logic elements based on SRAM technology can be reconfigured relatively fast, but they require a lot of chip area. In contrast to PROM devices, SRAM based technology has unlimited reconfigurability (it is after all a memory).

In this chapter, the considered FPGA fabric tiles are based on SRAM technology. The proof-of-concept MPSoC system, presented in Chapter 6, is based on Xilinx SRAM based FPGA technology. Today, this technology is mature enough to be used as software-like processing elements in a tile-based multiprocessor system.

The basic Xilinx FPGA consists of 3 major programmable components: the *Configurable Logic Blocks* (CLBs), the *Input/Output Blocks* (IOBs) and the *configurable routing resources*. In addition, it contains a clock distribution tree, configuration circuitry, and RAM components. More advanced FPGAs also contain multipliers and processors.

The CLBs are the programmable elements in charge of providing the desired functional behavior. The IOBs take care on the connections between package pins and the chip internals. Finally, the configurable routing resources (also denoted as interconnect resources) are composed of so-called switch matrices that are responsible for interconnecting CLBs and IOBs. In that sense, an FPGA can be considered as a 2D matrix of interconnected CLBs. Every CLB consists of two identical *slices*. In turn, every slice consists of 2 independent 4-input function generators based on *Look-Up Tables* (LUTs). In addition, there is some slice-internal routing functionality.

The design-flow for configuring an FPGA consists of several steps. In the design-entry step, the digital design is created by using a schematic entry tool or just by using a *Hardware Description Language* (HDL). The output of this step produces a *netlist*. This is a description of the connectivity of gates and flip-flops. The second step is the design implementation or mapping step. Here, the netlist is transformed into a bitstream that contains all the FPGA configuration bits for the required functionality. Coming to a bitstream requires (1) a mapping phase, where the functionality is mapped to the FPGA-dependent configurable elements, (2) a place & route phase, where the functionality is assigned to specific configurable resources.

Today, FPGA devices provide the equivalent of several millions of ASIC gates. and also enable partial configuration. Partial configuration allows us to (re)configure part of the available hardware while the remainder of the design is still operational. Thus, more parts of an application, or different applications, can be assigned to the hardware in sharing fashion with parallel processing. This feature is useful for applications that require the loading of different designs into the same area of the device or for applications that need the flexibility to change a portion of the design without having to completely reconfigure or reset the device.

A fine-grain reconfigurable hardware processor has some properties that cannot be found on a regular instruction set processor. The most prominent properties that directly affect run-time management are:

1. **High FPGA task setup time.** Configuring the FPGA fabric, which effectively means setting up a task on a fine grain reconfigurable hardware processor takes at least one order of magnitude more time than on a regular ISP. This is mainly due to the large amount of configuration bits.

2. **Inseparability of task setup and task execution.** In contrast to setting up a task on an instruction set processor (i.e. creating an execution stack), setting up a task in single-context fine-grain reconfigurable hardware always implies that (1) the target FPGA area is available (i.e. no other task occupies the area) and (2) the task immediately starts *executing* after configuration.

3. **Efficient use of reconfigurable area.** As the fine-grain reconfigurable hardware that we consider acts as a memory, it can also get fragmented. Two types of fragmentation are considered. Internal fragmentation can be defined as the *wasted* FPGA fabric when the tile is bigger than the task size. External fragmentation occurs after placing multiple tasks within a single FPGA fabric space in such a way that the remaining free area is not usable for additional tasks.

A more elaborate description and a taxonomy of existing fine-grain reconfigurable hardware platforms with their respective properties was created within the context of the AMDREL project [47].

3.3 Problem Definition

Every platform architecture is described by an Architecture Graph $AG(P, L)$, containing a set of processor tiles $p_i \in P$, interconnected by a set of NoC links $l_{ij} \in L$, where l_{ij} represents the link between tile i and tile j. The load of link l_{ij} is denoted as l_{ij}^{load}. The communication path $path_{ij}$ between tile i and tile j contains a set of H_{ij} links, where H_{ij} denotes the number of hops between tile i and tile j. The type of processor p_j is denoted by τ_j. The set of FPGA fabric tiles is denoted by $R \subset P$. If $p_j \in R$, then the size of its FPGA fabric is denoted by p_j^{size}. The number of tiles adjacent to processor j is denoted as $p_j^{connect}$.

Every application is described by a directed graph, further denoted as Task Graph $TG(T, C)$, where each vertex represents an application task $t_u \in T$ and every edge $c_{uv} \in C$ represents the communication link between task t_u and t_v. The communication load of link c_{uv} describes the amount of bandwidth task t_u requires. This load is denoted as c_{uv}^{load} ($c_{uv}^{load} = 0$ when there is no communication). The load task t_u imposes on tile p_j is denoted by t_{uj}^{load}. It is defined as the ratio of the task execution time and the task deadline. A single task can have support for multiple processing element types: $\tau_j(t_u) = 1$ denotes that task t_u support processor type τ_j. The size of the FPGA task implementation of task t_u (if supported) is denoted by t_u^{size}.

Figure 3.3 illustrates the problem definition of mapping a task graph $TG(T, C)$ onto an architecture graph $AG(P, L)$. The mapping of $TG(T, C)$ onto $AG(P, L)$ is further denoted as $TG(T, C) \rightarrow AG(P, L)$. To evaluate a certain mapping, we need a mapping cost function that considers the mapping cost from both a *communication* and *computation* point of view.

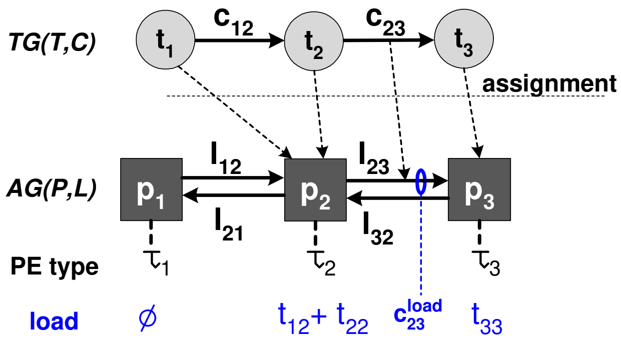


Figure 3.3: *Mapping a Task Graph $TG(T,C)$ onto an Architecture Graph $AG(P,L)$.*

The communication cost has two different angles: time and energy. The time needed to send a message (also denoted a message latency) between two tiles is dependent on a lot of parameters like the network topology (e.g. ring network, mesh network, torus network, etc), the switching technique (e.g. store-and-forward switching, wormhole switching, virtual cut-through switching), the type and the amount of buffers in the routers, the type of arbitration in the routers, the congestion in the network, etc.

In case of congestion-free virtual cut-through switching, Equation 3.1 describes the latency $Lt^{uncongested}$ for sending a packet M, consisting of N *flits*[1] between tile i and tile j [21]. In this equation, T_{flit} denotes the time needed for a single flit to pass a single router. For the *Gecko*[2] best-effort NoC implementation (detailed in Chapter 6), one has to consider the possibility of congestion. Hence, the latency $Lt^{uncongested}$ is augmented with a *bufferingdelay* component, coupled to a congestion probability P, i.e. the probability of temporarily being buffered at router n (Equation 3.2). This probability obviously relates to the amount of traffic that passes through this router.

$$Lt^{uncongested}(M_{ij}) = T_{flit} \times H_{ij} + (N-1) \times T_{flit} \tag{3.1}$$

$$Lt^{congested}(M_{ij}) = Lt^{uncongested}(M_{ij}) + \sum_{n=1}^{H_{ij}} P_n \times \text{bufferingdelay} \tag{3.2}$$

Similarly, the energy required to send a message from tile i to tile j depends on the switching technique, the length of the physical wires, the buffer allocation, the NoC topology, etc. [19]. In case of wormhole switching, the average energy E_{ij} required to send *one bit* of information from p_i to p_j is a function of the energy consumed by a router E_{router}, the energy consumed by a link E_{link} and the number of hops the information needs to travel H_{ij} (Equation 3.3) [100,236].

$$E_{ij} = H_{ij} \times E_{router} + (H_{ij} - 1) \times E_{link} \tag{3.3}$$

[1] A packet is composed of a number of *flow control units* (flits). These are the packet units on which flow control operates.

Hence, both the time component, as well as the energy component are strongly influenced by the number of hops packets need to take when traveling from sender task to receiver task and by the size of the packets. Therefore, the mapping algorithm has to take these variables into account. This can be done by minimizing the hop-bandwidth product (further denoted as ϕ) of the assignment $TG(T, C) \to AG(P, L)$, i.e. the product of the application communication load and the hop-distance (Equation 3.4).

$$\phi_{TG(T,C)\to AG(P,L)} = \sum_{\forall t_u \to p_i, \forall t_v \to p_j} H_{ij} \times c_{uv} \tag{3.4}$$

Furthermore, as equation 3.2 describes, in a best-effort NoC, the packet latency heavily depends on the load of the communication path (i.e. congestion probability). In order to ensure a common latency throughout the platform, it makes sense to spread the load over the available communication resources. In a guaranteed-throughput NoC, the path finding algorithm performs better in terms of resource usage efficiency (i.e. path length) in case of a balanced network load [126]. The *communication load variance*, further denoted as σ_L^2, can be used as a measure for link load balancing.

From a computation point of view, one should also distribute the load over the different processing elements. This will avoid so-called *hot spots* which will decrease the leakage current (leakage current rises exponentially with die temperature). The *processor load variance*, further denoted as σ_P^2, can be used as a measure for processor load distribution.

The performance of the task assignment algorithm is determined by (1) its speed, i.e. how fast a solution is found, (2) its assignment success rate, i.e. its ability to find a solution when one exists and (3) the assignment quality, i.e. minimal ϕ, minimal σ_L^2 and minimal σ_P^2. Note that the σ_L^2 and minimal σ_P^2 are platform related, i.e. take into account all previously assigned applications, while ϕ is only related to the assignment of the current task graph.

3.4 Generic MPSoC Task Mapping Heuristic

At design-time, it is often unknown which applications will run simultaneously. Only at run-time the resource usage of the platform as well as the application user quality requirements are known. Hence, one needs a run-time application resource assignment algorithm. A fast, lightweight heuristic that comes up with a reasonably good solution is preferred over an algorithm that comes up with an optimal solution requiring a lot of computation time. Hence, a so-called *greedy heuristic* is used. A heuristic is denoted as greedy when it makes locally optimal solutions for every decision variable, hoping that the global solution will also be optimal. The power of a greedy heuristic lies in its simplicity [142].

In order to assign a new application $TG(P, C)$ to a platform $AG(P, L)$ the heuristic requires a specification of the application, the quality requirements with respect to the application and, finally, the current resource usage of the platform as input. The three steps of the complete heuristic (Algorithm 3) to come to a complete resource assignment of an application are as follows.

Symbol		Definition
Architecture Graph	P	Set of processors
	R	Set of FPGA processors, subset of P
	L	Set of communication links
	p_i	Processor of tile i
	p_i^{max}	Maximum processor load of tile i, equal to 100% load
	l_{ij}	Communication link between tile i and adjacent tile j
	l_{ij}^{max}	Maximum communication load of link l_{ij}, equal to 100% tile
	$path_{ij}$	Communication path between tile i and tile j
	H_{ij}	Number of hops between tile i and tile j
	τ_j	Processor type of processor p_j
	p_j^{size}	Size of FPGA tile j
	$p_j^{connect}$	Number of tiles adjacent to processor j
Task Graph	T	Set of tasks, i.e. vertexes in the task graph
	C	Set of communication links, i.e. edges in the task graph
	t_u	Task graph task
	t_{uj}^{load}	Processor load of task t_u
	t_u^{size}	Size of t_u FPGA implementation
	$\tau_j(t_u)$	If equal to 1, denotes that t_u supports processor type j
	c_{uv}	Communication edge between t_u and t_v
	c_{uv}^{load}	Communication load imposed by edge c_{uv}
Mapping	$AG(P, L)$	Architecture Graph with processors set P and link set L
	$TG(T, C)$	Task Graph with task set T and edge set C
	ϕ	Hop-bandwidth product
	σ_P^2	Processor load variance
	σ_L^2	Communication link load variance

Table 3.1: *Symbols for the architecture graph, the task graph and the mapping process.*

The first step (Algorithm 1) prioritizes the tasks as follows. For every task $t_u \in T$, we first determine the load variance with respect to the different supported tiles (line 3, first factor). Intuitively it is easy to understand that tasks with a high processing load variance are very sensitive to which processing element they are assigned to. We also determine the tasks communication importance with respect to the total inter-task communication of the application (line 3, second factor). The tasks mapping priority $Prio(t_u)$ is the product of the load variance and the communication importance. In addition, tasks that can only be assigned to one specific tile should be mapped before all other tasks. This way, the heuristic avoids a mapping failure, that would occur if this specific tile would be occupied by previously assigned task of the same task graph TG. Finally, tasks are sorted by descending priority.

The second step (Algorithm 2) prioritizes the tiles for the most important unmapped task t_u. To this end, the cost $Cost(p_i)$ of a tile $p_i \in P$ with respect to the most important unmapped task t_u is determined based on the product of its current processing load (p_i^{load}), the already used communication resources to its neighboring tiles and the hop-bandwidth product (ϕ) to its already assigned communication peers (t_v). The path $path_{uv}$ between task t_u and t_v is either predetermined (e.g. XY routing) or

Algorithm 1: Determining task assignment priority.

```
Input: $TG(T,C)$, $P$
Output: Prioritized task assignment list
PRIORITIZETASKS($TG(T,C), P$)
foreach $t_u \in T$
  if (nr supported tiles $j > 1$)
    $Prio(t_u) = \sum_j \frac{(t_{uj}^{load} - \langle t_{uj}^{load} \rangle)^2}{nr\ supported\ tiles\ j} \times \frac{\sum_{\forall u} c_{uv}^{load}}{\sum_{\forall u,v} c_{uv}^{load}}$
  else
    $Prio(t_i) = \frac{\sum_{\forall u} c_{uv}^{load}}{\sum_{\forall u,v} c_{uv}^{load}} + Prio_{max}$
```

should be interpreted as a function that determines the path. Notice that the platform assignment quality measures, σ_L^2 and σ_P^2, do not appear in the algorithm. Indeed, recalculating the σ_P^2 and σ_L^2 for every possible assignment is time consuming. In order to ensure an equal spread of processing load and communication load, the tile cost factor includes the processor load and the load on its attached communication links. Taking the hop-bandwidth into account ensures that heavily communicating tasks are mapped close together. Tiles that lack the required resources have their cost set to infinity, indicating that the tile is not fit to accommodate t_u. Tiles are sorted by ascending cost.

Algorithm 2: Determining tile priorities for a task t_u.

```
Input: $t_u$, $TG(T,C)$, $AG(P,L)$
Output: Number of suitable tiles and their respective Cost for $t_u$.
PRIORITIZETILES($t_u, TG(T,C), AG(P,L)$)
foreach tile $p_i \in P$ supported by $t_u$
  if (($p_i^{load} + t_{ui}^{load} > p_i^{max}$) or ($t_u^{size} > p_i^{size}$))
    $p_i$ not suited for $t_u$
  else
    $\phi = 0$
    foreach $t_v \in T$ assigned to $p_j$
      if ($c_{uv} \neq 0$)
        foreach $l_{xy} \in path_{ij}$
          if ($l_{xy}^{load} + c_{uv}^{load} > l_{xy}^{max}$)
            $p_i$ not suited for $t_u$
      if ($c_{vu} \neq 0$)
        foreach $l_{yx} \in path_{ji}$
          if ($l_{yx}^{load} + c_{vu}^{load} > l_{yx}^{max}$)
            $p_i$ not suited for $t_u$
    $\phi = \phi + (H_{ij} \times c_{uv}^{load}) + (H_{ji} \times c_{vu}^{load})$
  if ($p_i$ not suited for $t_u$)
    $Cost(p_i) = \infty$
  else
    $Cost(p_i) = p_i^{load} \times \phi \times \frac{\sum_{\forall q}(l_{iq}^{load} + l_{qi}^{load})}{p_i^{connect}}$
$N(t_u)$ = amount of tiles with Cost $\neq \infty$
return $N(t_u)$
```

In the third step, the most important unmapped task t_u is assigned to the tile with the lowest cost. Steps two and three are repeated until all tasks are mapped (Algorithm 3).

Algorithm 3: *Mapping a task graph onto an architecture graph using the Generic Heuristic.*

```
Input: $TG(T,C)$, $AG(P,L), bt$
Output: Assignment $TG(T,C) \rightarrow AG(P,L)$
GenericHeuristic($TG(T,C), AG(P,L), bt$)
(1)  $PrioritizeTasks(TG(T,C),P)$
(2)  foreach unmapped $t_u$ with highest $Prio(t_u)$
(3)    $N(t_u) = PrioritizeTiles(t_u, TG(T,C), AG(P,L))$
(4)    if ($N(t_u) > 0$)
(5)      Assign $t_u$ to $p_i$ with lowest $Cost(p_i)$
(6)    else
(7)      if (($bt > 0$) and ($t_u \neq$ first task))
(8)        repeat
(9)          Undo allocation of previous task $t_v$
(10)         $bt = bt - 1$
(11)       until ($N(t_v) > 1$) or ($bt = 0$) or (first assignment)
(12)     if ((($bt = 0$) or (first assignment)) and ($N(t_v) \leq 1$))
(13)       Exit. //No solution found.
(14)     else
(15)       Assign $t_v$ to $p_j$ with second lowest $Cost(p_j)$
(16)       $N(t_v) = 1$
```

Occasionally the greedy heuristic is unable to find a suitable assignment for a certain task (Algorithm 3, line 4). This usually occurs when mapping a resource-hungry application on an already heavily loaded platform. *Backtracking* is the classic solution for this issue: it changes one or more previous task assignments (of the same task graph) in order to solve the mapping problem of the current task t_u.

The backtracking algorithm (Algorithm 3, lines 7-16) starts by finding a previously assigned task t_v with multiple assignment options (i.e. more than one suitable tile for assignment). Consequently, all resource allocations up until this task are undone. Then, task t_v is assigned to the *second best* tile. From then on, the heuristic starts all over assigning the most important unmapped task. If the assignment would fail again at the same task, backtracking will first consider other previous tasks than t_v for re-assignment. That is why $N(t_v)$ is set to 1 after its re-assignment (line 16). Backtracking stops when either the number of allowed backtracking steps is exhausted ($bt = 0$) or when backtracking reaches the first task assignment of the application (i.e. no previous tasks).

When that happens (line 13), the algorithm can (a) use run-time task migration (Chapter 4) to relocate a task of another application in order to free some resources or (b) restart the heuristic with reduced quality requirements (Chapter 7). Furthermore, the assignment success rate and the quality of the assignment solution can be improved by using hierarchical configuration (Section 3.8).

3.5 Reconfigurable Hardware Correction Factors

Incorporating FPGA fabric tiles requires some additions to the generic heuristic in order to take the following FPGA properties into account.

First, the *internal fragmentation of reconfigurable area* is considered, i.e. the FPGA fabric that is wasted because the tile size is larger than the task size. In case both the first and second priority tile, p_i and p_j, are both FPGA tiles, the heuristic will re-evaluate their priority ($Cost(p_i)$ and $Cost(p_j)$) by using a fragmentation ratio (Equation 3.5) in order to minimize the reconfigurable area fragmentation when placing a task t_u. Intuitively it is easy to understand that if placing the task on the best tile causes 80% area fragmentation while the second best tile only causes 5% area fragmentation, it might be better to place the task on the latter.

$$Cost(p_i)_{new} = Cost(p_i)_{old} \times \frac{(p_i^{size} - t_u^{size})}{p_i^{size}} \tag{3.5}$$

Secondly, the *binary state* (i.e. either 0% or 100% perceived load) and the *computational performance* of reconfigurable tiles are considered. Due to the attempt at load-sharing of the heuristic, unused FPGA tiles are often selected as best mapping candidates. In view of additional new applications, it would not be wise to sacrifice a reconfigurable hardware tile when an ISP could do a similar job. Therefore, if the first priority tile p_i for a certain task is a FPGA tile, while the second priority tile p_j is an ISP, the heuristic will use a load ratio (Equation 3.6) to re-evaluate their priority to avoid wasting FPGA fabric computing power. This way, the FPGA fabric tile can be saved for later use.

$$Cost(p_i)_{new} = Cost(p_i)_{old} \times \frac{p_i^{max} - t_{ui}^{load}}{t_{uj}^{load}} \tag{3.6}$$

The tile priority correction factors are applied after step two of the generic heuristic (Algorithm 3, line 4), i.e. after sorting all suitable tiles according to their priority.

The motivation for using the correction factors is to avoid wasting reconfigurable hardware fabric, either by placing the task on a smaller tile with similar performance or by placing the task on an ISP tile. This way, the reconfigurable hardware fabric will be more available for large tasks or for more compute intensive tasks. This not only improves the task assignment success rate of the current application, it is particularly important for the assignment success rate of future incoming applications.

3.6 Experimental Setup

As target platforms we consider heterogeneous multiprocessor systems, each containing different processing element types. The different tiles are interconnected by a *Network-on-Chip* (NoC). In this case[2], we assume a 3 by 3 mesh network with deterministic XY routing (Figure 3.4). Nevertheless, the proposed solutions are applicable

[2] This size and topology were selected to match the real-life *Gecko*^{2} demonstrator, detailed in Chapter 6.

on a wide set of platform architectures with different size, topology and routing scheme. We consider four processing element types: GPP, DSP, FPGA and a flexible accelerator, denoted as *Accel*.

The differences between these processor types will be reflected in (1) their task support, i.e. what percentage of tasks have support for a particular processing element, and (2) their load, meaning that a task will, on average, impose a higher load on a general purpose processor than on e.g. an accelerator. We also vary the number of FPGA fabric tiles and their size (i.e. small and large) in order to determine its influence on the performance of the mapping algorithm and, more specifically, the use of the FPGA correction factors. While a large tile can accommodate any reconfigurable hardware task, the small tile can only accommodate a subset of tasks (i.e. the ones that fit the tile).

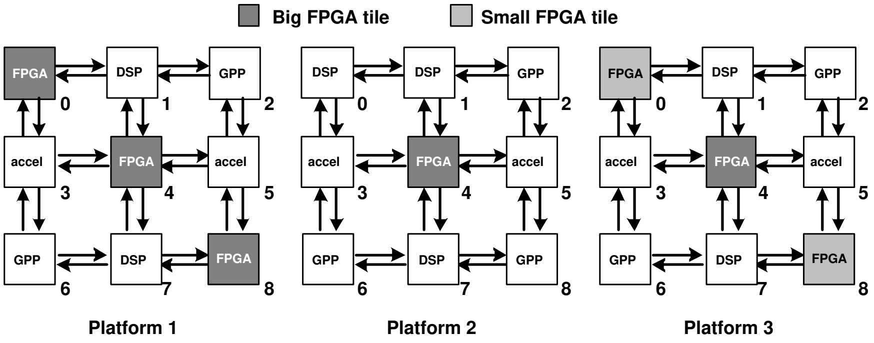


Figure 3.4: *Considered MPSoC platform architectures. Every architecture contains 4 different PE types: general purpose processors (GPP), Specialized processors (DSPs and Accelerators) and reconfigurable hardware tiles of potentially different sizes.*

To evaluate the performance of the resource assignment heuristic, a large set of task graphs is required. We used a software tool called *Task Graphs For Free* (TGFF) [60] to create 1000 random task graphs, each containing between 3 and 10 tasks and where every task contains up to three communication links. Every task potentially has multiple implementations in order to support up to four processing element types.

Depending on the quality requirements (high or low), every application task implementation as well as every application communication link is assigned a certain random load. Table 3.2 details the average load a task imposes on a processing element type with respect to the quality requirements. In addition, Table 3.2 also details the task support rate, indicating that e.g. 6 out of 10 tasks have support for executing on the GPP. A random load value is also assigned to every application communication link: on average 25% (spread 15%) and 50% (spread 25%) load for respectively low and high quality requirements.

The load figure averages are chosen in such a way that, for a given user requirement, multiple processing element types are eligible for hosting a certain task. Having load figure averages far apart, simplifies the solution search space. Nevertheless, the proposed algorithms remain valid.

Table 3.2 details the load a task imposes on the different processing element types. The actual values for every task are determined randomly by TGFF.

Table 3.2: *Application load and support factor for different processor types.*

	Load Low		Load High		
PE type	**average**	**spread**	**average**	**spread**	**support factor**
GPP	25%	15%	50%	15%	60%
DSP	20%	10%	45%	10%	40%
FPGA	18%	10%	30%	10%	20%
Accel	15%	10%	30%	10%	20%

The platform resources are pre-loaded to indicate the presence of previously assigned applications. The *low* and the *high* platform load parameter indicate that no platform resource (both computation and communication) is used for more than respectively 25% and 50%. A random function determines the actual resource usage for every resource. Due to the binary load state of FPGA tiles (i.e. either 0% or 100% load), they are always pre-set as free. This way, we can clearly determine the effect of changing the amount of FPGA tiles or their size.

The main goal of TGFF is to enable other researchers to reproduce the experimental input data simply by sharing the input parameter settings. Appendix B details the TGFF input settings for the following experiments.

In order to verify the performance of the heuristic algorithm, we also created a *full search* (fs) algorithm (Algorithm 4). This algorithm exhaustively searches all possible solutions for assigning an application task graph to a multiprocessor platform with a certain load state.

By using the full search algorithm one can verify if an assignment is at all possible in case the heuristic does not find a suitable assignment. Furthermore, it allows us to assess the quality of the assignment solution provided by the heuristic. The full search algorithm determines the quality Q of a total task graph assignment based on the product of the processor load variance σ_P^2, the communication load variance σ_L^2 and the hop-bandwidth product ϕ. While traversing the assignment solution space when mapping an application, the full search algorithm retains the extrema (further denoted *fs-min* and *fs-max*) found for each of these parameters. These values put the selected solutions into perspective.

3.7 Heuristic Performance Evaluation

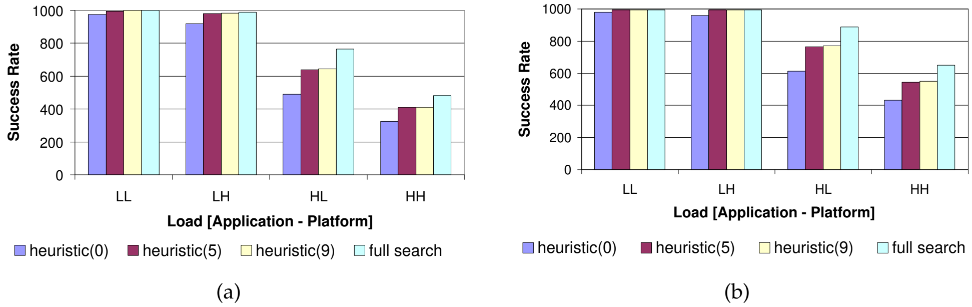
First, this section evaluates the performance of the generic heuristic with respect to the assignment success rate and the quality of the solution. Here, the full search algorithm serves as a benchmark. Secondly, we evaluate the impact of the correction factors detailed in Section 3.5.

Algorithm 4: *Task graph mapping by searching the full solution space.*

```
Input: $TG(T, C)$, $AG(P, L)$
Output: Optimal Resource assignment: $TG(T, C) \rightarrow AG(P, L)$
FullSearch($AG(P, L), TG(T, C)$)
$\sigma^2_{P,min} = \sigma^2_{L,min} = \phi_{min} = \infty$
$\sigma^2_{P,max} = \sigma^2_{L,max} = \phi_{max} = 0$
while $((TG \rightarrow AG) = NextMappingSolution(TG(T, C), AG(P, L)))$
    $\{\sigma^2_P, \sigma^2_L, \phi\} = EvaluateMapping()$
    $Q = \sigma^2_P \times \sigma^2_L \times \phi$
    if $Q < Q_{best}$
        $Q_{best} = Q$
        $RememberBestMapping(TG \rightarrow AG)$
    foreach parameter $p \in \{\sigma^2_P, \sigma^2_L, \phi\}$
        if $p < p_{fs-min}$
            $p_{fs-min} = p$
        if $p > p_{fs-max}$
            $p_{fs-max} = p$
$AssignBestMapping(TG \rightarrow AG)$
```

3.7.1 Generic Heuristic

Figure 3.5(a) and Figure 3.5(b) detail the assignment success rate (y axis) of the heuristic and the full search algorithm for platform 1 and platform 2 respectively (Figure 3.4) as a function of the application load (x axis, first letter) and the platform load (x-axis, second letter). The amount of allowed backtracking steps is indicated between brackets.



(a)

(b)

Figure 3.5: *Generic heuristic assignment success rate for various application and platform load [Application - PE Link] on (a) platform 1 and (b) platform 2.*

The success rate results show that the heuristic nearly always finds a solution for low application load irrespective of the platform load. In case of high application load the heuristic does not perform as good as the full search algorithm. In the worst case (i.e. platform 2 with a high application load and high platform load), it scores about 16% lower than the full search algorithm. Backtracking clearly improves the heuristic performance. Having more than nine backtracking steps does not make sense for three reasons: the system only has nine tiles, task graphs have maximally ten tasks, and the heuristic currently only considers the two best assignment options.

For high application load, the differences in success rate between platform 1 and platform 2 can be explained by the difference in amount of reconfigurable hardware tiles. Platform 2 can capitalize on having more ISPs that can accommodate multiple tasks and have a higher support factor. This results in a higher success rate. In case of a high platform PE load (i.e. HH), the success rate difference between the heuristic and the full search algorithm is 3% smaller for platform 1 than for platform 2. This effect is caused by platform 1 exploiting its multiple free reconfigurable hardware tiles.

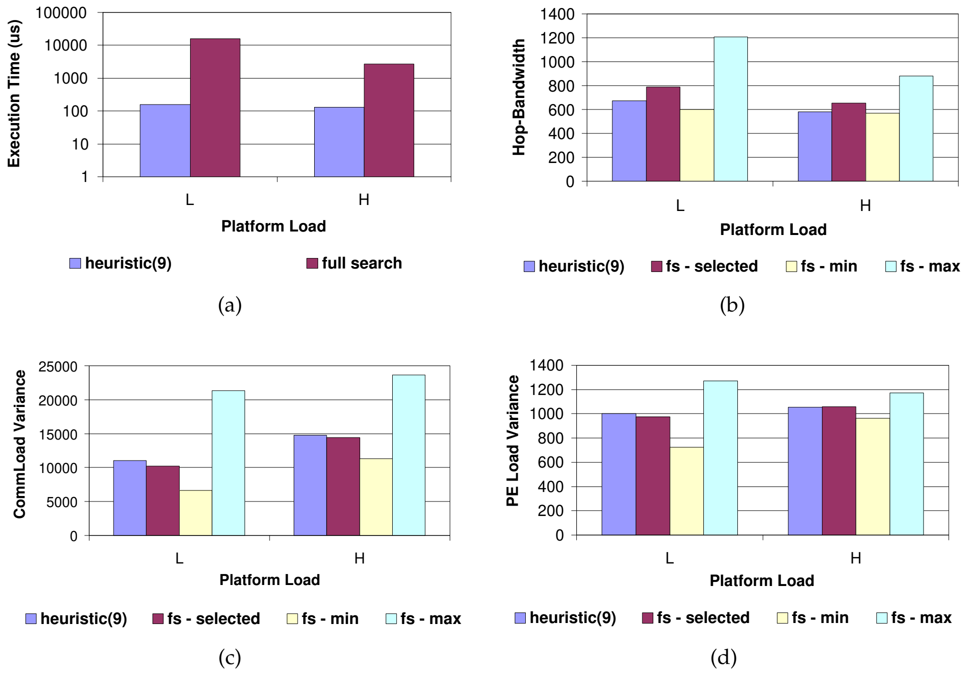
Figure 3.6(a) indicates the speed of both the heuristic and the full search algorithm measured on the SimIt StrongARM ISS with a clock speed of 206 MHz. The speed of the heuristic lies well within acceptable run-time boundaries. As a reference, the time required to start a new application (i.e. creating a new application process) in the Linux operating system is in the order of magnitude of 1 ms to 10 ms [138] depending on the hardware platform.

For platform 1 and platform 2, the heuristic requires on average (depending on the platform load) between 141 μs and 169 μs (stdev about 100 μs) to reach an assignment using at most 9 backtracking steps. In contrast, the full search algorithm requires on average between 2.7 ms and 15.9 ms with peaks reaching up to 4 seconds when faced with a low platform load and a small task graph where each task supports multiple PEs. These figures clearly indicate the speed benefits of using a greedy heuristic. The main reason for the difference between the high and the low platform load is that for high platform load, the number of assignment options is more limited.

Besides success rate and calculation speed, the quality of the results produced by the algorithm is equally important. Figure 3.6(b), Figure 3.6(c) and Figure 3.6(d) detail, respectively, the hop-bandwidth product, the communication load variance and the processor load variance (always the average over all experiments) for a high application load. We notice that the hop-bandwidth product of the heuristic solution is lower, while both the communication load variance and the processor load variance are higher.

We do not only compare the result of the heuristic with the chosen result of the full search algorithm (denoted *fs-selected*), we also provide the maximum (denoted *fs-max*) and minimum value (denoted *fs-min*) of every individual performance variable. These values are obtained during the exploration of the full solution space and put the quality of the results into perspective (Section 3.6). This comparison shows that, although the heuristic does not directly evaluate σ_P^2 and σ_L^2, it still provides good solutions (i.e. close to fs-selected and belonging in the lower part of the solution spectrum). As the hop-bandwidth product ϕ is a factor in the heuristic cost function, the ϕ of the provided mapping solutions comes quite close to the minimal value.

Overall, we can conclude that the quality of the results produced by the heuristic are quite close to those of the full search algorithm. On average, the solutions selected by the heuristic yield a lower hop-bandwidth product, but a slightly higher communication load variance σ_L^2.



(a)

(b)

(c)

(d)

Figure 3.6: *Experimental results for Platform 1 and high application load. (a) Average execution time of the heuristic and the full search algorithm. (b) Heuristic hop-bandwidth product ϕ. (c) Heuristic communication load variance σ_L^2. (d) Heuristic processor load variance σ_P^2.*

3.7.2 Heuristic with Correction Factors

Introducing the correction factors should have an effect on both intra-application and inter-application mapping effect. This means that, when starting two new applications, these factors improve both the mapping performance of both the first task graph (i.e. intra-application effect) and of the second task graph (i.e. inter-application effect).

In order to evaluate the performance of the heuristic with FPGA correction factors, we randomly selected 20000 task graph pairs (TG_i, TG_j) using the 1000 task graphs generated by TGFF. The rationale here is to evaluate the effect on the assignment success rate of both the *first task graph* (1/2) as well as the *second task graph* (2/2) (after the first task graph was successfully assigned).

In all of the experiments, platform resources are pre-loaded with a low load. The application load of the first task graph is always low, while the load of the second task graph varies. This setup allows to compare results with the evaluation of the generic heuristic. The improvement values do not significantly differ with respect to the initial platform load (low or high), so the presented figures show the average improvement with respect to the platform load.

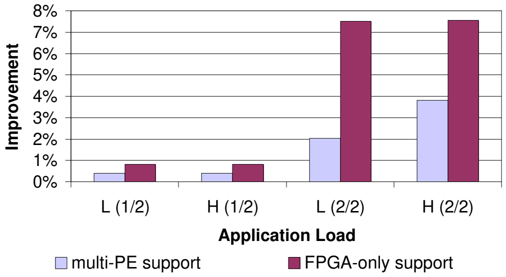


Figure 3.7: *Correction factor assignment success rate improvement on platform 3. The application load represents the load of the second application. Tasks support up to four PEs (multi-PE) or tasks with FPGA support do not support other PEs (FPGA-only).*

We conduct two experiments using platform 3, i.e. with different FPGA tile sizes (Figure 3.7). In this case, the small tile can only accommodate 35% of the tasks. In the first experiment, we assume that tasks can support up to four PEs according to the distribution given by Table 3.2. In the second experiment, we make the assumption that a task that supports an FPGA tile does not have support for any other processing element. This assumption can be justified by the fact that the design flow for a hardware implementation is significantly different than for a software implementation.

We conclude that the FPGA correction factors are important for improving the assignment success rate of new incoming applications i.e. the second task graph (2/2). The effect on the first task graph is negligible. The impact of using FPGA correction factors is even larger when tasks can only execute on an FPGA tile.

3.8 Exploiting a Configuration Hierarchy

This section deals with the ability of fine-grain reconfigurable hardware to create a configuration hierarchy and the way to manage it at run-time. First, Section 3.8.1 explains the concept and rationale of hierarchical configuration. Secondly, Section 3.8.2 details an algorithm to prioritize and instantiate softcores. Finally, we propose two ways of adding hierarchical configuration support into the generic heuristic. Section 3.8.3 details a hierarchical configuration correction factor. while Section 3.8.4 adds hierarchical configuration components into the main flow of the heuristic.

3.8.1 Rationale of Hierarchical Configuration

Historically, FPGA fabric allowed to separate the design from the actual physical hardware. In recent years, FPGAs have become large and fast enough to accommodate programmable IP cores (i.e. microprocessor). Hence, the boundary between software, executing on the IP core, and *soft hardware*, instantiated in the FPGA is fading. Essentially, this redefines the boundaries between hardware and software.

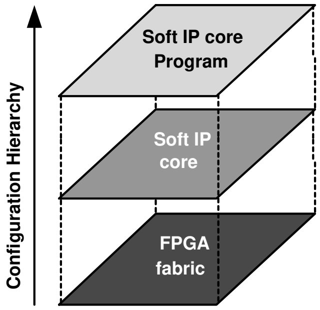


Figure 3.8: *Configuration hierarchy concept: the FPGA fabric runs a programmable soft IP core that, in turn, executes some user program.*

In order to understand the ability of fine-grain reconfigurable hardware (i.e. FPGA fabric) to create a configuration hierarchy, consider an FPGA that runs a programmable soft IP core (Figure 3.8). In turn this soft IP core executes some user program. While the soft IP core acts as *program code* for the FPGA fabric, it also acts as hardware for the user program. Consequently, the user program defines the actions for the programmable IP core, while from an FPGA point of view it is merely data being processed by the soft IP core circuit. This setup forms a *configuration hierarchy*.

In recent years, most FPGA vendors provide soft IP components ranging from data encryption to signal processing and communication. Furthermore, every FPGA vendor provides its own flavor of a soft general purpose microprocessor. Xilinx, for example, provides the PicoBlaze and the MicroBlaze, while Altera provides the NIOS embedded processor. Actel, on the other hand, promotes an ARM7TDMI core. Lattice Semiconductor introduced an 8-bit soft microcontroller for its family of FPGAs. ARM promotes its *ARM Cortex-M1* as the first ARM processor designed specifically for implementation in FPGAs. The Cortex-M1 processor targets major FPGA devices from Xilinx, Actel and Altera. This, again, highlights the benefits of using a soft processing element as all traditional ARM development tools can be used in this context. ARC International provides user-customizable, high-performance 32-bit processor IP cores with DSP functionality and their associated software development tools. Finally, a whole collection of freely available, open source IP cores can be found at OpenCores (www.opencores.org), which represents a community of people interested in developing digital open source hardware. The OpenCores collection includes microcontrollers, DSPs, arithmetic, and cryptography soft IP cores.

Using a soft IP core (also denoted as a *software decelerator* [108]) often results in a speed/performance penalty with respect to instantiating a hardware circuit with the same functionality. However, there are both design-time and run-time benefits associated to using a soft IP core.

From a design-time point of view, there is a trade-off between performance and other costs such as chip area needs, ease and speed of implementation [108, 158]. Meaning that creating a software implementation of a task is far easier than creating a dedicated hardware implementation.

From a run-time point of view, using a soft IP core can result in more efficient usage of the platform FPGA resources. First, it enables *time-multiplexing* of the FPGA fabric, i.e. having multiple tasks using the FPGA resources in a concurrent way. Secondly, it greatly improves *spatial task assignment freedom*, meaning that tasks can be placed more freely and communication resource bottlenecks can be circumvented.

Chapter 2 is concerned with adapting the run-time manager to the application in order to improve performance. By using an FPGA fabric in combination with soft-core architectural components (processing elements and on-chip interconnect) one can imagine also *adapting* the platform hardware to the needs of the executing applications. Already today, the Xilinx Virtex II Pro FPGA 2VP50 can accommodate up to 20 MicroBlaze processing elements. This means that flexible MPSoC platforms adapted to the specific application needs are well within reach.

Hence, the ability to use a configuration hierarchy creates significant design-time and run-time opportunities. However, these opportunities require a run-time manager capable of controlling such a hierarchy.

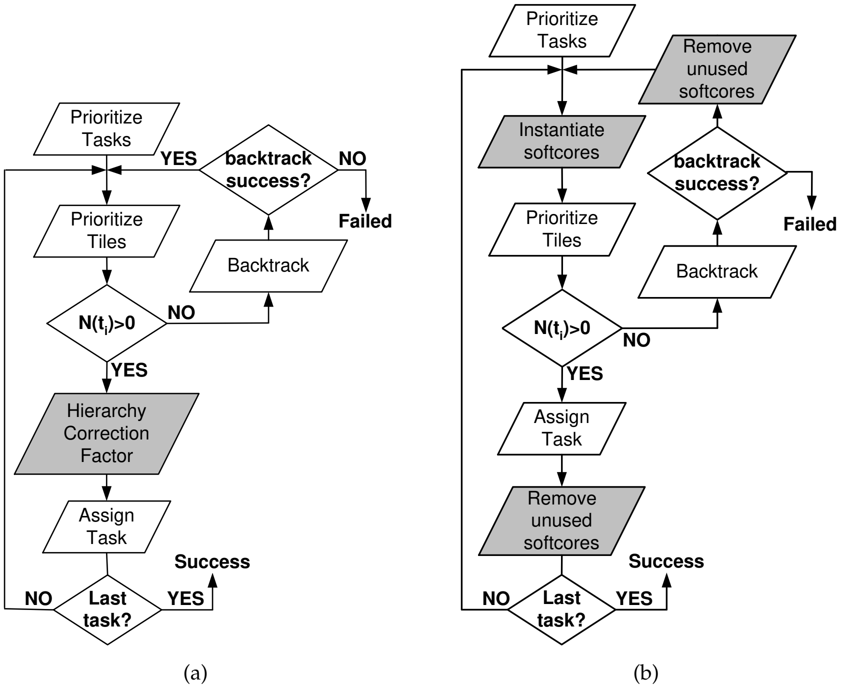


Figure 3.9: Two ways to add hierarchical configuration (gray) into the generic heuristic (white): (a) as a correction factor and (b) into the main flow.

3.8.2 Prioritizing Softcores

Both approaches to adding hierarchical configuration support into the generic heuristic (Figure 3.9) require a way to prioritize and instantiate the softcores onto FPGA fabric tiles. Algorithm 5 details in two steps how to prioritize the softcores $s_j \in S$ with respect to task t_u and how to instantiate them on the available FPGA fabric tiles $p_i \in R$.

Symbol		Definition
Hierarchy	S	Set of softcore processing elements
	s_k	Softcore processing element, $s_k \in S$
	p'_i	Instantiation of softcore s_k onto FPGA tile p_i
	P'	Modified set of processors P, where (some) FPGA tiles are hosting softcores
	$AG'(P', L)$	Architecture Graph AG with softcores instantiated.

Table 3.3: *Additional symbols used for hierarchical configuration.*

In the first step (lines 1-9), the softcores are sorted for a specific task t_i. The cost of every softcore is determined by its future reusability. Since physically instantiating a softcore (just like for a hardware task) is very time consuming, reusing an existing softcore is beneficial. Hence, a softcore with high support factor is preferred.

If the softcore supports multitasking, one also needs to consider the re-usability of the softcore when combining task t_i with a potential future task. This effectively means considering the load that the task t_i already imposes (i.e. how much can be re-used by a future task). Obviously, all softcores that cannot deliver the required performance with respect to t_i are neglected. Finally, we end up with a sorted list of softcores.

In the second step (lines 10-15), we logically[3] instantiate a softcore on every available reconfigurable hardware tile if the task has no FPGA tile support. In case the task does have FPGA tile support, a softcore is only instantiated on the tiles that are too small to fit the FPGA task (i.e. the tiles that would otherwise be unusable). The softcores are chosen based on the cost priority list. Before instantiating the softcore, we need to make sure that (1) the softcore fits on the tile and (2) the softcore will provide the required performance for the task. This evaluation is especially needed when softcores have a different performance depending on the host FPGA tile (not considered in this algorithm).

3.8.3 Hierarchical Correction Factor

The simplest way to use hierarchical configuration to improve the performance of the generic heuristic algorithm is to introduce a *hierarchical correction factor* (Figure 3.9(a)). Indeed, by using a softcore on an FPGA fabric tile, one can (potentially) use this tile for executing multiple tasks. This hierarchical correction factor assumes softcores capable of multi-tasking.

[3] We do not physically instantiate (i.e. configure) the softcores on the FPGA fabric tile in this phase.

***Algorithm 5:** Assigning SoftCore IPs to FPGA fabric tiles.*

```
Input: $t_u, P, S$
Output: Softcore Assignment for $t_u$: $P' = (S \rightarrow R) \bigcup (P \setminus R)$
InstantiateSoftCores($t_u, P, S$)
foreach $s_j \in S$ supported by $t_u$
  if $t_{uj}^{load} > s_j^{maxload}$
    $Cost(s_j) = \infty$
    continue
  if $s_j$ supports multi-tasking
    $Cost(s_j) = \frac{t_{uj}^{load} \times (100 - SupportFactor(s_j))}{s_j^{maxload}}$
  else
    $Cost(s_j) = (100 - SupportFactor(s_j))$
Sort by ascending $Cost(s_j)$, $\forall$ supported $s_j \in S$
foreach unused $p_i \in R$
  if $(\tau_j(t_u) = 1)$ and ($t_u^{size} \leq p_i^{size}$)
    continue
  else
    Find $s_k$ with lowest $Cost(s_k) \neq \infty$ such that $s_k^{size} \leq p_i^{size}$
    Instantiate $s_k$ onto $p_i$
```

Consider the following example. A task t_u with FPGA support can only be assigned to a certain FPGA tile $p_i \in R$, due to communication load issues. However, the task uses only little computation power due to the low application requirements. In this case, the classic correction factors (Section 3.5) are of no use when it comes to making the best use of the scarce FPGA tiles. However, if task t_u supports one or more softcores, one could instantiate a softcore $s_k \in S$ onto $p_i \in R$ and, consequently, assign t_u to this instantiated softcore p_i'. In case of a lightweight task, the remaining compute power for that tile can then be reused by another task.

From an implementation point of view, adding a hierarchical configuration correction factor means replacing line 5 of the generic heuristic (Algorithm 3) with the following code snippet. This code states that if the tile with the lowest cost for task t_u is a FPGA fabric tile and if the load this task imposes (i.e. the active use of the FPGA tile) is below a specified load threshold, then we resort to using a softcore. This means we instantiate the most suitable softcore for task t_u that fits onto FPGA tile p_i. Consequently, we assign task t_u to that newly created processing element p_i'.

```
Consider for $t_u$ that $p_i \in P$ has lowest Cost($p_i$)
if (($p_i \in R$) and ($t_{ui}^{load}$ < load-threshold))
  $p_i' = InstantiateSoftCores(t_u, p_i, S)$
  Assign $t_u$ to $p_i'$
```

Note that this approach to hierarchical configuration does not enlarge the solution space with respect to spatial task assignment freedom. This correction factor only optimizes the use of the FPGA fabric tile originally found by the generic heuristic. In essence, this means that if the generic heuristic cannot find an assignment solution, the hierarchical correction factor is not used. In order to really exploit the improved spatial task assignment freedom, one needs to add hierarchical configuration into the main flow of the generic heuristic (Figure 3.9(b)).

3.8.4 Hierarchical Support into the Generic Heuristic

Before deciding on *how* to add hierarchical configuration support into the main flow of the generic heuristic, one needs to determine if it makes sense and, if so, in what occasions hierarchical configuration is useful. To this end, we add hierarchical configuration support to the full search algorithm.

Experiments using platform 1 and a high application load reveal that between 57% and 75% of the solutions selected as best by the full search algorithm use hierarchical configuration. Consequently, we analyze the properties of the tasks assigned to a softcore and we perform a qualitative analysis of the task graphs using a configuration hierarchy.

We notice that on average 86% of all tasks that are assigned to a softcore, support only one non-FPGA processor type (i.e. GPP, DSP or accelerator). When the FPGA task does not fit on a small FPGA tile, it might be possible to assign this task to a softcore that, in turn, *does* fit the small tile. When analyzing the effects of hierarchical configuration using platform 3 (i.e. containing two small FPGA tiles), we notice that several such situations do occur.

Further qualitative analysis shows that hierarchical configuration is mainly used in case of communication constraints. First of all, this occurs when a task has a lot of communication peers, but does not have reconfigurable hardware support. Ideally, this task is assigned to the center tile (i.e. tile four). This becomes a possibility using hierarchical configuration. This assignment should, on average, also reduce the hop-bandwidth product. Secondly, using hierarchical configuration occurs when an assignment to a non-FPGA tile cannot be done because of a communication load constraint to a certain communication peer.

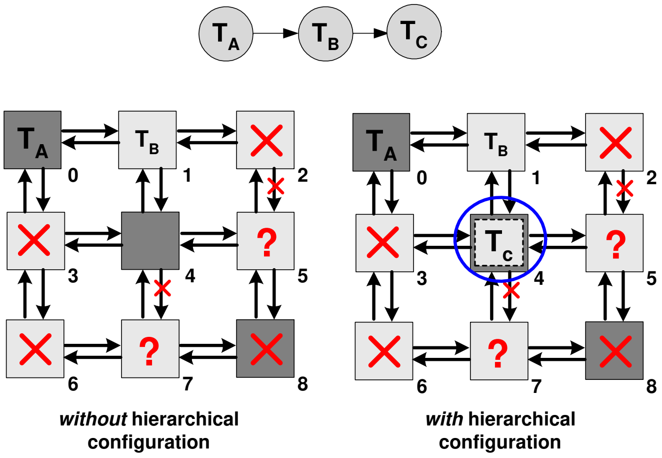

T_A
T_B
T_C
0
1
2
3
4
5
6
7
8
?
without hierarchical configuration
with hierarchical configuration

Figure 3.10: Hierarchical configuration rationale example.

Example 3.1: Hierarchical configuration to solve a communication load problem (Figure 3.10).
Consider the task graph containing tasks T_a, T_b, and T_c. Assume that task T_c still needs to be assigned, that T_c has no FPGA support (tile four) and all other tiles are occupied or unsupported. This means task T_c can only be assigned to either tile five or tile seven. Although both tiles can provide the required computing resources, they lack the required communication resources to support the communication between T_b and T_c. Without hierarchical configuration, the heuristic has no other option but to reconsider the assignment of T_a and/or T_b (i.e. perform backtracking) or to migrate tasks of previously assigned applications in order to free up resources. However, by means of hierarchical configuration, T_c can be mapped on a softcore instantiated on FPGA tile four. Also from a hop-bandwidth point of view (i.e. assignment quality), it is better to map T_c on a softcore on FPGA tile four.

Now that the benefits of using a configuration hierarchy are clear, we still need to introduce it into the generic heuristic. The initial idea of handling it as an alternative to backtracking [157] (i.e. only using hierarchical configuration when backtracking fails), did not yield the expected performance increase. Consequently, we add the hierarchical configuration concept into the main flow just before prioritizing the tiles. This is illustrated by Figure 3.9(b) and detailed in Algorithm 6.

The main difference of Algorithm 6 with respect to the generic heuristic (Algorithm 3) is that before determining the cost of the tiles, we determine if using a softcore is required. In case task t_i supports only one processor p_i (line 3), we prioritize its supported softcores $s_j \in S$ and instantiate them on the available FPGA fabric tiles $p_j \in R$ (line 4). Algorithm 5 details the process of prioritizing softcores and instantiating them onto the FPGA tiles.

After instantiating the softcores (line 4), we determine the assignment cost for every tile and consequently sort them according to ascending cost (Algorithm 3, lines 2-4). In Algorithm 6 this is denoted by $PrioritizeTiles$. Notice that the instantiated softcores are treated just as if they were normal processors. The underlying FPGA fabric tiles are no longer visible, meaning that the set of available processor tiles is now P' instead of P. Hence the difference between the $AG'(P', L)$ and the $AG(P, L)$ parameter of $PrioritizeTiles$ (line 5 and line 7 respectively).

After the task is assigned we need to introduce an extra step with respect to the generic heuristic, namely to remove all unused softcores in order to expose the FPGA fabric again (line 10).

This approach to handling a configuration hierarchy allows to combine backtracking and hierarchical configuration. If, during backtracking, a task is removed from a softcore and if no other task still uses that softcore (which is possible in case of multitasking softcores), the softcore itself is also removed. This way, the heuristic can opt for a different solution that does not make use of a softcore or that uses a different softcore.

Algorithm 6: *Mapping a task graph onto an architecture graph with a hierarchical configuration enabled heuristic.*

```
Input: $TG(T,C)$, $AG(P,L,S)$
Output: $TG(T,C) \rightarrow AG(S,P,L)$
HIERARCHYHEURISTIC($AG(S,P,L), TG(T,C)$)
$PrioritizeTasks(TG(T,C),P)$
foreach unmapped $t_u$ with highest $Prio(t_u)$
  if $\exists! \tau_j$ such that $\tau_j(t_u) = 1$
    $P' = InstantiateSoftCores(t_u, P, S)$
    $N(t_u) = PrioritizeTiles(t_u, TG(T,C), AG'(P',L))$
  else
    $N(t_u) = PrioritizeTiles(t_u, TG(T,C), AG(P,L))$
  if $N(t_u) > 0$
    Assign $t_u$ to $p_i$ with lowest $Cost(p_i)$
    Remove unused softcores
  else
    if ($bt > 0$) and ($t_u \neq$ first task)
      repeat
        Undo allocation of previous task $t_v$
        Remove unused softcores
        $bt = bt - 1$
      until ($N(t_v) > 1$) or ($bt = 0$) or (first assignment)
    if (($bt = 0$) or (first assignment)) and ($N(t_v) \leq 1$))
      No solution found. Exit.
    else
      Assign $t_v$ to $p_j$ with second lowest $Cost(p_j)$
      $N(t_v) = 1$
```

3.9 Hierarchical Task Assignment Evaluation

This section contains two parts. First, we have a look at the impact of using a hierarchical correction factor. Secondly, we evaluate the performance of the hierarchical configuration heuristic with respect to the generic heuristic and the full search algorithm.

In order to assess the performance of the heuristic when using a configuration hierarchy, we use the same set of task graphs as for evaluating the generic heuristic (Section 3.7.1). However, we added a collection of seven softcore processing elements. Given the current availability of softcore processing elements (see Section 3.8.1), this should be the order of magnitude of softcores available to the run-time manager. Still, every task has support for at least one non-soft processor $p_j \in P$, also denoted as a *hardcore* processor, and, in total (i.e. softcore and hardcore), for no more than four PEs.

Each softcore has a task support rate between 5% and 15%. In case of a low or a high task load, a softcore is loaded for 35% and 80% respectively. Except for the hierarchical correction factor, we assume only one task per softcore.

Compared with the properties of the hardcore processor types (Table 3.2), these softcore assumptions are quite conservative. Indeed, within a specific application domain, a specialized softcore would outperform a hardcore general purpose processing element (resulting in less or similar load). In case of a more general purpose softcore (e.g. ARM Cortex-M1 or Xilinx MicroBlaze), the support rate could be as high as for a hardcore general purpose processing element. This means that the results are also on the conservative side.

3.9.1 Hierarchical FPGA Correction Factor

In order to determine the performance of the hierarchical correction factor, we use the same randomly selected task graph pairs as for evaluating the previous FPGA correction factors. We evaluate the assignment success rate of the second task graph after the first task graph was successfully mapped. In all of the experiments, both the communication and processor load of the platform is set to high, while the application load of the first task graph is always set to low.

Table 3.4 details the success rate improvement for the assignment of the second task graph (after the first task graph was successfully assigned) when adding a hierarchical correction factor into the generic heuristic.

Table 3.4: *Hierarchy correction factor success rate improvement (%).*

Improvement with respect to heuristic(9)			
TG(2/2) Load	**Platform 1**	**Platform 2**	**Platform 3**
L	+4.8	+1.9	+2.0
H	+8.7	+2.3	+6.1

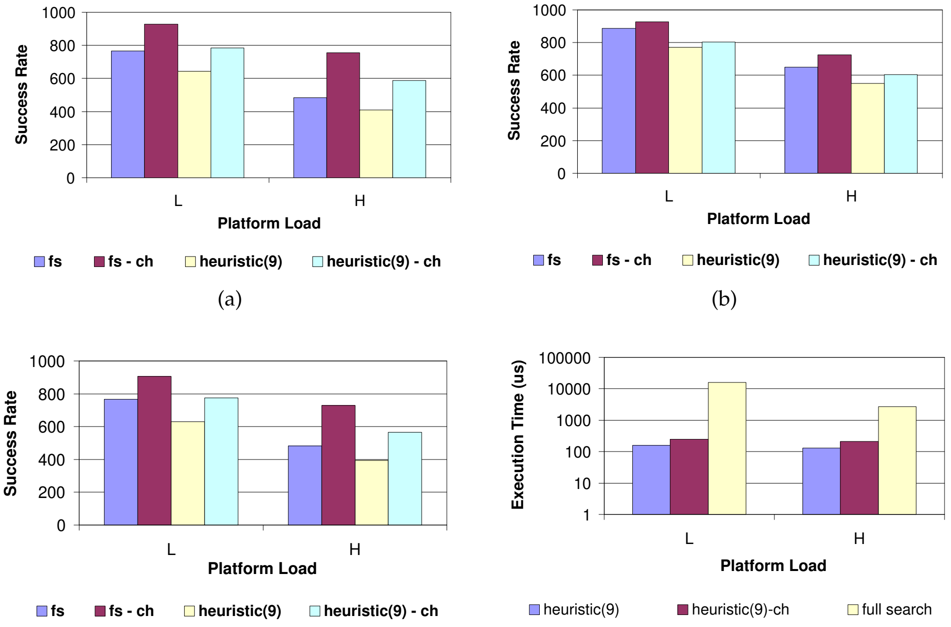
First of all, we notice that using the hierarchical correction factor yields significant improvements for platform 1 and platform 3. Reuse of a previously assigned softcore is important, especially in case of a high load for the second task graph. The improvement for platform 2 is less noticeable due to the fact that it has only one FPGA tile.

Note that, in case of a softcore with higher support rate or a more specialized softcore, the improvements would be higher.

3.9.2 Hierarchical Configuration Heuristic

Figure 3.11(a), Figure 3.11(b) and Figure 3.11(c) detail the (absolute) success rate for both the heuristic and the full search algorithm, both without and with *c*onfiguration *h*ierarchy support (denoted with *-ch*) for platform 1, platform 2 and platform 3 respectively.

For Figure 3.11(a) we first notice that using a configuration hierarchy clearly improves the task assignment success rate for both the full search algorithm (up to 27% better) and the heuristic (up to 20% better). Secondly, we see that the heuristic *with* hierarchical configuration support performs better than the full search algorithm *without* configuration support in three occasions. This means that, no matter how one would improve the generic heuristic, it could never outperform the success rate of the heuristic with hierarchical configuration support. This clearly illustrates the need for run-time managed softcore processors on MPSoC platforms with FPGA fabric tiles.



(a) (b)

(c) (d)

Figure 3.11: *Hierarchical configuration experimental results. Assignment success rate for a high load application on (a) platform 1, (b) platform 2 and (c) platform 3. (d) Execution speed for platform 1 and high application load using a StrongARM ISS (206 MHz)*

Platform 2 only contains one reconfigurable hardware tile. Obviously this limits the potential of using hierarchical configuration. Although we notice a significant performance improvement for both the full search algorithm and the heuristic, but the heuristic *with* with hierarchical configuration never outperforms the full search algorithm *without* configuration hierarchy.

Platform 3 contains three FPGA fabric tiles, although two of them are small. This implies that some FPGA tasks as well as three out of seven softcores will not fit on these reconfigurable hardware tiles. Nevertheless, we notice that the success rate for platform 3 is almost as good as for platform 1. This is due to the fact that if the FPGA tile is not big enough to accommodate the (FPGA) task, in some cases (for about 16 to 22 task graphs) this can be solved by assigning that task to a softcore that does fit the FPGA tile.

Figure 3.11(d) details the speed of the hierarchical configuration algorithm. For platform 1 and platform 2, the hierarchical configuration heuristic requires on average (depending on the platform type and load) between 179 μs and 248 μs (stdev about

180 μs). Depending on the platform type, platform load and the task graph properties, exploring the full hierarchical search space can take up to several minutes.

How does using a configuration hierarchy influence the task assignment quality? As predicted (Section 3.8.4), using a configuration hierarchy reduces the overall average hop-bandwidth product due to the fact that, for some task graphs, tasks can be mapped closer together. As a consequence the communication load variance is higher (i.e. communication is more concentrated on fewer links). On the one hand, this is an artifact caused by assigning a single task graph to a balanced platform. On the other hand, the resulting overall platform communication load and the inter-application communication interference (Chapter 5) will reduce!

Table 3.5 details the average reduction in hop-bandwidth product for both platform 1 and platform 2 with respect to the hop-bandwidth product of the successfully assigned task graphs of the generic heuristic. Hence, the heuristic with hierarchical configuration support combines a higher success rate with a lower average hop-bandwidth. Furthermore, the fact that the reduction is higher for platform 2 than for platform 1 is due to the communication load spreading which results in a significantly lower communication load variance. In addition, one should bear in mind that the success rate for platform 2 is lower with respect to platform 1.

Table 3.5: *Average hop-bandwidth reduction of the heuristic with hierarchical configuration with respect to the generic heuristic.*

Hop-Bandwidth reduction (%) - Application Load High		
Platform load	**Platform 1**	**Platform 2**
L	0%	6%
H	4%	10%

3.10 Related Work

The related work is split into three parts. The first part (Section 3.10.1) focuses on run-time management of FPGA fabric with respect to placing dedicated hardware tasks. We show how our algorithms fit into this context and how hierarchical configuration can be a solution for some issues. The second part (Section 3.10.2) focuses on the state-of-the-art with respect to using a configuration hierarchy. As this chapter only used synthetic examples, it illustrates the practical usefulness of a configuration hierarchy. The third part (Section 3.10.3) focuses on resource assignment in heterogeneous MPSoC systems and should put the presented algorithms into perspective.

3.10.1 Run-Time Management of an FPGA Fabric Tile

Almost a decade ago, researchers [35,62,92] started realizing that FPGA fabric should not be considered as a peripheral device, but as a regular computing resource. This means that the management of the FPGA fabric should be up to the operating system (also denoted as run-time manager) that acts as an arbiter for the resource requests of different applications. This also means that a designer can no longer directly control the FPGA resource. Instead, the designer should focus on the application functionality.

Adding the reconfigurable hardware into the overall computational pool also means taking into account the radically different properties of FPGA fabric with respect to an instruction set processor. Hence, the run-time services that should be provided to the applications are very different. These services include, for example, task footprint transformation and task placement, managing fragmentation of reconfigurable fabric, dealing with platform security and integrity issues (like e.g. FPGA viruses [124] and preventing so-called *forbidden configurations* [93]), enabling FPGA multi-tasking and pre-emptive task switching, handling the reconfiguration overhead/time, etc.

When considering that multiple tasks have to share the same FPGA fabric computing resource, the run-time manager has to optimize the usage of this resource. However, there are several ways to model the FPGA resource [230].

First, one could consider the FPGA fabric as one large space, where tasks can be placed freely. Task placement is the problem of positioning a task with irregular footprint somewhere in the reconfigurable hardware fabric. This means that the run-time manager needs to optimize the usage of the available space by e.g. relocating, transforming and reshaping the tasks in order to fit as much tasks as possible into the space, while minimizing the *external fragmentation* [233]. Over the years, quite some effort has been put into developing such (complex) task fitting algorithms [35, 229, 231]. In addition, FPGA vendors like Xilinx provided tools like JBits [86] that facilitate those transformations for specific FPGA types.

These task placement algorithms induce a considerable run-time overhead. Consequently, one could pre-partition the FPGA fabric (1D or 2D partitioning). Each partition could then accommodate a single FPGA task. In addition, the communication infrastructure that interconnects the different tasks and provides them a link with the outside world could be fixed [144]. This way, the run-time overhead is seriously reduced. However, the penalty for pre-partitioning is *internal fragmentation* [233], i.e. the FPGA fabric that is wasted because the tile size is larger than the task size. Our work is based on using a pre-partitioned model.

Spatial and temporal task assignment goes hand in hand with fragmentation. Hence, most task placement and scheduling algorithms, especially when considering free placement models, take fragmentation into account [41, 81, 230]. Walder et al. [230], for example, use a 1D FPGA model with fixed partitions of various sizes to achieve a better match between resource size and task size. When a task needs to be executed, it is configured onto the smallest idle partition that can accommodate the task. Tasks are selected on a First-Come-First-Serve (FCFS) or Earliest-Deadline-First (EDF) basis. The authors only consider non-communicating tasks. We also consider internal fragmentation when choosing a tile and, in addition, we consider inter-task communication.

Also a lot of effort was spent in developing techniques for hiding or reducing the task setup latency. A large task setup time could mitigate the performance benefits of using an FPGA. Surely, task setup latency has a close relation with the used FPGA model, but even when using a pre-partitioned model, the task setup time cannot be neglected. This is mainly due to the large amount of configuration bits that define a task. These bits need to be transferred and configured into the FPGA. Several solutions have been proposed: configuration caching, prefetching, partial reconfiguration, bitstream compression, difference configuration, multi-context FPGAs, etc. Compton et al. [46] provides a brief overview of these solutions. Resano et al. [186, 187] proposes a prefetching technique in order to have the FPGA task ready for execution when needed. Similar to our FPGA correction factors, the authors extended an existing scheduler in order to incorporate the task setup overhead scheduling effects. Compression of the amount of configuration data that needs to be transferred and configured is another approach. This can be achieved by either only configuring the parts that are different between two tasks [41, 110, 212] and/or by only configuring a small part of the FPGA denoted as *partial reconfiguration*. By using a configuration hierarchy, it is possible to mitigate the task setup latency for tasks that share or re-use a soft IP core [140, 158]. In a sense, this is similar to re-using part of the configuration of previously executing tasks except that the reuse exploitation is not at bitstream configuration level but at a higher abstraction level.

Enabling FPGA preemptive multi-tasking has been another focus of the reconfigurable computing community. The main idea is that FPGA fabric is a scarce resource and should be time-multiplexed. Meaning that the FPGA fabric is used by more tasks than it can accommodate at a single moment in time. This requires a reduced the task setup time. It also requires retrieving and restoring the state of the FPGA task. Classically it is done by reading back the FPGA configuration bits and extracting the relevant state bits [86, 107, 122, 203]. We have shown that time-multiplexing the FPGA resources can be done by raising the abstraction level, i.e. by introducing a soft IP core that handles the preemption in a correct way.

3.10.2 Hierarchical Configuration

Recently, FPGAs have become large enough to accommodate a significant amount of soft IP cores. In addition, most FPGA vendors already provide a set IP of cores to ease FPGA task development. Hence, the next challenge for run-time management of reconfigurable hardware is the ability to efficiently handle a configuration hierarchy, i.e. to use soft IP cores besides handling FPGA fabric with respect to dedicated hardware tasks.

Schaumont et al. [198] coined the term hierarchical configuration and described the configuration design space by means of three axis (Figure 3.12). The *vertical* axis describes the level of abstraction. From a hardware point of view it relates to gates and registers at the low level and to instruction set processors and complete systems at the high level. From a software point of view, it corresponds to having a virtual machine executing instructions using the functionality and primitives provided by the underlying abstraction layer. The *horizontal* axis describes the reconfigurable feature diversity. This axis is typically associated with terms as coarse-grained and fine-grained reconfigurability with respect to communication, computation and storage elements. The *time* axis denotes the binding time, i.e. the time when configuration data is send to the processing part. On one end of the spectrum there is design-time binding, i.e. configuration together with the soft IP core instantiation, on the other

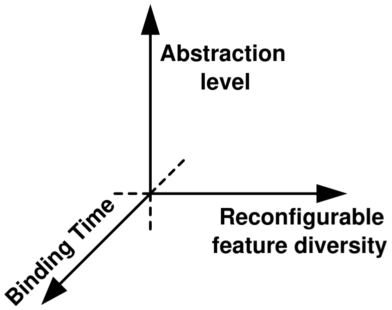


Figure 3.12: Configuration hierarchy design space.

end of the spectrum there is run-time binding, i.e. configuration when the processing is actually needed.

In addition, Schaumont et al. describe a design-time technique to determine the right point in the configuration design space. This can be achieved by profiling a set of applications from a certain application domain. This way, one can determine a set of commonly used, computationally intensive kernels. Parameterizable implementations of these kernels form the building blocks of the reconfigurable platform. Consequently, these blocks are pre-instantiated into the fully reconfigurable fabric. At run-time these soft IP blocks can be *programmed* with a minimal amount of *configuration* input. The authors demonstrate this concept with an image processing example. Consider a *Filter Operations Block*, where the filter coefficients can be provided as *programming* input. This way, it is possible to use the same block for a wide variety of filtering operations (e.g. edge detection, noise reduction, sharpening, etc.). The authors suggest to use compiler techniques to map an application onto a given set of parameterizable IP blocks.

The ThumbPod, an embedded fingerprint authentication system, illustrates the usefulness of a configuration hierarchy [197]. Due to the high design complexity, it is next to impossible to capture everything in one abstraction level. Instead, the system is composed as a stack of three machines: the bottom layer consists of a Virtex-II XC2V1000 FPGA. A LEON2 softcore processor with two co-processor components is instantiated on top of the FPGA. On top of the LEON2 processor executes an embedded Java virtual machine. Using a configuration hierarchy not only enables easy programming of the embedded system, it also allows the programmer to fully exploit the Java security architecture.

Keller et al. [108] describe the use of so-called *software decelerators*. By using freely available soft IP cores, the designer can take advantage of an *easier* application design process (i.e. a software design process instead of a hardware design process). In addition, certain algorithms use fewer hardware resources when implemented on a soft IP core (i.e. a sequential machine), while still meeting the necessary performance requirements. In addition to describing the desirable conditions for using a soft IP core, the authors describe a case study based on a finite state machine.

Memik et al. [140] describe a system architecture, denoted as *Strategically Programmable System* (SPS), that contains a set of *Versatile Programmable Blocks* (VPB) that are preplaced within the fully reconfigurable logic. When implementing an application, functionality will be mapped as much as possible onto those VPBs. This way, not only the amount of configuration bits required to represent an application can be drastically reduced, it also results in a diminished configuration time (i.e. application setup time). The generation of the VPBs and an SPS instance is automated. Given a set of applications, a design-time tool determines the amount and the types of the VPBs that will be instantiated. An image processing application is used as a proof-of-concept. This results in three VPBs: a parameterizable filter block, a thresholding block for simple pixel operations and a pixel modification block.

Jin et al. [104] detail a design-time exploration framework for designing FPGA-based multiprocessors using soft IP cores. The goal is to come up with an architecture (processing elements, interconnect and memory) that minimizes the makespan of a given task graph. As a proof-of-concept, the authors have mapped an IPv4 packet forwarding application onto a Virtex II Pro using the MicroBlaze soft general purpose microprocessor [167]. This is the configuration hierarchy equivalent of freely placing dedicated hardware tasks into FPGA fabric.

All the presented related work considers using a configuration hierarchy from a design-time point of view. This way, the design can be done (a) in a more efficient way by separating complex issues, (b) in a faster way by using a software design process and (c) in a more resource-efficient way. Although task assignment is typically part of the run-time resource manager, none of the authors considered having a run-time manager controlling the configuration hierarchy. However, the work of Memik et al. [140] could prove very useful when determining a suitable set of softcores for a certain application domain. To the best of our knowledge, our work [158, 161] presents the first run-time manager for reconfigurable systems capable of handling softcores and exploiting a configuration hierarchy.

3.10.3 Heterogeneous MPSoCs Resource Assignment

The generic resource assignment algorithm should be put into perspective with existing MPSoC resource assignment algorithms, both static (i.e. design-time assignment) and dynamic (run-time assignment).

Smit et al. [206, 207] describe a run-time task assignment algorithm based on the MinWeight algorithm [33]. The algorithm was designed to map a task graph at run-time to a tiled heterogeneous platform containing general-purpose processing elements, DSPs, Domain Specific Reconfigurable Hardware (DSRH) tiles and FPGA fabric tiles. The MinWeight algorithm takes only a few milliseconds to come up with an assignment solution. The algorithm takes the scarcity of resources into account. This means that, quite similar to our approach, the algorithm should map the tasks that need *a scarce resource* before all other tasks. In essence, this basic mapping technique bears some resemblance with our generic heuristic. Although the algorithm clearly targets architectures containing reconfigurable hardware and although the authors acknowledge that scarcity of resources can be problematic for the performance of the algorithm, they do not propose to adjust the algorithm with respect to the specific reconfigurable hardware tile properties.

Hu et al. [100] present a design-time mapping heuristic that statically schedules both communication transactions and computation tasks onto a heterogeneous MPSoC. The goal of the heuristic is to minimize energy consumption. The heuristic operates in three steps. In the *first step* the slack for each task is budgeted based on its mean execution time on the different PEs and a weight factor. The result of the first step is for every task a budgeted deadline. In the *second step*, the communication is taken into account. This is done by calculating the earliest finish time of tasks, based on the execution time and the data ready time. Then, a ready task list (i.e. a list of tasks whose precedent tasks have already been scheduled) is composed and sorted based on the earliest finish time. For each task in the ready task list, a list of PEs is created that ensure that the deadline will be met for that task. Finally, the task with the highest energy sensitivity with respect to the PE list is scheduled on its lowest energy PE. Step two is repeated until all tasks are assigned and scheduled. The *third step* is concerned with fixing missed deadlines in the schedule by local task swapping and global task migration. Local task swapping will change the execution order of tasks on the same PE, while global task migration swaps tasks between PEs. The authors created several benchmarks with TGFF, each containing 500 tasks with about 1000 communication transactions and mapped them onto a heterogeneous 4x4 NoC. In contrast to our algorithm, this static algorithm also takes resource scheduling into account. As a result, the algorithm requires a large computation time.

Recently, Stuijk et al. [215] detail a design-time heuristic that maps an *Synchronous Dataflow Graph* (SDF) onto an architecture graph. The heuristic operates in three steps. The *first step* is responsible for *resource binding*, i.e. assigning an SDF task, also denoted as *actor*, to a tile in the architecture graph. Similar to our approach, the heuristic first sort the tasks based on their relative importance with respect to the throughput of the application. Secondly, the heuristic sorts the tiles in order to balance the load of the application over all tiles. Consequently, each task is assigned to a tile. Finally, after all tasks have been assigned, the heuristic revisits the allocation of each task in order to balance the load of the tiles. The *second step* involves constructing, for each tile, a static schedule for its assigned tasks. The *third step* involves allocating the processor time-slices for each tile. The authors used the SDF^3 tool to generate a synthetic application benchmark and they considered three different architecture graphs, each with a 3-by-3 mesh architecture containing three different processor types. Their experiments also reveal a difference in mapping success rate based on the load of the platform and the applications. As a real-life example, the authors are mapping three H.263 video decoders (4 tasks each) and an MP3 decoder (13 tasks) onto a 2-by-2 mesh with 2 GPP and 2 accelerators, the algorithm requires, on 3.4GHz Pentium 4 processor, 8 minutes of run-time of which 90% is consumed by the time-slice allocation. In contrast to our approach, Stuijk et al. [215] also provide a task schedule for each tile with multiple tasks.

The generic heuristic (Algorithm 2) currently assumes a pre-existing path between tiles. However, as Section 3.4 briefly explains, one could extend this algorithm with a dynamic path finding heuristic, i.e. a heuristic that finds a path between a candidate tile for a certain task and its already assigned communication peers. In this context, Marescaux et al. [131] detail a fast run-time path finding heuristic. The heuristic operates on a TDMA guaranteed throughput NoC and is responsible for both path finding and time-slot allocation. This is achieved by walking a space-time graph by means of an *Iterative Deepening Algorithm*IDA. In their experiments, the time required to allocate a single path is dependent on the communication load and the number of hops: the heuristic requires, on average, 5 μs per hop per time-slot.

3.11 Conclusion

First, this chapter details a fast generic run-time resource assignment heuristic for a heterogeneous MPSoC platform. This generic heuristic produces very good results in terms of assignment success rate, quality of the assignment and speed of the algorithm when compared to an algorithm that explores the full solution space.

By incorporating specific FPGA fabric support that considers FPGA tile fragmentation and the fact that such a tile can only accommodate one task the assignment success rate can be further improved up to 8%.

Secondly, we show that handling soft IP cores is the next challenge for run-time management of reconfigurable systems. This is generally denoted as managing a *configuration hierarchy*. The rationale for using such a configuration hierarchy is that it provides both design-time (e.g. easier development) and run-time (e.g. spatial task assignment freedom) benefits.

Hence, we detail how to integrate support for managing such a configuration hierarchy into the generic heuristic. We show that exploiting a configuration hierarchy, can significantly improve the performance of the run-time task assignment algorithm. This entails increasing its assignment success rate (up to 27% for full search and up to 20% for the heuristic) and improving the task assignment quality (up to 10%, depending on the platform). In some cases, the average heuristic success rate improvements even exceed searching the full solution space without hierarchical configuration, while using only a fraction of the execution time.

CHAPTER 4

Task Migration in the MPSoC Environment

Applications targeted at MPSoC systems are typically composed of communicating tasks. These application tasks are assigned by a run-time manager onto the heterogeneous platform tiles. During the resource assignment phase, just after starting the application, the run-time manager takes the availability and suitability of the platform resources into account. However, varying run-time conditions (e.g. new user requirements, new incoming applications, etc.) can create the need to revise, in a flexible way, the initial resource assignment of one or more already executing tasks.

Hence, the run-time manager requires *run-time task migration* capabilities, i.e. a way to move tasks to a different tile without the need to completely stop and restart the application. The run-time task (or application) migration concept has been widely explored in the *Networks-Of-Workstations* (NOWs) environment. Besides the conceptual similarities between a NoC and a NOW (e.g. multiple processing elements, packet-switching, routing), there are some non-negligible differences like e.g. the available memory and the inter-task communication protocols. So there is a need for task migration techniques *tailored* to the NoC environment.

Besides introducing the run-time task migration concept, benefits and issues,this chapter contains four parts. The first part (Section 4.2) introduces a run-time task migration policy linked to the run-time task assignment heuristic. The second part (Section 4.3) tackles the HW/SW task migration issue. It shows how to migrate tasks between an FPGA tile and an ISP tile (and vice versa). This involves providing the right design-time and run-time infrastructure. The third part (Section 4.4) introduces and characterizes two novel task migration mechanisms targeted at the Networks-on-Chip environment. These mechanisms allow a global run-time manager with tile-local support functions to move one or more tasks at run-time between (heterogeneous) tiles. The final part (Section 4.5) deals with the migration initiation issue, i.e. how a cooperative migration request from the run-time manager will be detected and handled by the application. In addition, Section 4.6 discusses the related work and Section 4.7 presents the conclusions.

4.1 Run-Time Task Migration Introduction

Run-time task migration is a pretty broad topic. This section first introduces the run-time task migration concept. Consequently, it argues why it is beneficial to have support for run-time task migration both for classic multicomputer systems and for MPSoC platforms. Finally, this section provides an overview of the issues tackled by this chapter.

4.1.1 Concept

Run-time task migration can be defined as the relocation of an executing task from its current location, *the source tile*, to a new location, *the destination tile* (Figure 4.1).

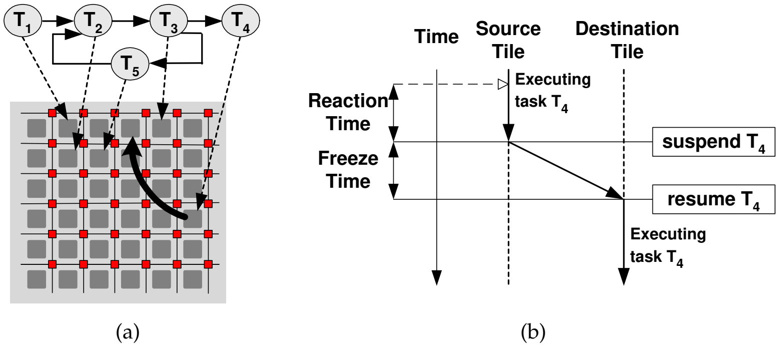


Figure 4.1: (a) After re-evaluating the task mapping on a tiled architecture, task T_4 is migrated from its source tile to a destination tile. (b) A sequence chart view of the migration mechanism employed to migrate T_4.

Run-time task migration is not a new topic and has been studied extensively for multicomputer systems since the beginning of the 1980s. Section 4.6 provides an overview of the current state-of-the-art run-time task migration mechanisms. However, most of these algorithms are not suitable for the MPSoC environment. In contrast to the components of a multicomputer system, the MPSoC tiles only have a limited amount of memory. Furthermore, the on-chip communication protocol significantly differs from the general protocols used for computer communication. The latter protocols provide a lot of flexibility, but have very low performance. Due to the specific characteristics of an on-chip network, such as a very low error rate and higher bandwidth, an on-chip (e.g. NoC) communication protocol will provide a different trade-off between performance and flexibility [119].

In addition, the granularity of the application tasks and their mapping will be different. Instead of containing a full-blown application, a tile only contains a single or a few tasks belonging to that application. In contrast to the multicomputer environment, this does not pose a problem, since the extremely tight coupling of the processing elements allows heavily communicating tasks to be mapped on different computing resources.

4.1.2 Benefits

One can distinguish two types of benefits of having task migration capabilities that a tile based system can exploit. On the one hand, there are the traditional benefits that also hold for classic multicomputer or multiprocessor systems. The traditional benefits are *improved system utilization* and the ability to *adapt to run-time QoS changes*. On the other hand, there are a few benefits that are specific to the MPSoC tile-based environment.

With respect to improved system utilization, task migration allows the run-time manager to implement a load-sharing mechanism. This avoids having some idle computing resources, while others are overloaded. This helps in improving the performance of the system as a whole. However, previous research [63, 67] has shown that due to the migration cost, task migration for load-sharing purposes is only beneficial for tasks that require a large amount of processing time with respect to the migration time. By minimizing the task migration cost, load sharing becomes more feasible and can be handled more effectively.

Adapting to QoS changes often involves task migration. Suppose the processing requirements of an application rise during the course of its execution, for example due to user interaction, the run-time manager can react by migrating critical application tasks on parallel processing elements in the NoC.

Specifically for a tile-based MPSoC system, task migration allows the run-time manager to maximize energy savings. As Lu et al. explain [242], task migration potentially increases the effectiveness of the Dynamic Voltage and Frequency Scaling (DVFS) algorithms.

In addition, task migration could enable OS controlled dynamic thermal chip management. It is well-known that controlling power density will be crucial in future (deep sub-micron) SoCs, since elevated die temperatures reduce device reliability, reduce transistor speed and increase leakage current exponentially. One potential solution presented by Heo et al. [95], denoted as *activity migration*, proposes to move computation from one processor core to another one in order to keep die temperature under control.

Finally, task migration can enable the run-time manager to manage the NoC communication by clustering tasks (i.e. placing them on the same tile or on adjacent tiles) that have a high inter-task communication bandwidth. This technique can also be used to minimize communication interference between parallel applications

[162]. Figure 4.2a illustrates an inter-application interference situation between two producer-consumer pairs (T_1, T_2) and (T_x, T_y). The run-time manager can decide to dynamically migrate the consumer task T_2 from tile 8 to tile 7 in the NoC (Figure 4.2b) and hence avoid communication interference.

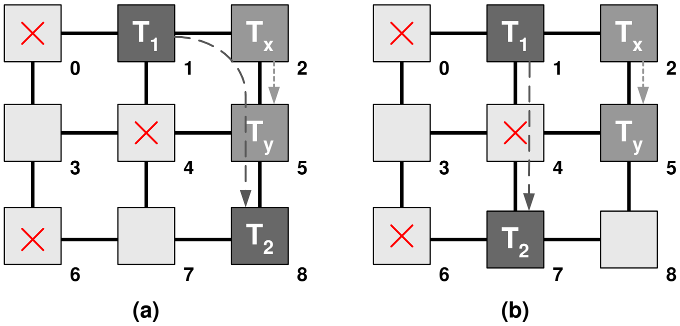


Figure 4.2: Migrating task T_2 from tile 8 to tile 7 removes the inter-application communication interference.

4.1.3 Issues

One should make a distinction between the migration mechanism and the migration policy. The *policy* is responsible for deciding on the migration of tasks in reaction to varying run-time conditions. Section 4.2 discusses our migration policy. The task migration *mechanism* is responsible for performing the actual task migration according to the decisions made by the policy.

Essentially, the migration mechanism needs to address three issues. First of all, it needs to capture and transfer the state of the migrating task in order to seamlessly continue task execution on the destination tile. Section 4.3 discusses this issue in the context of run-time HW/SW migration. Secondly, it needs to efficiently manage the continuing communication between the migrating task and the other tasks of the application. This is denoted as *message consistency*. Section 4.4 details two novel migration mechanisms that ensure message consistency. Finally, task migration in a heterogeneous environment is a cooperative process. This means that a good collaboration between the task or the application and the run-time manager is essential to create a fast and efficient migration mechanism that avoids overhead during regular task execution. In that context, Section 4.5 details a novel migration initiation mechanism.

4.1.4 Migration Mechanism Benchmarking

This section details the benchmark properties of a task migration mechanism. These properties will allow us to compare the performance of different mechanisms. A good task migration mechanism should exhibit:

- **Minimal reaction time.** As Figure 4.1(b) illustrates, the reaction time is defined as the time elapsed between selecting a task for migration until the task is actually ready to migrate (i.e. it reached its switchpoint).

- **Minimal freeze time.** The migration mechanism should cause as little interruption as possible to the execution of the migrating task (and hence to the entire application). This means that the freeze time, illustrated by Figure 4.1(b), needs to be minimized. This can be achieved on one hand by minimizing the time needed to capture and transfer the task state, on the other hand by minimizing the effort required to maintain message consistency.

- **Minimal residual dependencies.** Once a migrated task has started executing on its new tile, it should no longer depend in any way on its previous tile. These residual dependencies are undesirable because they waste both communication and computing resources.

- **Minimal system interference.** Besides causing minimal interference to the execution of the migrating task, the migration mechanism should avoid interference with other applications executing in the NoC or with the system as a whole.

- **Maximum scalability.** This property determines how the migration mechanism copes with an increasing number of tasks and tiles in the NoC.

- **Minimal steady-state run-time overhead.** Most of the task migration mechanisms introduce some run-time overhead by adding statements and function calls into the original source code in order to enable task migration. Obviously this overhead should be minimized during the regular execution. One way to reduce this overhead is to minimize the amount of time needed to detect a migration request.

4.2 Task Migration Policy

Task migration is useful in two distinct situations. First, in case the generic mapping heuristic (Chapter 3) is unable to assign all tasks of a newly started application. Secondly, when the quality requirements of an already running application change due to user interaction.

We distinguish two ways of adding task migration capabilities to the generic task assignment heuristic (detailed in Section 3.4 of Chapter 3). First, one can add task migration functionality when backtracking is impossible or fails (Figure 4.3(a)). Secondly, task migration could be a result of using the heuristic to co-assign an already assigned application with a newly arrived application (Figure 4.3(b)). These techniques are discussed in Section 4.2.1 and Section 4.2.2 respectively.

In addition to the symbols introduced in Table 3.1 (Chapter 3, page 54), this section defines a few new symbols used in the description of the task migration algorithms. They are briefly summarized in Table 4.1.

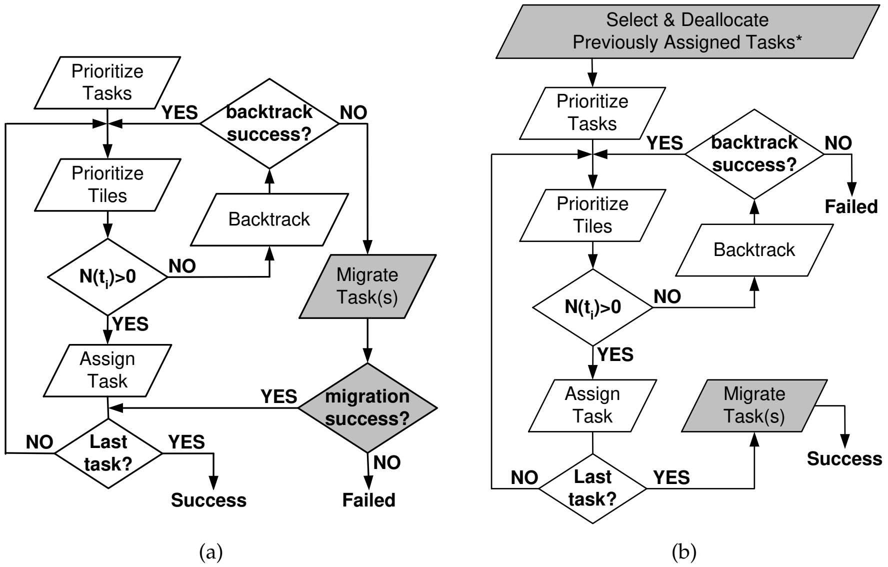


Figure 4.3: Two ways to add task migration functionality (gray) into the generic task assignment heuristic (white): (a) after backtracking failed and (b) as add-on before and after the generic heuristic.

4.2.1 Migration After Mapping Failure

Integrating the task migration concept into the task assignment heuristic is not trivial. The task migration policy should not interfere with the backtracking and vice versa. Since task migration is a costly operation and interferes with an already executing application, one should primarily rely on backtracking when no suitable processing elements are available for a specific task. Only when backtracking is no longer possible or fails, task migration should be considered.

Algorithm 7 details the generic task assignment heuristic (Algorithm 3, Chapter 3) augmented with task migration capabilities. The overall idea of the generic task assignment heuristic is to (1) prioritize the application tasks based on their relative importance, (2) for every task prioritize the tiles and (3) to assign the task to the best tile. Backtracking is used in case of a task assignment failure. This entails changing one or more previous assignments to solve a later assignment problem.

However, Algorithm 7 has three important differences with respect to Algorithm 3. First, the algorithm keeps track of the *best* partial application assignment so far. This means that the latest partial application assignment with most tasks assigned is stored (line 6) for later retrieval. Secondly, backtracking cannot be used in case the assignment of the first task fails. When that happens (lines 8-11), task migration can potentially solve the issue. Third, when the amount of backtracking steps is exhausted or when backtracking fails (line 22), task migration can be used[1]. This

[1] This is just the migration policy. The actual migration will only be performed after the assignment heuristic is finished.

	Symbol	Definition
Task Migration	M	Set of suitable migration candidate tasks
	t_v^{mig}	migration candidate task $t_v^{mig} \in S$
	$TG(T,C)_{prev}$	Previously assigned task graph
	t_u^{prev}	Previously assigned task $t_u^{prev} \in TG(T,C)_{prev}$
	$TG(T,C)_{new}$	Newly started, unassigned task graph
	t_u^{new}	Unassigned task $t_u^{new} \in TG(T,C)_{new}$
	$TG(T,C)^{load}$	Resource load of $TG(T,C)$
	$AG(P,L)^{load}$	Load of platform $AG(P,L)$
	$TG(T,C)_{superset}$	Sum of multiple $TG(T,C)_i$
	$AG(P,L)_{prev}$	State of the platform after deallocating $TG(T,C)_{prev}$
	δ	Number of (remaining) unassigned tasks
	p_{src}	Migration source tile, current location of t_v^{mig}
	p_{dst}	Migration destination tile, future location of t_v^{mig}
	p_{target}	Target (desired) tile for unassigned task t_u^{new}

Table 4.1: *Additional symbols used for task migration.*

means retrieving the best partial assignment and then finding a assignment for the remaining unmapped tasks.

This algorithm checks if the number of unassigned tasks in the stored partial assignment is below a certain threshold δ. If too many unmapped tasks remain, task migration could become too time consuming and too disruptive for other applications. This implicitly assumes that most application assignment failures are caused by not being able to map the last task(s). In this case (line 18), we expect maximally δ remaining unmapped tasks. These unmapped tasks can either be directly assigned (i.e. $N(t_u) > 1$) or a task migration needs to be performed. In the rare case that some backtracking steps still remain after reaching the first assigned task during backtracking, the number of remaining backtracking steps is set to zero. This avoids mixing backtracking with task migration. The actual algorithm to handle the task migration is detailed by Algorithm 8.

The goal of Algorithm 8 is to migrate a *single* task t_v^{mig} of a previously assigned application from its current (source) tile p_{src} to a destination tile p_{dst} in order to create a suitable location p_{target} for a task t_u^{new} of a new incoming application.

After determining the set of suitable migration candidates M, the algorithm has to optimally decide on (1) where to place the unmapped task t_u^{new} and (2) which previously assigned migration candidate task t_v^{mig} to migrate.

The first decision can be made using the hop-bandwidth product ϕ^{new} of task t_u^{new}. As Section 3.4 explains, considering the hop-bandwidth product ϕ^{new} (defined in Algorithm 3) ensures that heavily communicating tasks are mapped closely together. In this case, it helps determining to which tile the unmapped task should ideally be assigned to.

Determining which previously assigned task is the best migration candidate can be done using the mapping priority $MapPrio(t_v^{mig})$. As Section 3.4 explains, the mapping priority of a task determines its assigning importance both with respect to com-

Example 4.1: Finding a suitable task migration candidate (Figure 4.4.)
In this example, task T_B is the only remaining unassigned task after all backtracking steps are exhausted. Task T_B is capable of running on tiles 2, 5 and 6 (i.e. the candidate tiles). For every candidate tile, the *FindMigrationCandidateTasks* function (line 1 of Algorithm 8) determines the reason why task T_B cannot be assigned to these tiles. The assignment failure of task T_B on tile 6 is due to the fact that task T_4 consumes too much compute power. Assigning T_B to tile 6 will require task T_4 to be migrated. In this case, p_{src} and p_{target} are the same. In contrast, tile 2 has plenty of compute power. In this case, however, there is a communication resource problem when using a XY routing scheme. As task T_1 communicates with both T_2 and T_3, the link between tile 1 and tile 2 cannot provide enough communication resources to accommodate the communication between T_A and T_B. This means that placing T_B on tile 2 will require to migrate T_1 to for example tile 3. This example illustrates that the source tile p_{src} for migration and the target tile p_{target} for the unmapped task can be different. Finally, tile 5 cannot accommodate T_B because of a combination of problems. First, T_2 consumes too much compute power and would have to be migrated. Secondly, T_1 would have to be migrated as well in order to enable communication between T_A and T_B. This case is *not* supported by Algorithm 8 as it only supports migrating a single task.

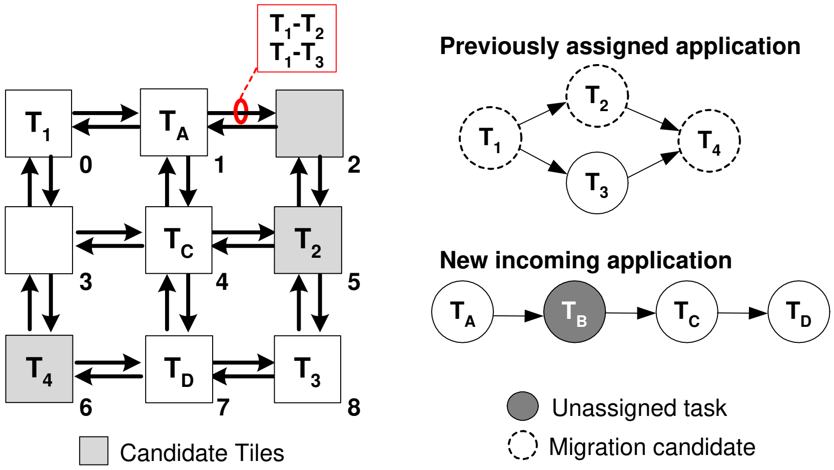


Figure 4.4: Task migration candidates of a previously assigned application with respect to an unmapped task of a newly incoming application.

Algorithm 7: *Resource assignment with migration after mapping failure (Figure 4.3(a)).*

```
Input: $TG(T,C)$, $AG(P,L)$
Output: $TG(T,C) \rightarrow AG(P,L)$
MIGRATIONHEURISTIC($AG(P,L), TG(T,C)$)
(1)  $PrioritizeTasks(TG(T,C), P)$
(2)  foreach unmapped $t_u$ with highest $Prio(t_u)$
(3)    $N(t_u) = PrioritizeTiles(t_u, TG(T,C), AG(P,L))$
(4)    if ($N(t_u) > 0$)
(5)      Assign $t_u$ to $p_i$ with lowest $Cost(p_i)$
(6)      $RememberPartialAssignment(\{t_u \rightarrow p_i, ...\})$
(7)    else
(8)      if ($t_u =$ first task)
(9)        $MigrateForTask(t_u, AG(P,L))$
(10)       if (Migration not successful)
(11)         Exit. //No solution found.
(12)     if ($bt > 0$)
(13)       repeat
(14)         Undo allocation of previous task $t_v$
(15)         $bt = bt - 1$
(16)       until ($N(t_v) > 1$) or ($bt = 0$) or (first assignment)
(17)     if (($bt = 0$) or (first assignment)) and ($N(t_v) \leq 1$))
(18)       if $PartialAssignment(TotalNumberOfTasks - \delta)$
(19)         $RestoreBestPartialAssignment(\{t_v \rightarrow p_j, ...\})$
(20)         $MigrateForTask(t_u, AG(P,L))$
(21)         if (Migration not successful)
(22)           Exit. //No solution found.
(23)       else
(24)         Exit. //No solution found.
(25)     else
(26)       Assign $t_v$ to $p_i$ with second lowest $Cost(p_j)$
(27)       $N(t_v) = 1$
```

putation and communication load requirements. A low mapping priority indicates that (1) the task location is not very important and that (2) the task could relatively easily be assigned to another tile.

A priority list of migration candidates $MigPrio(t_v^{mig})$ is created based on the product of the mapping priority of t_v^{mig} and the hop-bandwidth product ϕ^{new} of task t_u^{new} if it were assigned to the targeted destination tile (line 3). The rationale of determining the priority in such a way is that it finds a balance between finding the best tile for the unmapped task and finding the most appropriate task to migrate. In addition, one could consider adding a migration cost factor $MigCost(p_{src}, p_{dst})$. This application-independent cost factor could represent the system cost for migration tasks between heterogeneous tiles. This $MigCost$ would include components like cost of managing heterogeneous task state (Section 4.3) or ensuring message consistency (Section 4.4).

Consequently, the actual migration possibility is evaluated according to ascending $MigPrio(t_v^{mig})$ values. First, this includes finding a new location for the migrating task t_v^{mig}. This is done by prioritizing all supported and available tiles (details, see

Algorithm 8: *Task migration (Figure 4.3(a)).*

```
Input: $t_u^{new}, AG(P, L)$
Output: Assignment of $t_u^{new}$ by migrating $t_v^{mig}$)
MigrateForTask($t_u^{new}, AG(P, L)$)
(1)  foreach $p_i \in P$ supported by $t_u^{new}$
(2)    $M+ = FindMigrationCandidateTasks(p_i, t_u^{new})$
(3)  foreach $t_v^{mig} \in M$
(4)    $MigPrio(t_v^{mig}) = MapPrio(t_v^{mig}) \times \phi^{new} \times MigCost(p_{src}, p_{dst})$
(5)  foreach $t_v^{mig} \in M$ with lowest $MigPrio(t_v^{mig})$ value
(6)    $PrioritizeTiles(t_v^{mig}, AG(P \setminus p_{src}, L))$
(7)    if $(N(t_v^{mig}) > 0)$
(8)      Migrate $t_v^{mig}$ to $p_{dst}$ with lowest $Cost(p_{dst})$
(9)    else
(10)     continue
(11)   if $(CheckAssignment(t_u^{new}, p_{target}))$
(12)     Assign $t_u^{new}$ to $p_{target}$
(13)     Exit.
(14)   else
(15)     Undo migration of $t_v^{mig}$ to $p_{dst}$
(16)     continue
```

Section 3.4) *excluding* its currently assigned tile. Secondly, in case such a tile exists, the validity of assigning task t_u^{new} to the target tile p_{target} should still be checked (line 11) as new conflicts might arise (migrating because of computation load problems could e.g. result in new communication resource conflicts). If either step fails, the algorithm will try the next migration candidate with lowest $MigPrio(t_u^{mig})$.

4.2.2 Migration After Co-Assignment of Applications

Another way of tackling task assignment in combination with task migration is to perform application *co-assignment*. This means that prior to performing task assignment for a newly incoming application, the algorithm selects a set of previously assigned tasks and mixes them with the newly arrived tasks before starting the assignment algorithm (Figure 4.3b). In addition, the communication and the computation load imposed by the already assigned tasks are de-allocated, i.e. they are considered to be free for allocation. From then on, the $GenericHeuristic$ task mapping heuristic applies (Algorithm 3 in Section 3.4).

So in the first step of Algorithm 9, i.e. in the task prioritization step, the newly arrived unassigned tasks are mixed with the previously assigned tasks. This means that e.g. an assignment failure of the most important unmapped task can be amortized by performing backtracking in order to move a previously assigned task.

After the heuristic has found an assignment for all tasks, it has to compare the old tile assignment p_{src} of previously assigned tasks to the new assigned tile p_{dst}. If those differ, the task has to migrate.

In order to reduce the number of required migrations, tile prioritization with respect to a specific task of the generic heuristic (Algorithm 3) can be biased to e.g. *prefer* the

Algorithm 9: *Task migration by application co-assignment (Figure 4.3(b)).*

```
Input: $TG(T, C)_{new}, AG(P, L)$
Output: $TG(T, C)_{new}$ assignment and migration of $TG(T, C)_{prev}$ tasks.
APPLICATIONCOASSIGN($TG(T, C)_{new}, AG(P, L)$)
Select previously assigned task set $TG(T, C)_{prev}$
$AG(P, L)^{load}_{prev} = AG(P, L)^{load} - TG(T, C)^{load}_{prev}$
$TG(T, C)_{superset} = TG(T, C)_{prev} \bigcup TG(T, C)_{new}$
GenericHeuristic($AG(P, L)_{prev}, TG(T, C)_{superset}$)
foreach $t^{prev}_{u} \in TG(T, C)_{prev}$
  if $p_{src} \neq p_{dst}$
    Mark $t^{prev}_{u}$ for migration
```

tile it is currently assigned to hence avoiding a migration down the road. This biasing can be achieved by modifying the calculation of the tile priority (line 19 of Algorithm 3) with a *migration cost factor* (Equation 4.1). The cost factor $f(TG_{prev} \rightarrow AG)$ could be a simple constant like e.g. $1/4$ when $p_j = p_{src}$. This would favor the previously assigned tile. The cost function could be more complex like e.g. $f(p_{src}, p_{dst})$ when taking into account the effort required to retrieve and restore task state in case of a heterogeneous migration.

$$Cost(p_j)_{biased} = Cost(p_j) \times MigCostFactor \tag{4.1}$$

In contrast to Algorithm 7, *migration after co-assignment* can move multiple previously assigned tasks to solve an assignment problem of a single new task. With respect to the example of Figure 4.4, this would mean that both task T_1 and task T_2 could be migrated in order to assign task T_B to tile 5.

This technique could be used for switching resources between two simultaneously executing applications. This means e.g. one currently executing application moving from foreground execution to background execution, while another (newly started) application is brought to the foreground. This corresponds to one of our envisioned application user scenarios detailed in Section 6.2.1.

Selecting the right set of previously assigned tasks to be co-assigned is a separate remaining issue. Obviously, one needs to select the right amount of tasks with similar properties in order to properly enlarge the heuristic assignment search space. However, selecting too many previously assigned tasks will unnecessarily prolong the algorithm execution time, while selecting too few might result in an assignment failure. Furthermore, one needs to decide whether all selected previously assigned tasks should belong to a single application or to multiple applications. This decision will affect the enlargement of the search space as well as the overall system interference caused by the migration.

4.2.3 Experimental Setup

As the task migration policy builds on top of the generic task assignment heuristic of Chapter 3, we use a nearly identical experimental setup (see Section 3.6). In order to evaluate the performance of the migration policies, we use 20000 randomly selected task graph pairs (TG_i, TG_j) (the same pairs as in Section 3.7.2) based on 1000 task graphs generated by TGFF. Every task supports up to four PE types and, depending on user requirements, exhibits a PE-type specific processing and communication load. As experimental platform, we use Platform 1 detailed in Figure 3.4.

The rationale here is to evaluate the effect on the assignment failure rate of the *second task graph* (TG_j) after the first task graph (TG_i) was successfully assigned. In all of the experiments the application load of the first task graph is always low, while the load of the second task graph varies. However, in order to successfully complete the assignment of (TG_j), some tasks of the already assigned task graph (TG_i) can be migrated.

In order to evaluate the usefulness of both task migration policies, we assign the second task graph (TG_j) with the generic assignment heuristic (Algorithm 3), the full search algorithm (Algorithm 4, i.e. without migration capabilities), the heuristic with migration after mapping failure (Algorithm 7) and the heuristic with application co-assignment, i.e. co-assigning TG_i and TG_j (Algorithm 9) .

4.2.4 Experimental Results

Before evaluating the *migration after mapping failure* algorithm (Algorithm 7), we need to determine a suitable δ, i.e. the maximum number of unassigned tasks for using task migration. Figure 4.5 shows the failure distribution for assigning the second task graph when using the generic assignment heuristic. A distinction is made between failure of the *first task*, i.e. the task with the highest assignment priority, the failure of the *second last task* and failure of assigning the *last task*. The figure shows the variation of assignment failure with respect to the user defined application load (first letter) and the platform load (second letter).

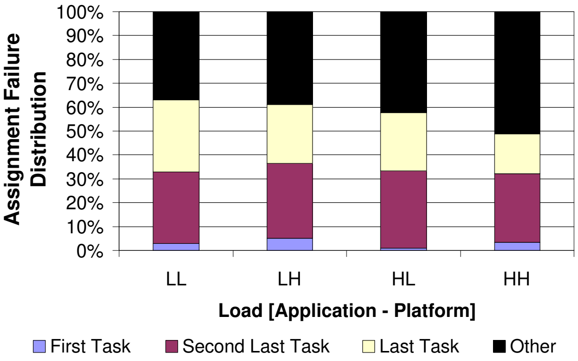


Figure 4.5: Assignment failure distribution for different application (first letter) and platform (second letter) load. This indicates that assignment failures are mainly caused by the last and the second last task. In case of a high platform load, failing to assign the first task also accounts for up to 5% of the assignment failures.

The mapping failure distribution shows that, for low application load, over 60% of all failure can be attributed to failure of either the first task, the second last task or the last task. Especially for the second last and the last task, it makes sense to perform a task-specific migration action. After all, these tasks have the lowest assignment priority, which either means that they do not have high platform resource requirements or that they e.g. support a variety of processor types. This makes it very likely to find an assignment solution by migrating one or more previously assigned tasks. Hence, it makes sense to set δ equal to two.

The rationale for performing migration in case the assignment of the first task fails is as follows. The first task represents the task which is most sensitive to mapping failure. This means (see Algorithm 1 on page 55) that this task requires a lot of platform resources and/or there is only one tile this task can be mapped onto. Intuitively it is easy to understand that, as there are no communication peers assigned yet, the reason for mapping failure is the unavailability of processing resources. Notice that assignment failure of the first task is most prominent in case of high platform load. Furthermore, if the first task fails there is no room for backtracking, in contrast to all other tasks.

Figure 4.6 details the assignment success rate of the second task graph with respect to the generic heuristic (heuristic(9)), the full search algorithm (fs), the generic heuristic with *migration after mapping failure* (heuristic-mig) and the *migration after co-assignment* (heuristic-duo) with 15 backtracking steps. In this case, the *migration after mapping failure* only migrates in case of assignment failure due to the first task, the second last task or the last task.

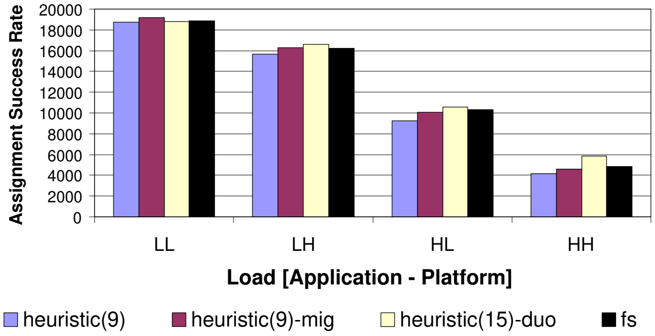


Figure 4.6: Assignment rate of the migration enabled heuristic algorithms versus the full search algorithm (fs) and the generic heuristic (heuristic(9)). The migration after mapping failure is denoted as heuristic-mig, while the migration after co-assignment is denoted as heuristic-duo.

We first notice the difference between the two task migration approaches. The *migration after mapping failure* performs better given a low application and platform load, while the *migration after co-assignment* performs better for high application and platform load. For low application load, the *migration after mapping failure* algorithm outperforms the full search (without migration) algorithm. The *migration after co-assignment* algorithm should be used in case of high application and high platform load, as this algorithm is not limited to only migrating for the first, the second last and the last task.

In general, the difference in performance between the algorithms can be explained by considering the size of their respective assignment solution search spaces and the way these algorithms respectively explore this search space.

The full search algorithm can traverse the entire solution space, however, *without* migrating tasks from previously assigned task graphs. This means that its search space is limited to the currently available resources, but any assignment within that search space can be reached.

The *migration after mapping failure* algorithm can use the available resources for task assignment. However, the heuristic assigns resources sequentially according to task priority and, for a given task, only the two best available computing resources are considered for assignment. This corresponds to searching a more limited space. For the first task, the second last task and the last task, the algorithm can use resources that are assigned to previously assigned tasks. However, the algorithm is limited to migrating a single previously assigned task to accommodate a failing newly arrived task. This explains the performance difference with the full search algorithm for low application load. As Section 3.7.1 explains, the performance of the generic heuristic already comes close to the performance of the full search algorithm for low application load. By enabling migration for the most failure-prone tasks, the heuristic can outperform the full search algorithm. In case of high application load and high PE load, the search space of the full search algorithm is very limited thus favoring the heuristic with migration capabilities.

The *migration after co-assignment* heuristic can use all available resources as well as the resources occupied by the previously assigned tasks that have been selected for co-assignment. However, this heuristic also assigns tasks sequentially according to their assignment priority and only considers the best two available computing resources for a given task. In contrast to the *migration after mapping failure* task, this algorithm can perform multiple task migrations for assigning a single newly arrived task. In addition, task migration is possible for all tasks and not just for the first or last and second last task. This explains why this algorithm outperforms the full search algorithm when computing resources are scarce (i.e. LH and HH). Scarceness of computing resources heavily limits the search space of the full search algorithm, but does not limit the co-assignment algorithm.

The reason for limiting the available search space is (1) to limit the execution time of the algorithm and (2) to limit the amount of task migrations. Limiting the execution time is needed because the algorithm will be executed at run-time. Limiting the amount of required task migrations is necessary because every migration will cause interference to another executing application.

The co-assignment algorithm with 15 backtracking steps requires on average about 670 μs when executing on a StrongARM (206MHz) general purpose processor. Similarly, Algorithm 8 requires on average[2] 395 μs for solving an assignment problem while migrating a single task. Considering that completing the generic heuristic requires up to 159 μs (Section 3.7.1), it does not make sense from a execution time perspective to consider performing more than two migrations using the *migration after mapping failure* algorithm.

The number of migrations resulting from the *migration after mapping failure* algorithm is limited by the design of the algorithm. In contrast, the amount of migrations required by the co-assignment algorithm can be quite large. In the worst-case scenario, all previously assigned tasks will have to migrate. In addition, it might be

[2] Non-optimized code with respect to data transfer optimizations.

that the co-assignment algorithm proposes a migration, while another, maybe less optimal, solution without migration would have been feasible. Without any precautionary measures, every successful assignment of the second task graph using the co-assignment algorithm will require on average 1.93 task migration actions. By using the constant *migration cost factor* of 1/4 in Equation 4.1 to bias the tile priority calculation, it is possible to reduce the average needed task migrations to 1.36 without significant impact on the assignment success rate.

4.3 Heterogeneous Task Migration Infrastructure

This section will focus on the global infrastructure required to migrate tasks between a general purpose ISP and fine grain reconfigurable hardware tiles [143, 144]. From that perspective, the problem of designing and managing relocatable tasks also fits into the more general research topic of hardware/software multitasking. However, the presented techniques can be used for heterogeneous processing elements in general.

As Chapter 3 explains, the run-time manager is responsible for deciding on the assignment of tasks based on the user requirements and the current platform resource usage. In that sense, the run-time manager abstracts the total heterogeneous computational pool in such a way that the application designer should not be aware on which computing resource the different tasks of the application will run.

The designer is, however, expected to create both a hardware and a software implementation of a single task in such a way that it is possible to move at run-time from one implementation to another at specific execution points. At these migration points or switchpoints, the tasks *state representation* should be transferable.

For the experiments described in this section, we used the OCAPI-xl design environment [227]. OCAPI-xl is a C++ library that enables unified hardware/software system design. Through the use of the set of OCAPI-xl objects, a designer can represent the application as a set of communicating tasks. Once the objects have been designed and simulated, automatic code generation for both hardware (HDL) and software (C code) is available. This ensures a uniform behavior for both the hardware and the software implementation of a single task.

From a run-time management point of view, there are two critical application interface components that need to be provided. First, a uniform communication API, which allows tasks to send/receive messages, regardless of their execution location, is required. Secondly, a mechanism to seamlessly transfer task execution at run-time from one implementation to another needs to be in place. These components are discussed in Section 4.3.1 and Section 4.3.2 respectively.

4.3.1 Uniform Communication Infrastructure

Migrating a task from hardware (an FPGA tile) to software (an ISP tile) should not affect the way other tasks are communicating with the migrated task.

By providing a uniform communication scheme for hardware and software tasks, the run-time manager hides this complexity. In our approach [144], inter-task communication is based on message passing.

Messages are transferred from one task to another in a common format for both hardware and software tasks. Both the run-time manager and the hardware architecture should therefore support this kind of communication.

During application mapping, the master run-time manager assigns a system-wide unique logical address to every task. Whenever a task is assigned to a certain tile, the master updates a *Destination Lookup Table* (DLT) residing within the local manager of its communication peers. This DLT is in fact an address translation table that enables the local run-time manager to translate a logical destination address to a physical address. The physical address of the destination task denotes the tile it is assigned to. The message passing API provided by the run-time management layers transparently performs this translation. Tile-local communication is handled by using only the logical address.

On the hardware side, the communication infrastructure provides the necessary support for message passing. In our specific case (see Chapter 6), the routing tables inside the switches actually translate the physical destination to a route.

4.3.2 HW/SW Migration Issues

It is possible for the programmer to know at design time on which of the heterogeneous processors the tasks preferably should run. However, the run-time manager does not guarantee availability of hardware tiles. Furthermore, the context switch latency of a typical hardware tile is in the range of 10ms [107,144], which severely limits the use of time-based hardware multiplexing (i.e. context switching). We therefore prefer spatial multitasking in hardware.

In order to achieve the desired user performance while consider a limited number of FPGA tiles, the run-time manager is forced to decide at run-time on the allocation of resources. Consequently, it should be possible for the run-time manager to pre-empt and migrate a task from the reconfigurable hardware logic to the ISP and vice versa.

The ISP registers and the task memory completely describe the state of any task running on the ISP. Consequently, the state of a preempted task can be fully saved by pushing all the ISP registers on the task stack. Whenever the task gets rescheduled at the ISP, simply popping the register values from its stack and initializing the registers with these values restores its state. This approach is not usable for a hardware task, since it depicts its state in a completely different way: state information is held in several registers, latches and internal memory, in a way that is very specific for a given task implementation. There is no simple, universal state representation, as for tasks executing on the ISP.

Nevertheless, the run-time manager will need a way to extract and restore the state of a task executing in hardware, since this is a key issue of enabling heterogeneous task migration. We propose to use a high level abstraction of the task state information. This way the run-time manager is able to dynamically reassign a task from the ISP to the reconfigurable logic and vice versa.

Figure 4.7a represents a migratable task, containing several states. This task contains two migration point states, at which the run-time manager can migrate the task. The entire migration process is described in detail by Figure 4.7b.

In order to migrate a task, the run-time manager can signal that task at any time $^{(1)}$. Whenever the signaled task reaches a migration point, it goes into the interrupted state $^{(2)}$. In this interrupted state all the relevant state information of the migration point is transferred to the run-time manager $^{(3)}$. Consequently, the run-time manager will re-initiate the task on the other tile by using the received state information $^{(4)}$. The task resumes by continuing to execute in the corresponding migration point $^{(5)}$.

The task described in Figure 4.7a contains multiple migration points (M), which makes it possible that the state information that needs to be transferred to the run-time manager can be different for each migration point. Furthermore, the unified design of both the ISP and FPGA implementation of a task ensures that the position of the migration points and their respective state information are identical.

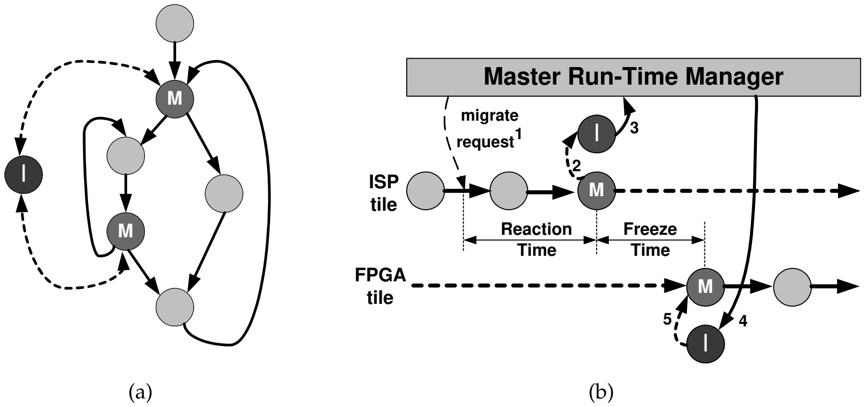


Figure 4.7: (a) Description of a migratable task containing two migration points M and an interrupted state I and (b) the HW/SW task migration process.

A design-time tool could be used to automatically insert these HW/SW migration points when the target architecture and the reaction and freeze time are given. The run-time manager will then use these migration points to perform the migration hidden from the designer.

4.3.3 A HW/SW Migration Case Study

The HW/SW migration experiment [144, 156] is performed on a platform[3] containing three tiles: a StrongARM tile, physically residing inside a Compaq iPAQ PDA) and two partially reconfigurable FPGA tiles inside a Virtex2 XC2V6000 device (speed grade -4). The tiles are interconnected by means of a packet-switched on-chip net-

[3] The platform used for the HW/SW migration experiment is actually the first *Gecko* platform. It is slightly different from the *Gecko*^{2} discussed in Chapter 6.

work. The master run-time manager executes on the StrongARM, while the local run-time manager is implemented in hardware [144, 156].

The case study consists of a motion JPEG video decoder application. The application contains three communicating tasks: a *send thread*, responsible for reading the encoded stream from disk and sends it to the *decode thread*, one macroblock at a time. The decode thread decodes the stream. Finally, the *receive thread* reconstructs the images and displays the video.

The send thread and the receive thread both execute in software on the iPAQ. The decode thread has both a hardware and a software implementation both derived from a single OCAPI-xl model. The migration point has been inserted at the end of the frame decoding pass because, at this point, no state information has to be transferred from HW to SW or vice-versa.

Two implementations of the JPEG decoder have been designed. The first one is quality factor and run-length encoding specific (referred as specific hereafter), meaning that the quantization tables and the Huffman tables are fixed, while the second one can accept any of these tables (referred as general hereafter). Both implementations target the 4:2:0 sampling ratio.

The results of the implementation of the decoders in hardware are 9570 LUTs for the specific implementation and 15901 LUTs for the general one. (These results are given by the report file from the Synplicity Synplify Pro advanced FPGA synthesis tool, targeting the Virtex2 XC2V6000 device, speed grade -4, and for a required clock frequency of 40 MHz).

The frame rate of the decode thread is 6 frames per second (fps) for the software implementation (executing on the StrongARM) and 23 fps for the hardware implementation. These results are the same for both general and specific implementation. The FPGA clock runs at 40 MHz, which is the maximum frequency that can be used.

When achieving 6 fps in software, the CPU load is about 95%. Moving the task to hardware reduces the computational load of the CPU, but increases the load generated by the communication. Indeed, the communication between the send thread and the decode thread on the one side, and between the decode thread and the receive thread on the other side, is heavily loading the StrongARM.

The communication between the iPAQ and the FPGA is performed using internal dual port BlockRAMs (DPRAMs) of the Xilinx Virtex FPGA. While the DPRAM can be accessed at about 20 MHz, the CPU memory access clock runs at 103 MHz. Since the CPU is using a synchronous RAM scheme to access these DPRAMs, wait-states have to be inserted. During these wait-states, the CPU is prevented from doing anything else, which increases the CPU load. Therefore, the hardware performance is mainly limited by the speed of the CPU-FPGA interface. This results in the fact that for a performance of 23 fps in hardware, the CPU is also at 95% load.

Although the run-time manager overhead for migrating the decoder from software to hardware is only about 100μs the total migration latency is about 108ms. The low run-time manager overhead can be explained by the absence of a complex task placement algorithm due to using a pre-partitioned FPGA model. Most of the migration latency is caused by the actual partial reconfiguration through the (slow) CPU-FPGA interface. In theory, the total software to hardware migration latency can be reduced to about 11ms, when performing the partial reconfiguration at full speed. When moving a task from hardware to software, the total migration latency is equal to the run-time manager overhead, since, in this case, no partial reconfiguration is required.

4.4 Handling Message Consistency in a NoC

Whenever a task is migrated from source tile to destination tile there might be some unprocessed messages in the tile input buffers or in the communication path between the migrating task and its producer tiles. Ideally, these messages have to arrive at the destination tile without requiring extra on-chip memory or functionality like e.g. a message re-ordering block. Furthermore, one has to take into account that dropping and re-transmitting messages requires extra work from the designer or requires a more complex on-chip communication protocol.

So the unprocessed messages have to be forwarded in the same order as they would arrive on the source tile of the migrating task. Furthermore, it is easy to imagine that the destination tile is closer (i.e. smaller hop-distance) to one or more producer tiles. This means that one has to make sure that newly produced messages do not arrive before the forwarded messages.

This section presents two task migration mechanism that consider message consistency while considering the limitations of the Network-on-Chip environment.

4.4.1 Generic Task Migration Mechanism

The different steps that need to be performed by the general NoC migration mechanism to actually migrate a task are described in detail by Figure 4.8.

When the run-time manager sends a migration signal to the source tile (1), the task running on that tile may be in a state that requires more input data (i.e. more messages) before it is able to reach the next migration point. This input data originates from other tasks further denoted as *producer tasks* instantiated on so-called producer tiles. Neither the run-time manager, nor the producer tasks know how many input messages are still required for the task on the source tile to reach a migration point.

When the task on the source tile finally reaches a migration point, it signals this event to the run-time manager (1 to 2). In turn, the run-time manager instructs the producer tasks to send one last *tagged message* to the source tile and then stop sending further messages (2). The run-time manager then sets up, initializes and starts the migrating task on the destination tile (3).

The next step is to forward all buffered and unprocessed messages to the new location of the migrated task. To this end, the run-time manager initializes a new message destination lookup table[4] (the so-called *forward-DLT*) on the source tile and instructs to orderly forward all incoming messages (4) (Figure 4.9).

[4] As Section 4.3.1 explains, a Destination Lookup Table or DLT is a translation table that helps the local run-time manager to resolve the destination tile of the messages. This avoids having to resolve the location of a communication peer every time a message is sent. A forward-DLT instructs the local run-time manager to forward all incoming messages to a specific tile.

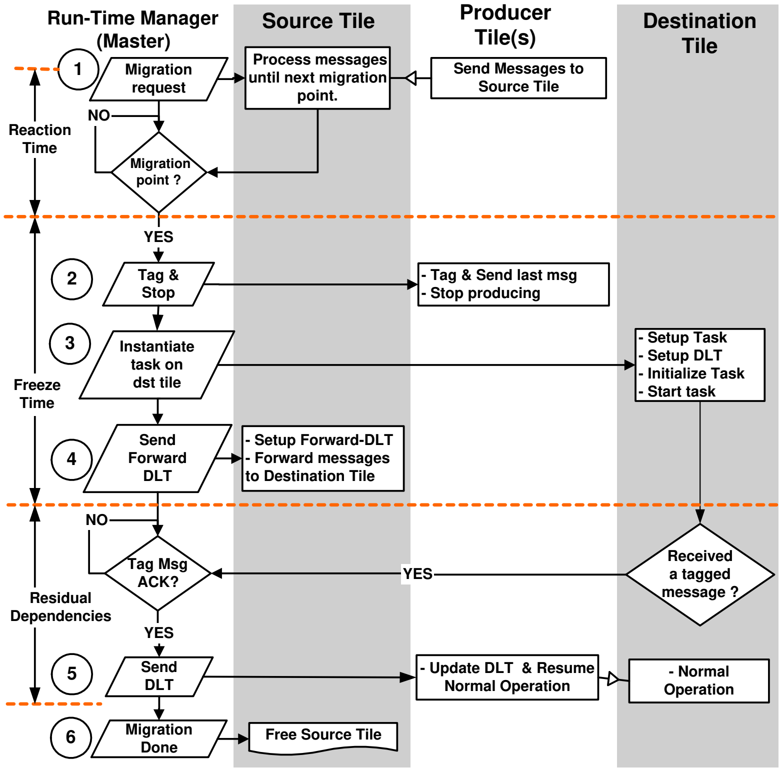


Figure 4.8: Flow of the generic task migration mechanism

The destination tile informs the run-time manager whenever it receives a tagged message. In that event, the run-time manager updates the DLT of the source tile to reflect the new location of the migrated task and the producer tiles can resume sending messages (5).

The arrival of all tagged messages in the destination tile indicates the end of the migration process. Hence the run-time manager can free the origin tile (6).

Here, the application designer is responsible for introducing suitable migration points into the application tasks. In that respect, the designer should find an optimal balance between minimizing reaction time and limiting the amount of task state that needs to be transferred.

This mechanism can also be classified as a *receiver-initiated*. Meaning that the receiver of the migration request is responsible for determining a suitable migration point.

4.4.2 Pipeline Migration Mechanism

The pipeline migration mechanism is based on two basic assumptions.

The first assumption is that multimedia algorithms are often pipelined (e.g. MP3 decoding, image/video decompression, etc.). Meaning that they are composed of com-

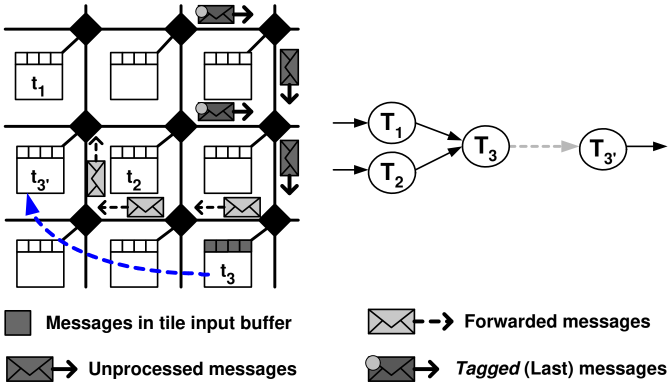
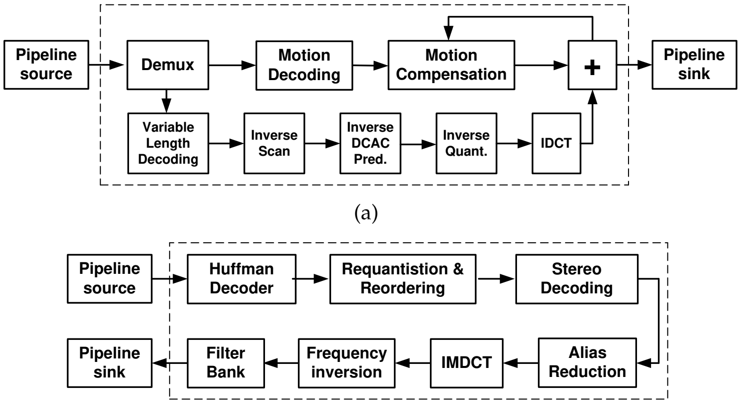


Figure 4.9: Forwarding buffered and unprocessed message to the destination tile. All last messages coming from the producer tiles are tagged.

municating tasks organized in a pipeline fashion. Consequently, different pipeline components are assigned by the run-time manager to different processing elements depending on e.g. their computational requirements. Figure 4.10a illustrates the MPEG-4 simple-profile decoding pipeline [59], while Figure 4.10b details the MP3 decoding pipeline [40].



(a)

(b)

Figure 4.10: Pipelined multimedia algorithms: (a) MPEG-4 simple profile decoding pipeline [59] and (b) MP3 audio decoding pipeline [40].

The second assumption is that most of these multimedia algorithms have stateless points. This means that, at certain points in time, a producer task puts new and independent information into an application processing pipeline. This producer task is also denoted as the *pipeline source* task.

Example 4.2: Pipeline migration
Consider an MPEG stream is composed of I, B and P frames (Figure 4.11). The P-frames depend on the previous I-frame, while the B-frames depend on the enclosing I-frames and P-frames. Periodically, the pipeline receives a new I-frame to decode. This I-frame does not depend, in any way, on previously processed information. Hence, this I-frame and the following frames could be decoded by a newly instantiated MPEG decoding pipeline. Similarly, an MP3 stream is composed of frames, each containing a header and audio data.

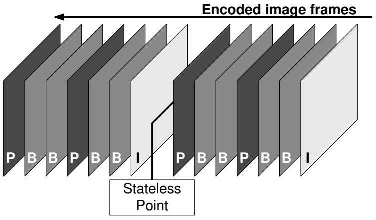


Figure 4.11: Organization of I, B and P frames in an MPEG stream. Both groups (before and after the stateless point) could be processed by a separate decoding pipeline.

Based on these two assumptions, a migration mechanism with the ability to move an entire pipeline at once can be created. In contrast to the generic task migration mechanism, pipeline migration is *sender-initiated*. Meaning that a producer task is responsible for determining a suitable migration point for one or more tasks in the pipeline. The different steps that need to be performed by such a mechanism are detailed by Figure 4.12.

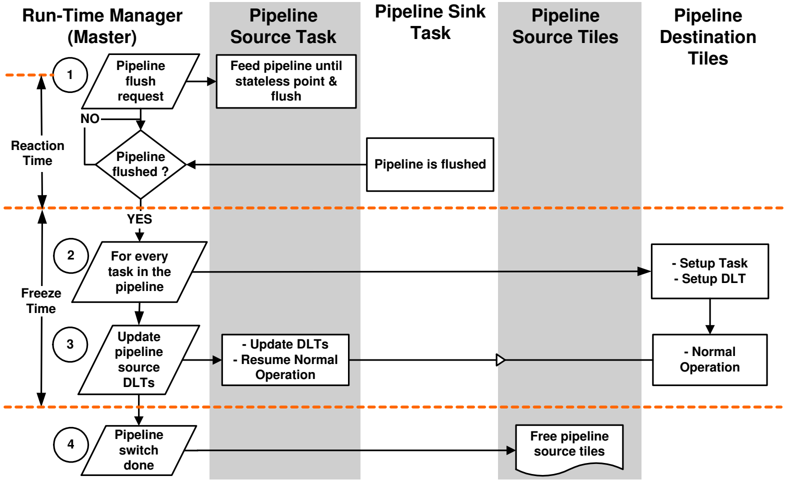


Figure 4.12: Flow of the pipeline migration mechanism.

Whenever the run-time manager wants to move one or more pipelined tasks, it instructs the *pipeline source task* (Figure 4.10), to continue feeding data into the pipeline until a stateless point is reached.

At that point, the pipeline source task should issue a pipeline flush by sending a special message either through the pipeline or directly to the pipeline sink task. As soon as the pipeline is flushed, the pipeline sink task notifies the run-time manager (1 to 2).

In contrast to the generic task migration mechanism, there are no unprocessed or buffered messages in the path between pipeline source and pipeline sink. At this time, the run-time manager can re-instantiate every task of the pipeline on a different location (2). This includes updating the DLTs of every new task. The only thing that remains before resuming normal operation is to update the DLT of the pipeline source task in order to reflect the new location of the first task in the (new) pipeline (3). Finally, the run-time manager frees the resources occupied by the original pipeline.

In this case, the application designer is responsible for enabling the pipeline migration by introducing knowledge about the stateless application point into the pipeline source task.

4.4.3 Migration Mechanism Analysis

This section analyzes the performance of both mechanisms. In this context, we assume that the communication link between the local run-time manager(s) and the master run-time manager provides a guaranteed, real-time communication service. The key issue for every mechanism is being able to determine the worst-case total migration time for migrating one or more tasks. This section compares both mechanisms in that respect.

Generic Migration Mechanism

The *reaction time* $T_{i,react}$ for migrating a single task using the generic migration mechanism is dependent on the task implementation, i.e. the positioning on the migration points. Obviously, the position of the migration points will be strongly influenced by the relevant task state at these (potential) points.

The *freeze time* $T_{i,freeze}$ entails (1) sending the *tag & stop* command from the master run-time manager to the local run-time manager of every involved communication peer, (2) instantiating the migrating task on the destination tile and (3) forwarding the messages after setting-up a forward-DLT.

As this is a distributed environment, the time needed to send a *tag & stop* message to multiple peers can typically overlap. In the worst case, it amounts to the time needed for one *tag & stop* message $T_{i,t\&s}$ multiplied by the number of communication peers C_i for the migrating task t_i.

The time $T_{i,create}$ needed to instantiate the new task $T_{i,create}$ the new task is the sum of (1) the time needed for transferring the task binary $T_{i,setup}$, (2) the time needed for initializing the task with task state $T_{i,init}$ and, finally, the time needed for setting up the task DLT T_{dlt}. The task setup time $T_{i,setup}$ is PE and task dependent. For example, setting up a task on a FPGA fabric tile typically requires more time than setting up the same task functionality on an instruction set processor. Again, as we are in a distributed environment, $T_{i,create}$ could overlap with sending the *tag & stop* messages.

The time T_{fdlt} needed for setting up a forward-dlt and instructing the migrating task T_i to orderly forward unprocessed data is task independent. At this point, the unprocessed messages can be forwarded and the application can restart.

Finally, the time needed to clear the *residual dependencies* is composed of the time needed to forward the unprocessed data and to reset the DLT of the communication peers of the migrated task. However, the time needed for all messages to be forwarded T_{fmsg} is dependent on (1) the allocated bandwidth or (in case of a best effort service) the available bandwidth and (2) the amount of unprocessed messages that need to be transferred.

The total (worst case) time required to migrate a *single task* is given by Equation 4.2.

$$T_{i,react} + \overbrace{(T_{t\&s} \times C_i) + (T_{i,setup} + T_{i,init} + T_{dlt}) + T_{fdlt}}^{T_{i,freeze}} + T_{fmsg} + (T_{dlt} \times C_i) \quad (4.2)$$

As a consequence, making an accurate estimate about the total migration time requires platform data communication service guarantees. This involves guarantees with respect to communication flow control, on the one hand, and communication bandwidth and latency, on the other hand. These guarantees will enable estimating an upper bound of T_{fmsg}. The time needed by the run-time manager for setting up a guaranteed communication service [131] could be considered part of T_{fdlt}.

The *Gecko*^{2} demonstrator, detailed in Chapter 6, does not provide *hard* guarantees with respect to communication bandwidth nor with respect to flow control. This means that reasonable migration time estimates can only be given in case of an uncongested network (see Equation 3.2 on page 52).

Similarly, the absence of hard end-to-end flow control requires precautions with respect to the generic task migration mechanism. Consider the situation depicted by Figure 4.13. Migrating task T_C to tile 6 will require sending a *tag & stop* command to task T_B. However, as there is no platform flow control mechanism between T_A and T_B, the link between tile 0 and tile 3 might be congested. This will, at least, seriously delay or stall the flow of unprocessed messages that are forwarded between T_C and $T_{C'}$, which will either prolong the migration or cause a deadlock situation. In this case, the run-time manager has to consider these situations and avoid that the forwarded stream aligns with a blocked stream or, in general, a congested part of the NoC.

A way to optimize the generic migration mechanism, is to let the local run-time manager of the migrating task handle the entire mechanism instead of reporting back to the master run-time manager. This would mean that the master run-time manager not only indicates to the local run-time manager that a task needs to migrate, but also indicates what communication peers to stop and the new location (through the forward-DLT) of the task. The local run-time manager of the newly instantiated

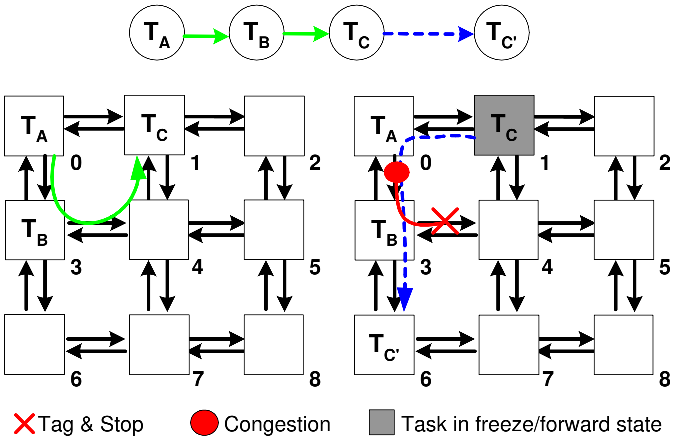


Figure 4.13: *Using the generic task migration mechanism requires hard platform communication services in order to avoid e.g. deadlocks. In this case, a link is blocked due to a* tag & stop. *Unfortunately, this blocking also affects the forwarding of messages resulting in a deadlock.*

task would then inform its communication peers to resume sending messages after receiving all tagged messages. In essence, this would result in a more distributed algorithm, imposing less load on the master run-time manager.

Pipeline Migration Mechanism

The *reaction time* $T_{app,react}$ for migrating an application pipeline is dependent on the application characteristics and, potentially, dependent on the data it processes. For example, the I-frame frequency in a MPEG stream can vary. The pipeline flush time T_{flush} depends on the size of the pipeline and the platform communication service guarantees.

In the worst case, the *freeze time* is equal to the time required for setting up the application. This involves the time needed to instantiate $T_{i,create}$ every task t_i of the migrating pipeline. This involves the time needed for transferring the task binary $T_{i,setup}$ and the time required for setting up the task DLT T_{dlt}. Again, the task setup time $T_{i,setup}$ is PE and task dependent. In contrast to the generic task migration mechanism, a pipeline task does not need to be initialized (i.e. no $T_{i,init}$) with a previously saved state. Finally, after setting up all the pipeline tasks, the DLT of the pipeline source needs to be updated. There are no residual dependencies when using the pipeline migration mechanism.

The total (worst case) time required to migrate a pipeline of N tasks is given by Equation 4.3.

$$T_{app,react} + T_{flush} + \overbrace{\sum_{i=0}^{N}(T_{i,setup} + T_{dlt}) + T_{dlt}}^{T_{app,freeze}} \tag{4.3}$$

In some circumstances, optimizations can be performed. First, in case flushing the entire pipeline takes a long time, the run-time manager could already remove a task (i.e. free platform resources) whenever the tagged message has passed a certain task. Secondly, in case enough platform resources are available, the run-time manager could instantiate a new pipeline during the *reaction time*. This would effectively hide the application *freeze time* $T_{app,freeze}$ by the application *reaction time* $T_{app,react}$. When $T_{app,freeze} < T_{app,react}$, the only thing the run-time manager still needs to do when reaching the application migration point is resetting the DLT of the pipeline source task (i.e. requiring T_{dlt} time).

Migration Mechanism Comparison

The main difference between both mechanisms is that the generic mechanism is developed for migrating a single task, while the pipeline mechanism assumes that a complete pipeline will be moved at once. With respect to the migration policies defined in Section 4.2, the generic migration mechanism would be preferred when using *migration after mapping failure*, while the pipeline migration mechanism should be more appropriate for *migration after co-assignment*.

When migrating a complete pipeline, the generic migration mechanism has to find a migration point for every single task instead of using a application generic migration point. Although this surely is an advantage for migrating a single task, the global reaction time of all pipeline tasks could outgrow the application reaction time. Especially if there exists a task t_i where the $T_{i,react}$ is equal to $T_{app,react}$.

With respect to freeze time, the generic migration mechanism has to reset the same DLT multiple times. Consider the example pipeline of Figure 4.14. When moving task T_B, a DLT will be initialized on its new location. The same holds for T_C. However, when migrating T_D, the DLT of both T_B and T_C will have to be altered to reflect the new location of T_D.

Furthermore, forwarding data is in fact a disturbing with respect to the rest of the system. When forwarding is done using a guaranteed communication service, this service needs to be negotiated and set up (i.e. the guaranteed communication channel needs to be established. This can include path finding and/or allocating communication bandwidth). Experiments [131] have shown that creating such a communication service is quite costly for a short-lived communication channel. Using a best-effort communication service for forwarding the unprocessed data does not require setting up such a service, but it is impossible to guarantee real-time deadlines. These effects accumulate when moving an entire pipeline in a task by task manner. These issues do not arise in case of using the pipeline migration mechanism as it does not require message forwarding.

Finally, when using the pipeline migration mechanism, task state does not need to be transferred and tasks do not need to be initialized. This means that when enough platform resources are available, a new pipeline could be created while waiting for the application to reach its migration point (i.e. during $T_{app,react}$).

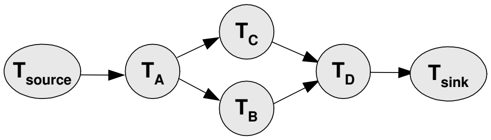


Figure 4.14: *Example task graph containing a pipeline of tasks to be migrated.*

4.5 Low Cost Task Migration Initiation

One of the main issues in heterogeneous task migration is the overhead incurred by checking for a pending migration request during normal execution (i.e. when there is *no* pending migration request).

Especially since a task requires to frequently pass a migration point in order to reduce the *reaction time*. The reaction time is the time elapsed between selecting a task for migration and the selected task reaching the next migration point (Figure 4.7b). Adding more migration points thus reduces the reaction time, but increases the checking overhead.

4.5.1 Migration Initiation Taxonomy

As Section 4.6.4 details, there are currently two main techniques to check for a pending migration request:

- **Polling.** In this case, *polling points* are introduced into the execution code (into the source code by the programmer or into the object code by the compiler), where the task has a migration point. This technique is completely machine-independent, since the architectural differences will be taken care of by the compiler in one way or another. However, this technique potentially introduces a substantial performance cost during normal execution due to the continuous polling.

- **Dynamic code modification.** Here the execution code is altered at run-time to introduce the migration-initiation code upon a migration request. This way, these techniques can avoid the polling overhead. These techniques have their own downsides. Besides the fact that changing the code at run-time will most likely require a flush of the instruction cache, changing an instruction sequence the processor is currently executing can result in non-deterministic effects.

In order to minimize the checking overhead during normal execution, further denoted as *migration initiation*, we propose a novel technique [155] targeted at embedded systems, that uses the debug registers present on most modern PEs as *migration initiation registers*.

4.5.2 Hardware Supported Migration Initiation

Most contemporary microprocessors (PowerPC, ARM, i386, etc.) contain a set of debug registers. The PowerPC 405, present in the Xilinx VirtexII-Pro FPGA, contains four 32-bit *Instruction Address Compare* (IAC) registers.

Whenever the program counter (PC) register reaches a value (i.e. a certain instruction) present in one of the activated IAC registers, an exception is generated. In normal conditions, this exception is caught and handled by the debugger.

However, this mechanism could also be used as a *poll-free migration* initiation technique that does not require any copying or insertion of code to enable migration points. The setup of our proof-of-concept implementation is illustrated by Figure 4.15.

After starting the task (1), a migration handler is registered with the local run-time manager (2). This handler will be responsible for collecting the logical task state after the task reaches a migration point. In addition, all migration point addresses (denoted as *mp*) are registered with the local run-time manager (3). Then, the task starts executing. In the absence of a migration request, there is no run-time overhead (4).

The run-time manager maintains the migration point addresses in a task-specific data structure. With every (potential) context switch to another task the local run-time manager updates the IAC registers. When the resource manager decides to migrate a task, it activates the IAC registers (i.e. simply setting some register flags) (5)(6).

As soon as the task reaches the instruction on a migration point address (denotes as *mp*), a hardware interrupt is generated (7), which activates the migration signal handler (8). After gathering the logical task state, the task is ready for migration to the destination PE.

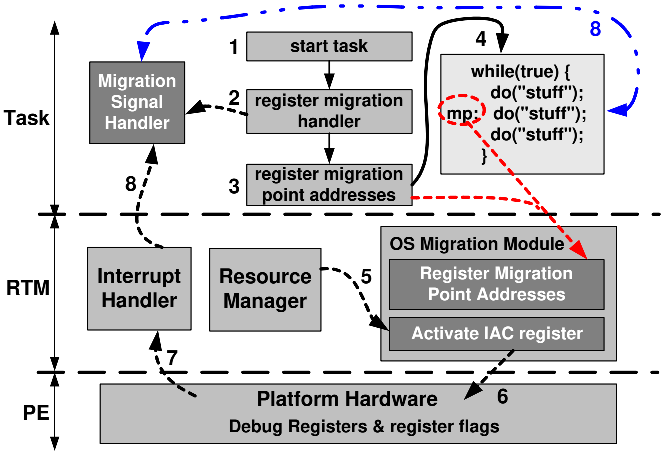


Figure 4.15: *Migrating initiation by reusing the microprocessor's debug registers.*

4.6 Related Work

When it comes to run-time task migration in a heterogeneous environment, there is also quite some related work mostly coming from the parallel & distributed systems domain. This section provides an overview of different techniques with respect to task migration policies (Section 4.6.1) managing task state in a heterogeneous system (Section 4.6.2), maintaining message consistency (Section 4.6.3) and, finally, migration initiation (Section 4.6.4).

4.6.1 Task Migration Policies

The task migration policy is responsible for determining if a task migration is needed and, if so, for selecting the task-to-be-migrated and the destination processor.

Bishop et al. [24] detail a single process migration policy for performing load balancing in heterogeneous multicomputer systems. They define a graph model for calculating the costs associated with migrating a process. Each vertex of the graph represents a computing host, while each directed edge represents the inter-host connectivity. Edges and vertices have associated weights. Vertex weights represent the execution cost per time unit of a particular host. Edge weights represent the costs of moving a process and are calculated by taking the several costs into account: transfer negotiation cost, transfer of state cost, the cost of the state translation and the cost of (possible) residual dependencies. Vertex weights change when the system evaluates the load and selects the candidate migration processes. A good time to evaluate is, for example, at task creation or when the task moves to a different phase with new resource (QoS) requirements.

Within the context of an MPSoC platform or a *Chip Multi-Processor* (CMP)[5], task migration is still a fairly young subject. Just like for large-scale multiprocessor and multicomputer systems, most authors employ task migration as a load balancing technique.

Bertozzi et al. [23] describe an on-chip task migration mechanism to perform load balancing on a homogeneous multiprocessor platform. In their setup, every processing element (ARM) has a local private memory containing the task state (stack, heap, open I/O resources). Inter-processor communication is implemented by means of a shared memory. All system components are interconnected by means of a shared bus. It is clear that, in this way, message consistency is not an issue. Despite the homogeneous environment, the authors base their technique on designer-specified migration points similar to our approach. Their main motivation is to only migrate a task when the task state information is minimal. This avoids having to copy too much state information between the private memory of source and destination PE. The authors determine the break-even point between leaving a task in an unbalanced situation (i.e. getting only limited access to CPU time) and migrating with respect to the amount of task state information that needs to be transferred and the frequency of the migration points. They show that the size of the task state has a

[5] The CMP community is slightly different than the MPSoC community. The CMP community is more focused on general purpose computing and is, as a result, less interested in the relation between multiprocessor platform and application or application domain.

non-negligible effect on the task freeze-time. They propose to have a migration policy that is tailored to the migration mechanism and the task freeze time. However, their proof-of-concept example application based on DES encryption uses a stateless migration point, since moving it otherwise does not make sense: a stateless migration point is visited often, while migrating otherwise would incur a significant task state (i.e. large task freeze time).

Kumar et al. [119] state that task migration can reduce energy dissipation and keep die temperature under control. They show, by means of a simple example, that by combining *Dynamic Voltage and Frequency Scaling* (DVFS) with task migration, it is possible to increase the efficiency of the dynamic power management by 25%. However, as the authors acknowledge, the inter-task communication is not taken into account.

Shaw et al. [200] describe a task migration based load balancing policy for homogeneous CMPs organized in a mesh. The actual inter-tile communication infrastructure is abstracted. The operating system gathers run-time information about inter-tile communication and about the load of the tiles. The migration policy is based on a combination of *forces*. Attraction forces pull interacting data and tasks closer together in order to improve locality, i.e. to reduce the hopbandwidth. Repulsion forces push threads away from highly loaded tiles. The resulting combined force migrates tasks to simultaneously reduce hopbandwidth and distribute the resource load. The migration forces are recalculated either at fixed intervals or e.g. after a number of communication operations of a certain task. As the authors rely on run-time information gathering, they make a distinction between tasks with a clear communication pattern and tasks with an unclear communication patterns. They conclude that task migration improves the application performance in case of clear communication patterns. Furthermore, the locality improvements and distribution of resource load reduces the contention of the on-chip communication network. This approach obviously only works in a homogeneous environment where message consistency can be ignored. We also consider the hopbandwidth and the load distribution in our migration policy. However, our policy is not triggered by a platform load unbalance.

Ozturk et al. [166] consider selective code (i.e. the task) and/or data migration in the MPSoC environment in order to reduce energy consumption. The rationale of their approach is to either move the code to the processing element where the data resides or to transfer the data to the processor where the executing task resides. In addition , the authors take into account where the resulting data (i.e. the output data of the task) should be located. The proposed method involves analyzing the task graph of a given parallel application and performing application profiling at design-time. Consequently, the application code is annotated with statements describing the correlation between the code and the current and future data that it requires. At run-time, the migration policy algorithm decides (on an abstract level) whether to migrate the task or the data. Unfortunately, the authors have abstracted away the architectural constraints that have an impact on migration cost. In addition, the authors do not specify any mechanism to go with their policy.

Pittau et al. [180] focuses on task migration for homogeneous multiprocessor platforms with a single non-cacheable shared memory. This shared memory is used for inter-task communication by means of message passing. Similar to our approach, the authors only allow tasks to migrate at designer-provided migration points. They propose two task migration mechanisms. The first mechanism, denoted *task recreation*, deletes the task on the source processing element and recreates it on the destination processing element. In the second mechanism, denoted *task replication*, a replica of every task is present on every local processing element. This second approach is introduced for deeply embedded platforms with run-time managers that are not capable of dynamically creating new tasks. The authors avoid the message consistency problem by only considering a bus as on-chip interconnect. Although the authors do not link the provided mechanisms to an actual task migration policy, they list the task migration options by means of a set of Pareto optimal configurations. In the end, this Pareto information could be used by a migration policy to limit its search space.

4.6.2 Managing Task State in a Heterogeneous System

When it comes to managing task state with respect to HW/SW task migration, one can distinguish two kinds of related work. First, we will briefly detail ways to retrieve and restore FPGA fabric state. Secondly, we will have a look at how task state is retrieved and translated in conventional heterogeneous parallel & distributed systems.

The most widely adopted way [107, 122, 203] to handle FPGA task state extraction (and consequent restoration) is to read back the bitstream representing the task. Consequently, all status bits are retrieved or filtered out of this bitstream. Restoring state is done by manipulating the original (empty state) configuration bitstream of a task to include the previously retrieved state. Adopting this methodology to enable heterogeneous task migration would require a translation layer in the run-time manager, allowing it to translate an ISP type task state into FPGA task state bits and vice versa. In order for this approach to succeed in general, one would also require some unified task description like the OCAPI-xl task description. Furthermore, this technique is very FPGA technology dependent since the exact position of all the configuration bits in the bitstream must be known. It is clear that this kind of approach does not produce a universally applicable solution for saving and restoring HW/SW task state.

Another technology independent idea [107] is to automatically add additional hardware structures that can read/write to all relevant task state registers. In order to reduce the amount of registers that need to be handled, one could imagine having a *shutdown process* that reduces the amount of relevant state information. This approach bears some similarity with our technique. First, it is also (FPGA) technology independent since the additional hardware structures are part of the design description. Secondly, the proposed shutdown process resembles our task interrupted state. However, the extra hardware structures and the task shutdown process both require additional hardware and design effort.

Attardi et al. [16] show that two distinct migration mechanism classes can be identified: the *interpretation* and the *translation* mechanism. The interpretation-based mechanism requires running the same interpreter on both the source and the target computing resource. This approach effectively reduces the problem to homogeneous migration since the same universal machine is emulated on both computing resources. In case of a translation-based mechanism, the task or process to be migrated is kept and executed in a machine-dependent form. Before migration the task/process state needs to be translated into a machine-independent form. After transferring the captured state to the destination machine, it needs to be translated into the local format in order to continue task/process execution. When the run-time manager supports hierarchical configuration (Chapter 3), it is possible to migrate a task from one FPGA tile to another FPGA tile in an interpretation-based way, i.e. by re-instantiating the same softcore.

Quite some authors have addressed the issue of tool-assisted code modification to e.g. insert migration points or to keep track of task state.

The Tui Heterogeneous Process Migration System, described by Smith et al. [179], is able to migrate type-safe ANSI-C programs between a set of heterogeneous ISP architectures. These ANSI-C programs need to be compiled for every single target architecture with a modified version of the *Amsterdam Compiler Kit* (ACK).

A different technique based on compile-time transformations combined with a run-time library (Ythreads library) is described by Sang et al. [101]. This technique allows migrating threads (i.e. lightweight processes) across computing resource boundaries. Again the issue of creating machine-independent task state information in order to allow heterogeneous migration needs to be addressed. In this case, the C programming language was extended with a *thread construct* in order to create the logical state of a thread.

The heterogeneous thread migration mechanism, described by Jiang et al. [90], is also created by combining compile-time and run-time support. At compile-time, the pre-processor scans the application, automatically adds migration points and inserts suitable thread migration primitives.

4.6.3 Ensuring Message Consistency

The second issue when dealing with heterogeneous task migration in a network environment is assuring inter-task communication consistency during the migration process. The main problem is that it is generally not known in advance when a certain task will reach its migration point. This means that when the task actually reaches its migration point, there might be a number of messages buffered in the communication path between other (sending) tasks and the migrating task. Obviously, these unprocessed messages need to be transferred or redirected to the new location of the migrating task.

Russ et al. [190] describe a dynamic communication switching mechanism. This mechanism not only ensures communication consistency during task migration, it also allows to switch between different types of communication (e.g. from a message passing model to a shared memory model) at run-time. The mechanism can be summarized as follows. After receiving a migration signal, the task sends out an end-of-channel message to all its communication peers. As an acknowledgment, the communication peers in turn reply with an end-of-channel message. These special messages serve two functions. First of all, they mark the end of communication between the migrating task and its peers. In addition they force all the messages still stored in the kernel buffers to be copied to an unexpected message queue. Once all end-of-channel messages have been received, the task is able to migrate (together with the unexpected message queue). After migration, all communication peers are notified, while their task table (sort of destination lookup table) is updated. However, the migrated task can resume execution even before all communication is restored. Communication consistency is preserved by emptying the unexpected message queue before receiving any messages send after completion of the migration process.

A similar technique to preserve communication consistency is described in [210,211]. The migrating process sends a special synchronization message to the other processes of the application. In turn, these processes send a ready message to each other. After receiving such a message from all other processes, it is safe to migrate. Messages that arrive before the last ready message has received are buffered. After the migrated process has been restarted, it is served with the buffered messages first. CoCheck, described by [211], is a migration environment that uses this technique and that resides on top of the message-passing library.

Both mechanisms are not applicable in a NoC environment. Due to the extremely limited amount of message tile buffer space it is impossible to store all incoming messages after a task reached its migration point. This implies that messages might remain buffered in the communication path. Adding more buffer space to accommodate these messages is not an option, because on-chip memory is expensive and the maximum amount of required storage is highly dependent on the application and its run-time mapping.

The Amoeba distributed operating system offers a different way of dealing with the task migration communication consistency issue [209]. Instead of queuing the incoming messages during the task freeze time, they are rejected, while the message sender receives a *process is migrating* response. Then it is up to the message sender to retry after a suitable delay. After migration, any task that sends a message to the previous location of the migrated task will receive a *not here* reply. This response triggers a mechanism to find the new location of the destination task. In contrast to the previously described techniques, this technique does not require buffer space to queue the incoming messages, which for example avoids a certain memory penalty in case of an upfront-unknown amount of messages. The drawback of this technique is that the migration mechanism loses some transparency with respect to the communication peers. Meaning that it is up to the application, either up to the communication protocol transport layer to retransmit messages.

From a Network-on-Chip point of view, this technique is also not suited. Dropping and re-transmitting messages reduces network performance [87] and increases power dissipation [57]. To ensure reliable communication in a task-transparent way, this technique requires (costly) additional on-chip functionality [12]. Furthermore, dropping messages will lead to out-of-order message delivery. Special message reorder functionality combined with a large amount of extra buffer space is needed to get messages back in-order in a task-transparent way.

Rutten et al. [191] face a similar problem: they wish to (re)configure different tasks of an application pipeline without causing data inconsistency. To this end, they reconfigure the application tasks at a specific point in the streaming data (e.g. before an I-frame). This is achieved by inserting a special *location ID* packet into the stream. This packet passes through the pipeline and triggers its receiving tasks to get reconfigured by a *control processor*. This approach resembles the packet flushing of the pipeline migration mechanism. In addition, the Eclipse hardware shell (see page 40), acting as a run-time library between the control processor (i.e. master) and the application task, provides support for this application reconfiguration mechanism.

4.6.4 Task Migration Initiation

Two distinctive migration initiation approaches can be identified.

The first approach is based on polling [8, 89], i.e. introducing statements that check whether a migration is requested.

It introduces initiation polling points into the execution code at the location of the migration point. The amount of poll points and their placement in the code is critical for performance: too many poll points introduce a large run-time overhead, while not enough poll points increase the reaction time. The amount of work required for the OS to enable the migration request can be as little as setting a global variable.

The second approach [179, 183] is based on dynamic modification of code. In this case, the execution code is altered at run-time to introduce the migration-initiation code after receiving a migration request.

The Tui System [179] stops the concerning task and places a breakpoint instruction at every migration point. Then the task continues until a breakpoint trap occurs. In order to avoid overwriting other instructions when inserting these breakpoint instructions, extra space needs to be reserved. This can be done using dummy instructions, which introduce some performance overhead during normal execution.

Prashanth et al. [183] introduce a technique that detects the fragment of code containing the next migration point and places migration initiation instructions in a copy of that code. Hence, this technique does not require any placeholder instructions. The amount of work to enable a migration point (for the second approach) is quite large in contrast to the first approach, which prolongs the reaction time.

4.7 Conclusion

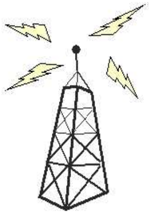
This chapter introduces the concept of run-time task migration in the MPSoC context. The benefits of having run-time task migration capabilities are briefly discussed and the issues that arise are explained. These issues entail: providing a suitable run-time *task migration policy*, taking care of the *task state* when migrating between heterogeneous PEs, ensuring NoC *message consistency* during task migration and having a low-overhead *migration initiation mechanism*. Consequently, the presented issues are tackled.

First, we detail two novel run-time task migration policies linked to the run-time task assignment heuristic. The choice of the migration heuristic depends on the situation (i.e. how many tasks need to be migrated) and on the platform properties at migration time. We show that adding task migration capabilities to the run-time assignment heuristic clearly improves the assignment success. For low application load, the application assignment failure rate drops by up to 34% using the *migration after mapping failure* extension, while for high application load the failure rate drops by up to 14% using the *migration after co-assignment* extension. In most cases, the success rate of a migration-enabled task assignment heuristic exceeds searching the full solution space without considering migration, while using only a fraction of the execution time.

Second, we detail a novel method of handling task migration between an ISP and a FPGA fabric tile [144]. This method uses an application-specific task state instead of translating the state from source PE type to the destination PE type. We show that the run-time environment needs to be adapted for handling HW/SW task migration. A MJPEG video HW/SW migration case study shows how this technique can be used by the run-time manager to speed up execution of a specific task at a specific moment in time.

Third, we detail two novel task migration mechanisms adapted to the NoC environment [157]. Both mechanisms ensure low-cost message consistency during task migration. The first mechanism is a *generic task migration mechanism*, while the second mechanism, denoted as *pipeline migration mechanism*, makes certain assumptions concerning the target applications.

Fourth, we detail a novel poll-free task migration initiation technique [155]. The presented mechanism re-uses the debug registers, commonly found on contemporary embedded PEs, to initiate the migration.



CHAPTER 5

Reactive Communication Management of a Network-on-Chip

Networks-on-Chip are reconfigurable and thus bring flexibility to programmable MPSoC platforms. As a consequence, several applications can be run concurrently on the platform, raising the problem of application composability. Indeed, as the NoC is a shared medium, how are communications of different applications impacting one another? In fact, two questions are raised, the first one relates to the matching between application and platform resources and the second one to imperfect platform virtualization.

Because applications can be developed independently of the MPSoC platform (and possibly downloaded onto it), there can be potential mismatches between platform communication resources and their use by an application. There are several types of mismatches. For instance, applications can have been developed for another platform under the assumption no resource sharing would occur. Another cause of mismatches are misbehaving applications that generate more traffic than authorized, either because of a bug or intentionally (security threat). Can mismatches between expected and actual application communication behavior be detected at run-time? How can application communication be controlled at run-time? How can a run-time manager dynamically decide to give more priority to the communication of one (already executing) application over another (for instance giving priority to an audio decoding task over a file downloading task)?

A possible answer to the previous questions is to provide perfect virtualization of platform communication resources, so that applications execute independently of one another (virtual resources are private to an application). By characterizing a static application at design-time, one can often determine both the communication and computation requirements of every task with respect to the user requirements. This characterization information can be included in an application's task graph to be used at run-time in order to statically reserve sufficient computation and communication resources. This guarantees correct operation and expected application performance. Moreover a run-time manager can exploit the design-time application characterization and steer the available platform hardware components to achieve composability of several applications. However, the dynamism inherent to many applications makes them difficult to accurately analyze. In these situations the system has to operate with *estimations* of application requirements and must therefore dynamically adapt the platform resource usage to the needs, making perfect virtualization difficult to exploit. Another reason for dynamic adaptation of platform resource usage is the high cost of perfect platform virtualization support.

To either match application behavior to platform resources or to deal with partial (or no) virtualization of platform communication[1], the run-time manager needs to detect changes to its environment and reactively tune platform parameters to maintain the projected operating point. To create such a feedback loop, the run-time manager needs (hardware) support to monitor changes in the usage of platform resources, algorithms to react to unexpected changes and (hardware) actuators to act upon the environment.

This chapter, which is also part of the PhD thesis of Théodore Marescaux [132], discusses reactive communication control of a NoC steered globally by a run-time manager. Section 5.1 introduces the 3 components of reactive run-time NoC congestion management: *monitoring*, *decision-making* and *actuating*. The section furthermore discusses the differences of congestion-control and flow-control. Then, Section 5.2 describes the actuator used to perform communication traffic shaping. It also discusses the concepts, theoretical aspects and presents a proof-of-concept demonstration of traffic shaping on an emulated MPSoC platform. Section 5.3 presents the monitoring and decision-making options. We choose to implement a globally controlled communication management scheme, while Marescaux [132] deployed a distributed communication management scheme using the same platform monitors and actuators. Section 5.4 details the selected global connection-level management scheme. Finally, Section 5.5 presents the related work and Section 5.6 presents the conclusions.

5.1 NoC Communication Management Concepts

This section deals with important concepts of NoC communication management. It starts with a definition of the components of reactive communication-control mechanisms. We then discuss the differences between flow-control and of congestion-control.

[1] Reactive communication control is also relevant to NoCs with more virtualization capacity, such as a guaranteed throughput NoC, for instance to regulate the traffic of lower priority QoS classes. The interaction of reactive and proactive communication management systems is out of the scope of this thesis. It is an interesting and relevant topic for future work.

5.1.1 Components of Reactive Communication Control

Reactive communication control is a control mechanism so it can be decomposed into three key components (Figure 5.1):

1. *Monitoring,*
2. *Decision-Making,*
3. *Actuating.*

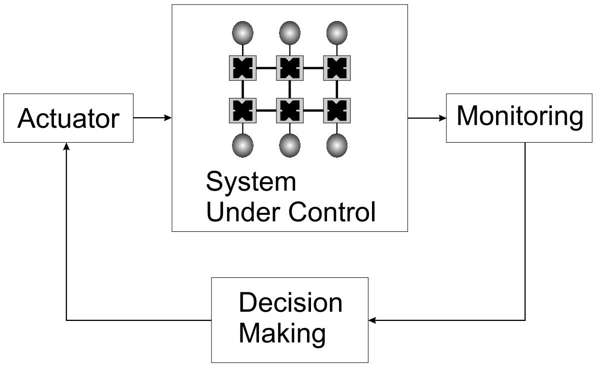


Figure 5.1: *Control Loop. A system under control is monitored (for instance a given network interface in a NoC). Statistics of the NI are passed to the decision-making module that given an algorithm and an objective function generates the inputs of the actuator. The actuator in turn modifies a parameter of the system under control (it shapes traffic for instance).*

As its name indicates, the *monitoring* mechanism measures NoC parameters at runtime and provides statistics over the communication. Monitoring of NoC communication can be performed at each layer of the NoC protocol stack. For instance, at the link layer, the link throughput can be measured at the individual local links. At the network layer, measures are local to a router and relevant parameters are average throughput, packet service time, buffer occupancy, etc. At the network interface layer (i.e. transport layer) we can measure end-to-end latency of packets, throughput, jitter, NI buffer occupancy, etc. Monitoring at the level of the network interface gives information about a connection and abstracts away details of the links and routers used by this connection. At the system level, statistics give latency (and jitter) of messages (or transactions) as well as data throughput.

To capture accurate statistics about the ongoing NoC communication, the monitoring mechanisms need to function at least at the speed of the considered NoC layer. Monitoring is thus often implemented in hardware.

Actuators are systems that, given an input control signal, act to regulate a particular parameter. In our case, the actuators regulate the communication properties of NoCs.

Differing by the manner in which they reduce the amount of data communicated per unit of time, actuators fall into two categories, they are either work-conserving or non-work-conserving. In presence of data to be communicated, a work-conserving actuator always outputs data albeit at a lower rate (for instance by serializing it). On the contrary, a non-work-conserving actuator, may decide to delay the sending of data currently available. So far, for reactive communication control on NoCs only non-work-conserving actuators have been used. There are at least two reasons to prefer them over work-conserving ones. On the one hand they do not require complex bit serialization/deserialization hardware and on the other hand non-work-conserving mechanisms allow to centrally schedule the traffic in the NoC (work-conserving actuators independently distort the traffic in the NoC making it much more complex for a predictable global schedule).

Naturally, as monitoring, actuators can be defined at each of the network layers. For instance, at the link layer an actuator can enforce the moment when a flit can be sent (or received to be buffered). At the network-layer, actuators can delay the service of given packets to regulate the traffic. At the network-interface layer, an actuator can control when a packet is to be injected into the network. Finally at the system-layer, actuators can decide when to issue a certain message or transaction. By delaying certain packets (or flits) by given amounts, the actuators shape the traffic at the various layers of the NoC.

The *decision-making* components have as inputs the statistics from the monitoring modules and produce as outputs the control signals to feed to the actuators. Given a certain objective and control algorithms, these components try to regulate NoCs by creating a feedback-loop (or control loop) between the actuators that shape the traffic and the monitoring mechanisms that report measured communication properties (Figure 5.1). Because monitoring mechanisms and actuators can be defined at all layers of the network stack, so can the decisions-making mechanisms.

Decision-making mechanisms can not only be implemented at one or more layers of the stack, but they can also decide to integrate monitoring information from several layers. Moreover, within a layer, decision-making mechanisms can be central (they integrate information from all monitoring modules in the layer), fully distributed (one decision-making system for every monitoring and actuator in the layer) or all possible hybrid combinations.

5.1.2 Flow-Control versus Congestion-Control

Flow control is defined by Tanenbaum as a point-to-point data-link or transport layer issue that deals with one data producer task outrunning a single data consumer task [218]. Flow control is employed to ensure that the producer (temporarily) does not send out more data than the consumer is able to receive and process. For an NoC flow-control can be defined at every layer of the network stack (Figure 5.2). For instance, at the data-link layer, handshaking controls the exchange of flits between the two end points of a link. Flow-control is used here for instance to avoid buffer overflow at the receiving router and/or to ensure integrity of the transmitted flit. Another example of flow-control at the data-link layer is virtual channel flow-control, or the flit-level interleaving of packets from different sources onto the same physical link [178]. At the network-layer the flow-control is in charge of scheduling packets for output as a function of several parameters such as: presence of backpressure[2], internal congestion, or packet priorities. At the network-interface layer flow control can for instance be used on a connection base. Mechanisms such as end-to-end credit-based flow control ensure that the receiver end has sufficient buffering space to receive all packets sent. At the system level, flow-control deals for instance with the level of occupancy of the buffers within the network layer and transactions can be blocking if the latter are full.

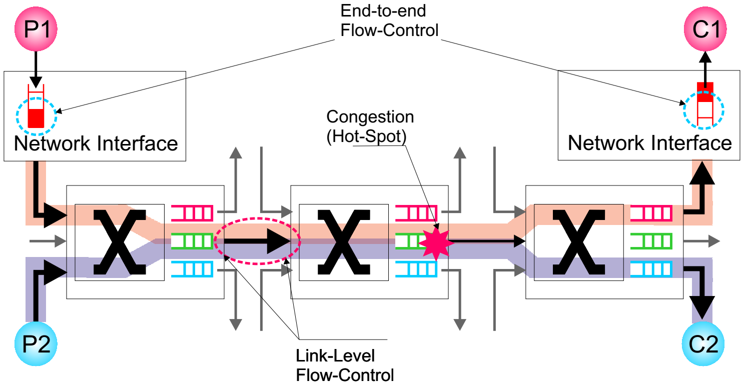


Figure 5.2: *Flow Control and Congestion. Flow control can be defined at all network layers. At the network interface layer it pertains to a single connection. Congestion occurs, at the network layer, when multiple connections interact. Congestion can be controlled by synchronizing the flow-control between several connections.*

Congestion control is usually defined only at the network-layer. Whereas flow-control deals with data flowing onto a single point-to-point connection, congestion-control addresses the impact *multiple* point-to-point connections (sharing network or data layer resources) have on each other. At the network layer, when congestion builds up, packet transmission latencies as well as jitter (latency variations) increase tremendously and throughput plummets possibly resulting in unacceptable degradations for multi-media applications. Figure 5.3 illustrates the evolution of latency when congestion builds up in an NoC that does not allow packet dropping. When no packet dropping can occur the latency is bound by the worst case delay a packet would incur by loosing arbitration at all hops. Let λ be the routing time for one packet once arbitration has been won and assume a connection going over n hops, where p_i is the number of input ports at hop i. Assuming a buffer of size one, the worst case end-to-end latency ($\Lambda(n)$) for a packet occurs when loosing (round robin) arbitration at every hop is $\Lambda(n) = \lambda \sum_{i=1}^{n}(p_i - 1)$. Congestion control covers techniques to monitor network utilization (to detect early signs of congestion building-up) and to modify data transmission to keep traffic levels from overwhelming the network medium.

Though distinct, the concepts of flow and congestion control are nevertheless connected. One could use the flow control mechanisms to control the amount of traffic that is put onto the network hereby reducing the network congestion. For instance at the data-link layer, virtual channel flow-control reduces the effect of head-of-line blocking and thereby globally decreases congestion. At the network layer, packet

[2] The amount of backpressure regulates the number of packets injected into the network.

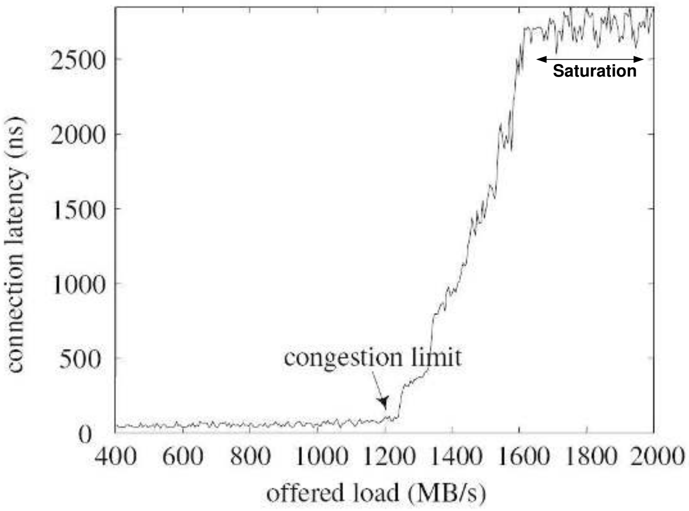


Figure 5.3: *Congestion build-up as function of the offered load [223]. Congestion is characterized as a dramatic increase in latency with increasing offered network load. In an NoC that does not allow packet-dropping, the latency saturates. The saturation level corresponds to the worst case delay a packet would incur by loosing (fair) arbitration at all hops.*

dropping is a common congestion control technique in classic networking, though it is not generally desirable in NoCs because retransmission requires extra buffer space for packet re-ordering.

5.2 Actuator - Injection Rate Control

This section discusses the non-work-conserving actuator (injection rate control) used throughout this chapter. This actuator performs traffic shaping at the network-interface layer. Restricting ourselves to actuators at this layer has been a deliberate choice because the network-interface layer controls the initial injection of traffic into the network layer, so it has a major impact on congestion. At the network layer, actuators typically discard packets or deflect packets from their optimal path. Both packet-dropping and adaptive routing are often undesirable for industrial-grade NoCs because of high implementation costs (re-transmission and re-ordering buffers are required). Only very recently have actuators at the data-link layer been addressed in related work (Section 5.5).

This section first gives an overview of the conceptual mechanism of injection rate control actuators. We then discuss theoretical aspects of algorithms to perform injection rate control. Finally, we show the effects of a proof-of-concept implementation of this actuator in the $Gecko^2$ MPSoC platform emulator ($Gecko^2$ is further discussed in Chapter 6).

5.2.1 Mechanism Concept

The amount of traffic that is injected into the network layer can be limited by providing an injection rate control mechanism at the level of the network interface (NI). Traffic is shaped at the sender NI by controlling three parameters of the packets sent: their priority-level, their length and the time they are injected in the NoC. Adapting traffic shapes to the current quality requirements allows to control the communication in the NoC and is a pre-requisite to providing soft real-time guarantees.

The time wherein an NI is allowed to inject packets into the network is denoted as the *send window*. The shape of the send window is defined in the injection rate control actuator thanks to one running counter giving a global timing and three registers. The first shape register, `Low` (L) specifies when the window starts and the second shape register `High` (H) when it ends. By setting the low and high value, a controller is able to describe a single send window within the whole period of length T (Figure 5.4(a)). The third register `Spread` (S) specifies how often the window appears over time within the period of a running counter wrap-around time. So by increasing the spread S, a single send window can be spread over the whole period T (Figure 5.4(b) and Figure 5.4(c)).

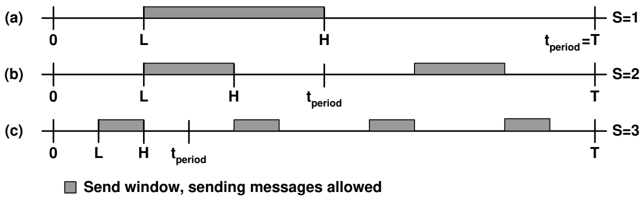


Figure 5.4: *It is possible to specify the size, the location and the amount of spreading of the 'send window' by adjusting the low (L), high (H) and spread (S) value, with $t_{period} = T/S$.*

The concept of window spreading over a maximum period T is related to the implementation of injection rate control in the *Gecko*^{2} MPSoC platform emulator (Chapter 6). The maximum period T is defined by a running counter (19 bits wide[3]). The spreading[4] S is a unit-less value that allows to reduce the maximum period T to the actual period used $t_{period} = T/S$. Hereafter when we speak about window spreading over the maximum period T, we really mean a smaller window, but repeating more often as t_{period} is smaller. We define Ω to be the fraction of the window $\omega = H - L$ over t_{period}:

$$\Omega = \frac{H - L}{t_{period}} = \frac{\omega}{t_{period}}$$

By adjusting the send windows for the different network interfaces in such a way that they do not overlap, it is possible for a central run-time controller to ensure that two tiles never inject packets at the same time.

[3] The most significant 16-bits are exported as a time-stamp to the run-time manager. *Gecko*^{2} is clocked at 33MHz so that the counter wraps around after 16 ms ($2^{19} \cdot \frac{1}{33} \cdot 10^{-6}$), roughly matching the scheduling time of the run-time manager.

[4] S is really implemented as a bit-mask and is used to set $t_{period} = T/2^n$ ($n \in [1, 19]$).

5.2.2 Theoretical Aspects of Injection Rate Control

The traffic shaping actuator in our system is based on a sliding window mechanism. Packets are only injected in the network during the time a window of size Ω is opened. The purpose of defining a control algorithm is to answer two questions. (1) How should the size of this window be modified in time to quickly reduce throughput? (2) How should the window be modified in time once throughput has been reduced? The size of the window is based on binomial congestion avoidance [20]:

$$Window\ Increase: \quad \Omega_{new}(t) = \quad \Omega(t) + \frac{\alpha}{\Omega(t)^k} \quad (\alpha > 1) \tag{5.1}$$

$$Window\ Decrease: \quad \Omega_{new}(t) = \quad \Omega(t) - \beta\Omega(t)^l \quad (0 < \beta < 1) \tag{5.2}$$

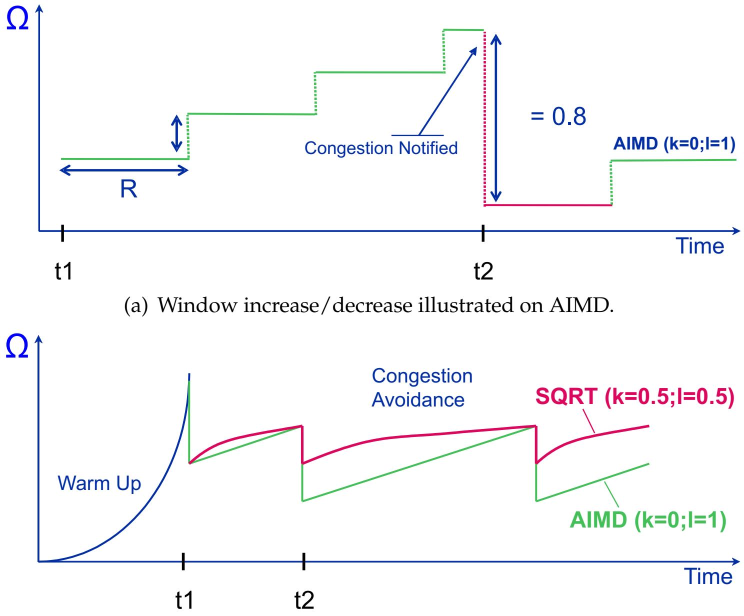
While no congestion is notified, the window size is gradually increased every time-interval R (Figure 5.5(a)).In the general case, the increase amount is proportional to Ω^{-k} (Equation 5.1). When congestion has been notified, the window size is decreased (Equation 5.2), usually by a larger amount than it is increased (Figure 5.5(a,b)). The parameters k and l in Equations 5.1 and 5.2 define the aggressiveness at which the windows are opened and closed and therefore their impact on response to congestion. To ensure a good trade-off between probing aggressiveness and congestion responsiveness, we use the $k + l$ rule[5] defined in [20]: $k + l = 1$ and $l \leq 1$. Figure 5.5 shows the effect of two different sets of (k, l) values that follow the $k + l$ rule. Additive Increase Multiplicative Decrease (AIMD) uses $(k, l) = (0, 1)$ and yields a windowing mechanism that is both efficient and simple to implement (Figure 5.5(a)). The square root algorithm (SQRT) uses $(k, l) = (0.5, 0.5)$ and thus changes the window size proportionally to $\sqrt{\Omega}$ which yields a smoother traffic shaping but is more computationally intensive (Figure 5.5(b)).

5.2.3 Traffic Shaping - Proof-of-Concept

In order to demonstrate the effects of the injection rate control actuator, we created on our emulation platform the setup detailed in Figure 5.6 (further detailed in Chapter 6). We use the traffic generated by a Motion-JPEG video decoder running on the emulator. It is composed of four tasks running concurrently on the computation resources of the platform (Figure 5.6). Two of these tasks, the *sender* and the *receiver*, run in software on the StrongARM processor (tile 3). The two other tasks, are hardware blocks: a task that performs the *Huffman decoding* and the *dequantization*, further denoted as *Huffman block*, and a task that performs a *2D-IDCT* and a *YUV to RGB conversion*, further denoted *IDCT block*. These tasks are mapped on tiles 1 and 8 respectively. The *sender* task sends an encoded video data-stream to the *Huffman* block. The Huffman block sends the decoded data stream to the *IDCT* block. The output of the *IDCT* is sent back to the StrongARM to be displayed by the *receiver* task.

The purpose of the experiment is to show the effects of the actuator (varying window sizes and spreading) has on real NoC traffic. In this example monitoring is

[5] The $k + l$ rule ensures that the system responds quickly and remains in a stability region. See [20] for a thorough analysis.



(a) Window increase/decrease illustrated on AIMD.

(b) SQRT gives a smoother window variation compared to AIMD (AIMD steps from previous figure are here smoothed-out due to a coarser time-scale).

Figure 5.5: *Traffic shaping is based on a sliding-window mechanism.*

performed at the network-interface layer. Statistics are gathered locally by dedicated hardware in the NIs and are centrally collected by the run-time manager. They include, for every task in the application, the number of messages effectively sent, received and blocked at a given NI. In this experiment, the run-time manager samples the relevant NIs every 20 ms[6].

Figure 5.8 shows a combination of the message statistics captured at NIs 8 and 3 as they are communicated to the run-time manager (Figure 5.7). It is a stack plot composed of the number of packets sent at NI 8 and at NI 3, of the number packets received and among them how many have been blocked (input buffer overflow at NI 3). This figure shows two experiments ran one after the other. The same video sequence has been played twice with different windowing techniques: spread (i.e. $S > 1$) and a single continuous send window ($S = 1$). During both experiments the send window size is gradually diminished from a size T (window completely opened during the send period of T) down to the smallest possible (non-zero) value, corresponding to the sending of a single packet over the period T. Experiment 1 in Figure 5.8) has been obtained by applying a window spreading technique and gradually diminishing the size of the window (Figure 5.4(b,c)). Experiment 2, uses the same diminishing window sizes, but shows the effect of having the window composed of continuous blocks of bandwidth (Figure 5.4(a)).

[6] The regular sampling rate on the emulator is of 50ms, but to capture more details in the graphs displayed in here we have increased the sampling rate.

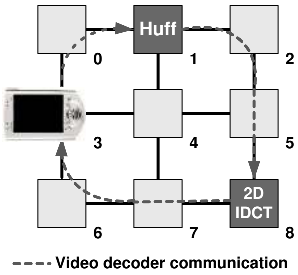


Figure 5.6: Mapping of the Motion-JPEG application on the platform. We monitor the connection 2D IDCT → StrongARM (path $8 \rightarrow 7 \rightarrow 6 \rightarrow 3$).

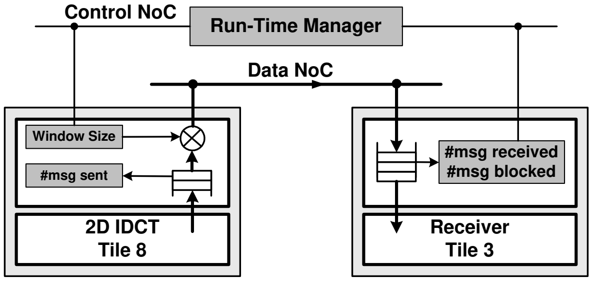


Figure 5.7: Data and control networks on the path $8 \rightarrow (7 \rightarrow 6) \rightarrow 3$. The control network interface in the sender NI (Tile 8) enforces the window size to shape traffic and collects the number of messages sent. The control network interface in the receiver NI (Tile 3) collects the number of messages received and the number of messages blocked in the NI upon receiving. The run-time manager in the StrongARM (Tile 3) collects message statistics and varies the window size in the sender NI.

Experiment 1 in Figure 5.8 shows how, for a spread window technique, a good window size value can be found to match the application to the platform and reduce the network congestion. Indeed, around $t = 3.3$ (10^9 OS ticks), the number of blocked messages drastically diminishes without impacting the number of sent/received messages). Further diminishing window size throttles the IDCT sender and the number of sent and received messages goes below the application requirements (but can be used to give priority to another application sharing one of the links on the connection $8 \rightarrow 3$). Experiment 2 shows that, for this application, the block allocation technique very rapidly throttles the IDCT sender without actually diminishing the level of (instant) blocking at the network interface layer. Because windows are allocated in a complete block, when the window is closed the IDCT quickly fills up its send buffers and then its computation stalls. Only when the window is opened again (in a single block) and its send buffers are emptied can the computation be resumed. As the granularity of the send period T is much larger than the computing time of the IDCT block it is impossible to find a good window size to match application and resources. This experiment outlines the importance of using the spread value S in addition to the send window size ω.

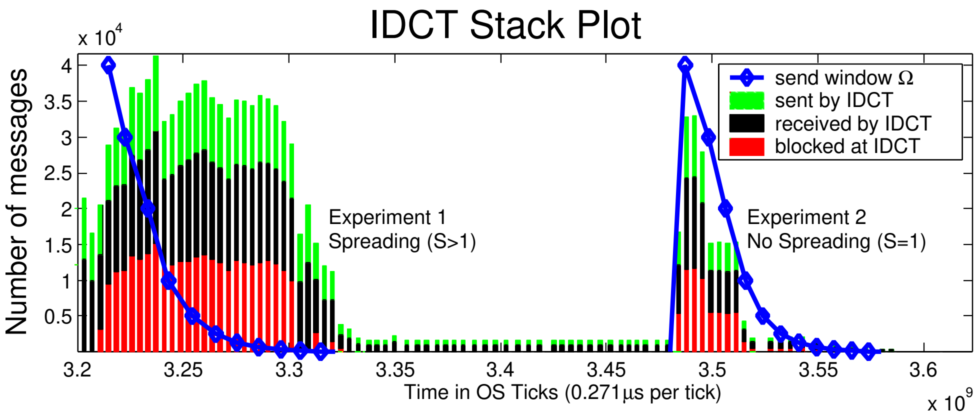


Figure 5.8: *Traffic shaping of the MJPEG on connection* $8 \longrightarrow 3$*. Two experiments using the same video sequence are ran one after the other. In both experiments the window size is diminished from completely opened down to almost closed. The first experiment spreads the window opening over the maximum period T by increasing the value of S and decreasing* $\omega = H - L$*, the second one keeps* $S = 1$ *and only decreases* ω*.*

5.3 Monitoring and Decision-Making

The design space of reactive communication control can be naturally decomposed along three axis that correspond to the three components previously mentioned: monitoring, decision-making and actuator. In our exploration, we have chosen to fix the actuator at the network-interface layer, because it is the layer that injects packets into the network, thus the one that has most impact on congestion and on end-to-end flow control. This injection rate control actuator has been implemented in $Gecko^2$ emulator.

This section discusses the options with respect to monitoring and decision-making. So, given an actuator, how to explore the decision-making and monitoring axis of reactive communication control? What should be the granularity of communication control? This section also positions our approach with respect to the work performed by Marescaux [132]. Tables 5.1 summarizes the differences.

5.3.1 Decision-Making Axis

The role of the decision-making algorithms is to control the actuators in order to regulate traffic.

Types of Decision-Making

We have seen in Section 5.2.2 that the window size is one of the control parameters of an actuator. It is up to the decision-making algorithm to choose an appropriate value for these parameters. Nevertheless, there are actually two different types of decision making:

1. *Need to act at all?* The very first question a decision-making algorithm has to make is **if** it is needed to trigger an actuator at all. Indeed it is desirable to maintain a stable system, avoiding to permanently be in a transient mode. A certain level of hysteresis is thus desirable.

2. *How to regulate the actuator?* The second level of decision-making is **how** to modify the input parameters of the actuator. In the example of injection rate control of section 5.2.2, parameters are window size, window spread and values for k and l (Equations 5.1 and 5.2).

Levels of Decision-Making

Decision-making, whether pertaining to flow-control or to congestion-control can be made by integrating information from one or more layers of the network protocol stack. Moreover, at each of these layers, information can be considered locally or globally.

At one extreme of the decision-making spectrum, decisions can be done independently of each other, locally at the data-link layer. There are as many independent decision points as there are links in the system. At this decision-level, called $Local_{FC}$, only (data-link) flow-control decisions can be taken because information for congestion-control is not available.

In terms of congestion-control, the extreme of the spectrum of decision making is localized in the routers at the network layer. The most localized decision making mechanism can independently run in every router by taking into account any subset of the links entering the router. The local congestion-control decision-making can take into account any combination of the links into the local router.

At the other extreme of the decision-making spectrum, information about all layers of the network protocol stack is globally integrated for a (single) central congestion-control decision-making mechanism, called $Global_{CC}$. $Global_{CC}$ (global congestion-control) considers all connections globally so as to control congestion in the system.

Decision-Making Options

Remember that the decision making algorithms can be defined at all layers of the NoC protocol stack. We have also seen that there is a spectrum of locality of the decision making process. At one extreme it can be fully distributed to make decisions on every individual link and at the other extreme it can be completely centralized and integrate information from all NoC monitoring modules throughout the platform.

It is interesting to note that the two types of decision making (**if** and **how**) are orthogonal to whether decisions are made locally or globally. As all combinations of decision types and levels are possible, the resulting design space, illustrated in Table 5.1, and explained next is quite large.

GI (Global-If) At the network-interface layer, it is relevant to consider a global co-scheduling of all connections in the system (GI_{CC}). The assumption here is that the run-time manager manages all connections and can for instance assign them priorities. Based on these priorities and monitoring information, the run-time manager can decide whether certain actuators have to be reconfigured or not. A global view of the system, pertaining to flow-control (GI_{FC}), is explored in the central communication control management system discussed in Section 5.4.

LI (Local-If) It is also possible to locally decide on the activation of an actuator only based on local monitoring information. For instance monitoring information from the local router can be used by a local decision-making mechanism (situated in the router or associated network-interface).

GH (Global-How) At a global level, more information can be taken into account to optimize the control parameters of the actuators. One particular example exploiting a global view of the system (for flow-control) is the pipelined window allocation addressed in Section 5.4.5. In essence, the start times of the windows are set as to minimize the waiting send times for a pipeline of tasks. Setting the window start time only based on local information would fail to capture this pipelining.

LH (Local-How) A simple algorithm such as AIMD can be locally applied to decide on the window size variations. This local view of the first level of decision-making is addressed in the distributed congestion control mechanism, detailed by Marescaux [132].

	Decision Level					
	Flow-Control			*Congestion-Control*		
Decision Type	$Local_{FC}$	$\Leftrightarrow$	$Global_{FC}$	$Local_{CC}$	$\Leftrightarrow$	$Global_{CC}$
If	LI_{FC}	$\ldots$	GI_{FC}	LI_{CC}	$\ldots$	GI_{CC}
How	LH_{FC}	$\ldots$	GH_{FC}	LH_{CC}	$\ldots$	GH_{CC}

Table 5.1: *Design-Space of Decision-Making. For both flow-control (FC) and congestion-control (CC), between the Local and Global decision-level extremes, there is an ensemble of solutions (denoted by . . .).*

5.3.2 Monitoring Axis

Monitoring modules measure NoC communication parameters at all layers of the network protocol stack. This section mainly focuses on monitoring at the network interface layer and at the data-link layer, because they are the two layers that inject data into the network layer so that they have an important impact on congestion[7].

[7] Additionally, at the network layer the parameters with high impact in board-level multi-processor systems, such as the packets dropping rates are irrelevant to the NoCs we consider.

Connection-Level Monitoring

At the network interface layer, monitoring is performed inside the network interface and measures parameters pertaining to end-to-end connections. Figure 5.9 shows the network interfaces of a connection between a producer and a consumer. At the producer network interface we measure the amount of packets sent per unit of time, whereas at the consumer network interface we measure the total amount of packets received and the fraction of these packets that got blocked due to buffer overflow.

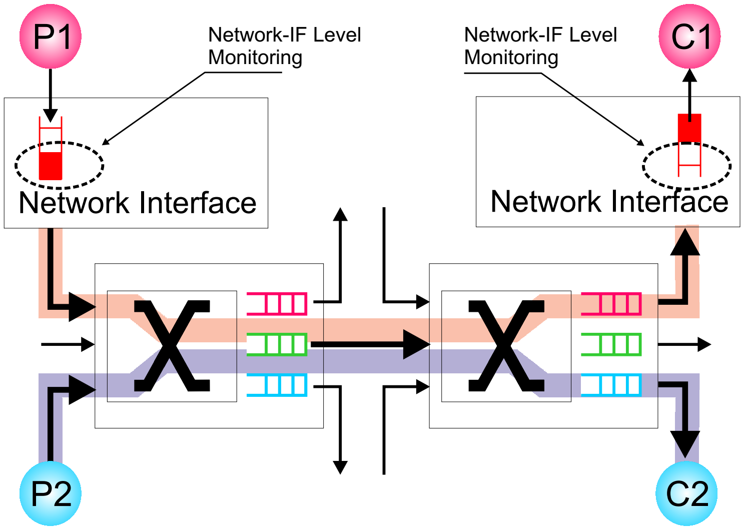


Figure 5.9: Monitoring at the network-interface layer. The connections $P1 \rightarrow C1$ and $P2 \rightarrow C2$ share a link. Monitoring at the NI layer measures parameters such as number of packets received and sent and end-to-end packet latency. Congestion can only be indirectly detected.

Congestion occurs at the network layer, when several connections compete for access to a shared link (saturation occurs when requirements are above link capacity). On a connection it is possible to detect congestion by observing the variations of the communication properties (increased packet latency for instance). However, determining which link on the connection is congested (hot-spot detection) is only possible indirectly. Indeed, one needs the statistics (and information about the placement) of all connections that share links with the congested connection to locate the hot spot. When using only network-interface layer monitoring, a global view of the connections is thus required to determine how they interact with one another. Section 5.4 discusses reactive communication control in a system where only connection-level monitoring is available.

Data-Link-Level Monitoring

The other extreme of monitoring is measuring the properties of individual links. Figure 5.10 shows an example containing two producer, consumer pairs that share the output link of router R1. For instance, by measuring the levels of the input buffers of router R1, we get an indication of the level of congestion a particular data-link (one data link is associated to one input buffer).

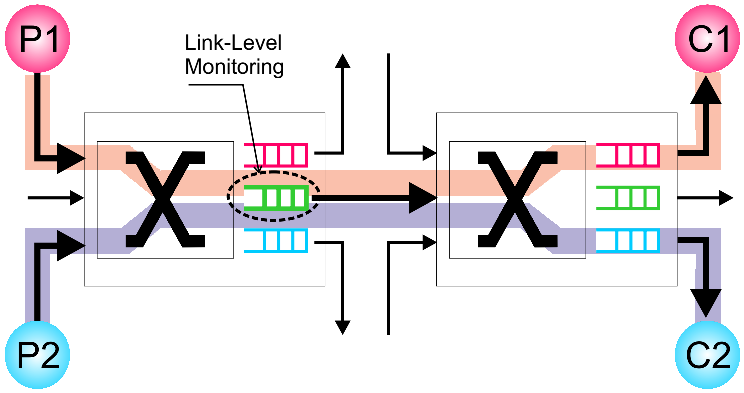


Figure 5.10: *Data-link monitoring. The connections $P1 \rightarrow C1$ and $P2 \rightarrow C2$ share a link. Monitoring at the data-link layer measures parameters such as throughput or buffer occupancy and permits to directly detect congestion.*

However, note that though the measure of buffer occupancy of a particular data-link directly locates a hot-spot, it is not sufficient to determine which particular connections are responsible for creating congestion. Network interface layer information such as the *source* of congesting packets needs to be associated.[8] This implies that data-link monitoring requires higher network layers to transmit information about the source of the packet and in many NoCs this information is optimized away to reduce packet header size. Marescaux [132] uses data-link monitoring. This, however, required to modify our emulator platforms specifically to change the packet header in order to add a packet source field.

5.3.3 Design-Space Choices

We have seen that both axis of monitoring and decision-making permit wide variations in the design-space. What granularity of control is relevant for reactive communication control of NoCs? Table 5.2 gives an overview of the reactive congestion-control design-space.

As the described design-space is fairly large, we fix the actuator to be the injection rate controller at the network interface layer and study two points at the extremes of the design space (one associated to flow-control, the other to congestion control). The *Connection-level monitoring* + *Global*$_{FC}$ *(*GI_{FC}*,*GH_{FC}*,) decision-making* is discussed in this thesis, while the *Link-level monitoring* + *Local*$_{CC}$ *(*LI_{CC}*,*LH_{CC}*) decision-making* is detailed by Marescaux [132].

The configuration of monitoring at the connection level and of global decision making for flow-control is studied in detail in Section 5.4. Because monitoring is performed at the network-interface layer, we need to assume that connections are co-

[8] If connections are to be individually identified, both source and destination of the packet are required.

	Parameter	$Values_{FC}$			$Values_{CC}$		
Monitoring	Layer	Link	Net.	NI*	Link$^+$	Net.	NI
Decision-Making	If	LI_{FC}	$\Leftrightarrow$	GI^*_{FC}	LI^+_{CC}	$\Leftrightarrow$	GI_{CC}
	How	LH_{FC}	$\Leftrightarrow$	GH^*_{FC}	LH^+_{CC}	$\Leftrightarrow$	GH_{CC}
Actuating	Layer	Link	Net.	NI*	Link	Net.	NI$^+$

Table 5.2: *Design-Space of Reactive Communication Control. The asterisk symbol (*) indicates the design point used in this thesis. The plus symbol (+) indicates the design point used by Marescaux [132].*

scheduled and co-placed globally (at the system level) so that hot-spots in connections can be detected. As a consequence, the decision-making process also needs to assume a global system-level view, taking into account which connections have priority over others. Both levels of decision-making (when to react, how to modify the parameters) are assumed to be global.

The configuration of monitoring at the link-level and local (hence distributed) decision making for congestion-control is discussed by Marescaux [132]. Marescaux details router extensions with link-level monitoring mechanisms that detect hot-spots at a fine granularity. In terms of decision-making, the first-level (i.e. the **if** decision) is based on connection priorities and assumes a global system-level placement of connections (and priority assignment). The distributed part of the decision-making process is actually the adaptation of actuator parameters (window sizes).

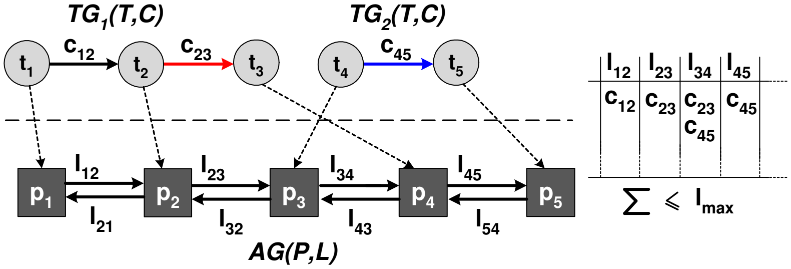
5.4 Global Connection-Level Management

In our NoC communication management scheme, the run-time manager is able to monitor the traffic at every network interface. Based on this information the run-time manager can manage traffic by limiting and/or shaping the amount of packets a tile is allowed to inject into the NoC. This way, the packet rate of the data producer can be matched with the consumption rate of the data consumer in order to handle network blocking conditions.

5.4.1 Setting and Problem Definition

Consider the assignment of two task graphs, TG_1 and TG_2, depicted in Figure 5.11. The run-time manager's resource assignment algorithm (Section 3.4), attempts to cluster communicating tasks as much as possible in order to optimize resource usage and minimize inter-application interference. However, it still occurs that a communication link is shared between two applications. In this example case, the link l_{34} is shared between TG_1 and TG_2.

Every task graph edge is associated with a load estimate, i.e. a number that denotes a worst case estimate of the amount of communication throughput required by this edge. This number is provided by the application designer either by design-time analysis or by application profiling. During the assignment, the run-time manager



l_{12}	l_{23}	l_{34}	l_{45}
c_{12}	c_{23}	c_{23} c_{45}	c_{45}

Figure 5.11: *Assignment of two application task graphs, TG_1 and TG_2, to an architecture graph AG (graph representation of the MPSoC communication resources). The link l_{34} is a shared link between the two applications. The run-time manager ensures that the total assigned communication load does not exceed the link capabilities.*

ensures that the cumulated communication load of all connections sharing a certain link does not exceed the maximum load of the link.

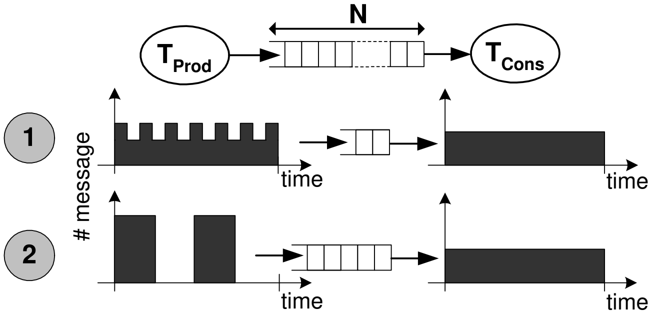


Figure 5.12: *Although producer and consumer can be matched on average with respect to production and consumption of messages, there is still a need for buffer space to handle temporary differences due to e.g. message bursts. The amount of required buffer space depends on the difference in production/consumption rate: small variations require, small buffers (1) while very bursty traffic requires a large amount of buffer space (2).*

Although both producer and consumer can be well matched on average when it comes to production and consumption of messages, there might be transient differences. For example, the production of messages can happen in bursts, while the consumption happens at a steady rate. To overcome these transient differences, message buffers are inserted in the network interface both for sending and receiving messages. Figure 5.12 illustrates a producer-consumer pair that is, on average perfectly matched. However, due to the burstiness of the producer, temporary buffer space is required. For small bursts or variations in the sending pattern (1), a small amount of buffer space is sufficient, for very bursty traffic (2), a larger amount of buffer space is required.

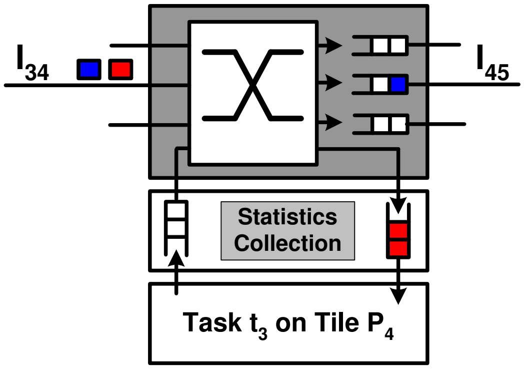
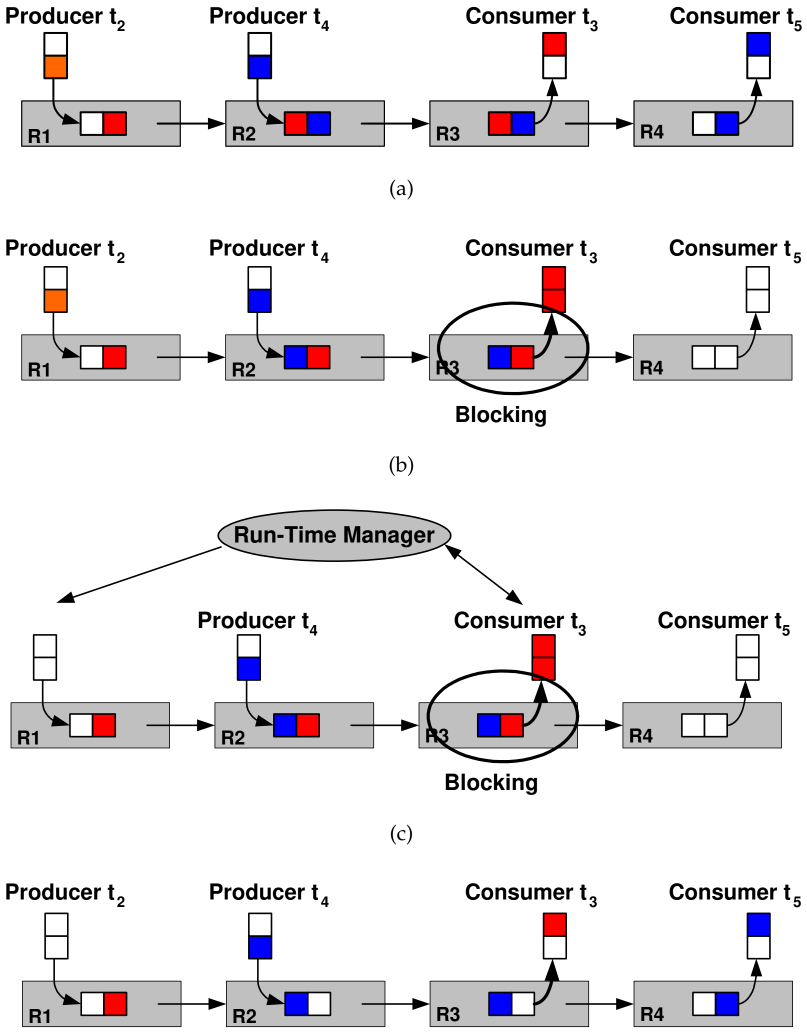


Figure 5.13: Blocking conditions in a router. Congestion builds up at the network layer because of buffer overflow at the network interface layer (mismatch between consumer and producer tasks). Unrelated applications, sharing the congested link, are impacted.

As the amount of message buffers is platform dependent, a buffer overflow is still possible even when producer and consumer are well matched (assuming the absence of end-to-end flow control at the network-interface layer). Figure 5.13 zooms-in on the situation of the shared link at tile P_4. Due to bursty communication of task t_2 combined with steady consumption of task t_3, the buffer in the network interface is still full, while a new message is already lining up on link l_{34}. As l_{34} is a shared link, a message coming from t_4 is waiting on the same link to be routed to its destination. The router has two options in this case. The first option is to drop the packet. This option assumes a mechanism that (1) detects packet loss and that retransmits messages at the sender side and that (2) re-orders messages at the receiver side. The second option is to temporarily queue the packet inside the network router buffers (rely on the flow control at the network layer to block communications). This second option assumes that such situations occur very rarely as producer and consumer are well matched. However, queuing the packet in the network router can create *blocking* conditions. This means that the message coming from t_4 cannot be routed until the message from t_2 has been routed.

Example 5.1: Solving blocking conditions (Figure 5.14).
Producer t_2 sends messages to *Consumer* t_3, while *Producer* t_4 communicates with *Consumer* t_5. Both communications share a common link (between R2 and R3) (a). Suppose that Producer t_2 temporarily produces more messages than Consumer t_3 can handle. In our approach, incoming messages for Consumer t_3 that cannot be stored in the NI input buffer are stored in the router until the required buffer space becomes available (b). However, this creates communication interference between Producer t_4 and Consumer t_5 resulting in more jitter and a decreased throughput. This kind of interference is further denoted as (network-layer) *blocking*. As soon as the run-time manager is informed of the blocking issue, it limits and/or shapes the output of the offending Producer t_2 (c). This is achieved by using the injection rate controller to apply a *send window*. This will result in fluent traffic for all communications (d).



(a)

(b)

(c)

(d)

Figure 5.14: *(a) Two communicating producer consumer pairs: t_2 communicates with t_3 and t_4 communicates with t_5. (b) Blocking occurs due to a mismatch between t_2 and t_3 with respect to production and consumption. This causes interference with producer-consumer pair t_4 and t_5. (c) The run-time manager monitors traffic, collects statistics information and reshapes producer traffic to reduce blocking. (d) The end result is fluent traffic and a reduction of inter-application interference.*

Intuitively, it is clear that the amount of buffer space inside the network interface will influence if and when such a router blocking condition occurs. In this case, one extra buffer space would avoid the message coming from t_4 being blocked. Hence, it is up to the run-time manager to match the communication speed between producer and consumer given the amount of buffer space available. Furthermore, if blocking occurs due to the fact that producer and consumer are (temporarily) not well matched, the run-time manager is responsible for minimizing any potential inter-application interference. This scenario is illustrated by Figure 5.14.

The experiments of Section 5.2.3 have shown that this mechanism is able to effectively limit the amount of communication interference caused by blocking. Furthermore, the same mechanism (traffic re-shaping) can be used to provide a soft form of QoS.

It is important to realize that, with the presented approach, the run-time manager only needs to react in case there is disturbing interference on a shared link between two or more communication pairs. A temporary usage of network buffer space on an unshared link will not cause any interference to other communications and hence does not require immediate action. Furthermore, it might be that no action is needed because the blocking (e.g. multimedia) application has a higher priority than the affected (e.g. batch) application. In addition, when producer and consumer are well-matched, there is equally no need to take any controlling action and hence no resources are wasted for actively managing the communication.

5.4.2 Base Window Management Algorithm

As Figure 5.15 illustrates, one can model an application task graph as a combination of a set of producer-consumer pairs. Hence, the base algorithm focuses on window management for a single producer-consumer pair that needs to be matched. The run-time manager receives input from the statistics collection component of the consumer network interface and, if necessary, adjusts the send window values of the injection rate controller of the producer.

When confronted with blocking conditions, the first goal of the run-time manager is to minimize inter-application interference as fast as possible. This effectively means reducing the send window of the producer that causes blocking, even if that temporarily reduces the throughput of the offending producer-consumer pair. Secondly, the run-time manager is responsible for matching producer and consumer given the platform properties and the assignment of producer and consumer. This is done by reshaping the send window in such a way that blocking is eliminated, while optimizing throughput. Algorithm 10 details this procedure.

This window management algorithm, starts by calculating an initial window size ($\omega = H - L$) for an initial t_{period}^{init} (line 3). ω is defined as the time the window is opened during t_{period} (i.e. $\omega = \Omega \times t_{period}$). In this case, ω is a multiple of a minimum send window slot ω_{min}. The run-time manager calculates these initial values based on the application task graph communication details, the user requirements with respect to the application (e.g. required video throughput) and the available buffer space in the network interface.

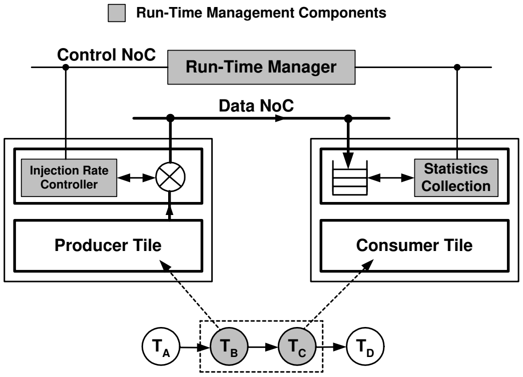


Figure 5.15: Modeling an application task graph as a combination of producer-consumer pairs. The run-time manager is modeled as a (central) algorithm capturing statistics and handling the injection rate controller.

Equation 5.3 and Equation 5.4 detail the relation between production rate P, consumption rate C, the network interface buffer space N and the time ω that the producer is allowed to inject messages into the network. Equation 5.3 states that initially (during the first window opening) the difference between producer and consumer rates can be completely absorbed by the buffering space available. To avoid overflows during steady state, it is sufficient to impose that the difference between producer and consumer rates (window opened) is smaller than the consumption rate while the producer window is closed (Equation 5.4).

$$(P - C) \times \omega \leq N \tag{5.3}$$

$$(P - C) \times \omega \leq C \times (t_{period} - \omega) \tag{5.4}$$

After setting the initial (or new) window values (line 6), the run-time manager periodically collects traffic statistics (line 7) from the relevant network interfaces. These statistics include the amount of sent, received and blocked messages together with the maximum message blocking time. After acquiring the statistics, the run-time manager checks if the reported amount of blocked messages exceeds a certain threshold value (line 8). If so, the algorithm has to (re)calculate and update the currently used window values: ω and t_{period}. This primarily involves reducing the size of the send window ω of the producer until blocking drops below the threshold. The reduction of the send window size (in amount of *slots*) depends on the ratio of the amount of blocked messages versus total amount of received messages. Two cases can be distinguished: *LargeReduction* (line 16) and *SmallReduction* (line 18), depending on whether the amount of blocked messages is larger or smaller than half the amount of received messages (Table 5.3).

If the amount of blocking does not exceed the threshold value and the throughput is lower than required (line 21), then the send window size will be increased by a single window-slot (e.g. 100μs) for as long as the new window size ω_{new} remains smaller than the *blocking* window size $\omega_{blocking}$ (line 22).

Algorithm 10: *Run-Time communication window management.*

```
Input: $TG_i$, $AG, TG(T,C) \rightarrow AG(P,L), t^{init}_{period}, t^{min}_{period}$
Output: $\omega$,$t_{period}$
COMMUNICATIONMANAGEMENT()
(1)  WindowStable=false
(2)  $t^{new}_{period}$=$t^{init}_{period}$
(3)  $\omega_{init}$=CalcInitialWindow()
(4)  $\omega_{new}$=$\omega_{init}$
(5)  while (WindowManagement is active)
(6)    ($\omega_{current}, t^{current}_{period}$)=SetWindow($\omega_{new}, t^{new}_{period}$)
(7)    CurrentStats=GetStats()
(8)    if (CurrentStats.Blocking $>$ BlockingThresholdLow)
(9)      if (WindowStable)
(10)       $\omega_{new}$=$\omega_{init}$
(11)       $t^{new}_{period}$=$t^{init}_{period}$
(12)       ResetValues($\omega_{best}$,$\omega_{blocking}$,$t^{best}_{period}$)
(13)       WindowStable=false
(14)     else
(15)       if (CurrentStats.Blocking $>$ BlockingThresholdHigh)
(16)         $\omega_{new}$ = ReduceWindow(LargeReduction, $\omega_{current}$)
(17)       else
(18)         $\omega_{new}$ = ReduceWindow(SmallReduction, $\omega_{current}$)
(19)       $\omega_{blocking}$=$\omega_{current}$
(20)   else
(21)     if (CurrentStats.Throughput $<$ RequiredThroughput)
(22)       $\omega_{new}$ = IncreaseWindow($\omega_{current}$,$\omega_{blocking}$)
(23)   if (($\omega_{new}$==$\omega_{current}$) for $\rho$ consecutive times)
(24)     if (CurrentStats $>$ BestStats)
(25)       $\omega_{best}$ = $\omega_{current}$
(26)       $t^{best}_{period}$ = $t^{current}_{period}$
(27)       BestStats = CurrentStats
(28)     if $t^{current}_{period} > t^{min}_{period}$
(29)       ($\omega_{new}, t^{new}_{period}$) = SpreadWindow()
(30)     else
(31)       $\omega_{new}$ = $\omega_{best}$
(32)       $t^{new}_{period}$ = $t^{best}_{period}$
(33)       WindowStable=true
```

As soon as the window values converge (i.e. are stable for ρ sampling times) (lines 28-33), the algorithm will spread the send window by decreasing t_{period}. Spreading the window over the send spectrum has two effects. First, it reduces the impact of bursty communication by spreading it over the send spectrum. Secondly, it enables a more effective use of the send window for non-bursty communication, meaning that a number of contiguous timeslots in a single block provide less throughput than the same number of timeslots spread over the send spectrum. However, the downside of send window spreading is that it makes communication less controllable by the run-time manager with respect to minimizing inter-application interference (see Section 5.4.5). Table 5.3 details how window size ω and period t_{period} are modified.

Table 5.3: Window size ω and period t_{period} changes.

SmallReduction	$\omega = \omega - \omega_{min}$
LargeReduction	$\omega = \omega/2$
Throughput $< 80\%$	$\omega = \omega + \omega_{min}$
Spreading	$\begin{cases} \omega = \omega/2 \\ t_{period} = t_{period}/2 \end{cases}$

For each t_{period}, the algorithm determines the send window size $\omega_{blocking}$ at which blocking starts. This way, the algorithm searches to reduce the impact of bursty communication, while keeping throughput as high as possible. After maximally spreading the send window over the send spectrum, the window values ω_{best} and t_{period}^{best}, that deliver the best communication performance in terms of throughput and blocking are selected and instantiated. Note that, to keep the base windowing algorithm simple, send window low value (L) is always kept at zero, i.e. the window high value (H) equals the window size ω.

If a significant amount of blocking (re)appears in future (line 9: blocking occurs for a *stable* window), for example due to a change in burst characteristics, the send spectrum is again searched in order to eliminate blocking as quickly as possible.

5.4.3 Base Algorithm Experimental Results

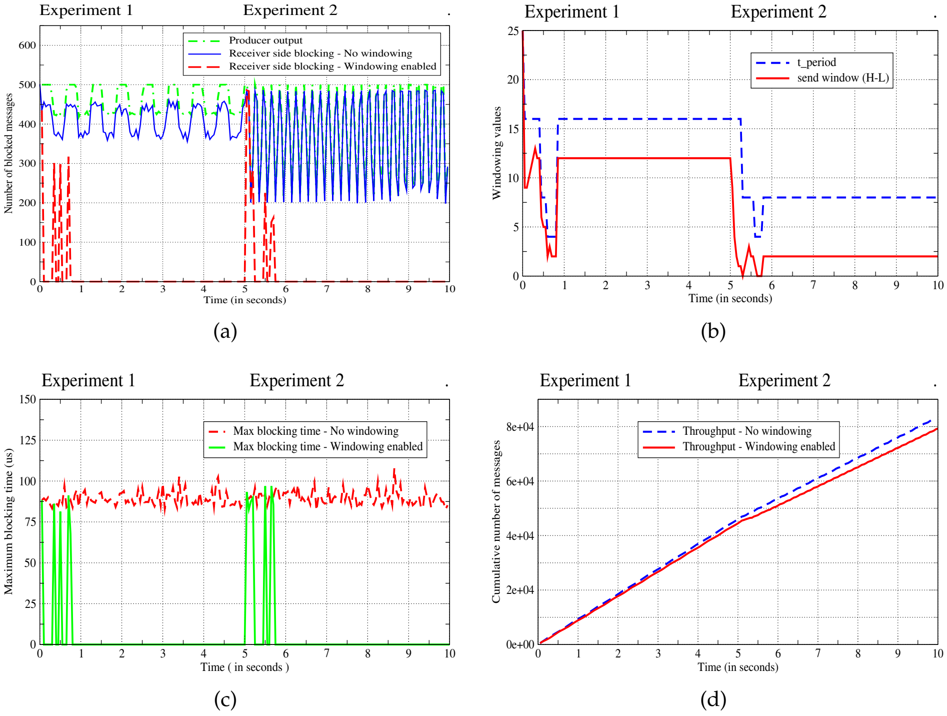
The efficiency of this algorithm is measured in terms of two key factors: one regarding blocking and throughput with respect to the NoC communication and the second one regarding the computational resources that the algorithm requires.

As Chapter 6 explains, the maximum period T of the final demonstrator platform spans 16ms. In the following experiments, we have divided the send spectrum into time-slots of 100μs wide. Hence, the window size ω denotes the amount of 100μs slots.

In our experiments, we found that a t_{period}^{init} value of 16 is a good starting point for the window-spreading in absence of detailed burst characteristics. The higher t_{period}^{init}, the more iterations are needed to reach the optimum window values. Typically one higher order[9] of t_{period}^{init} will result in four to five additional iterations before reaching the optimum. Hence, the practical upper bound for t_{period}^{init} with respect to the presented results is 64.

In order to simulate changing application characteristics due to e.g. altered user settings, all experiments contain two experiment phases, denoted *Experiment 1* and *Experiment 2*. When transitioning to Experiment 2 (i.e. after five seconds), the producer generates the same number of messages in only half the amount of time. This results in more bursty traffic causing more NoC communication blocking.

[9] A factor of two in case of our algorithm since it uses a window-spreading factor of two as shown in Figure 5.4)



(a)

(b)

(c)

(d)

Figure 5.16: *Window management experimental results. In Experiment 2 ($t \in [5, 10]$) twice as much data is sent as in Experiment 1 ($t \in [0, 5]$). (a) Blocking traffic comparison with and without communication management. Without communication management, message blocking at the receiver is proportional to the producer output. With communication management, blocking only occurs during the transient period. Once the run-time manager has found good window values, matching sender and receiver rates, all blocking is removed. (b) Finding optimal send window values. Upon change of the communication characteristics (at the beginning of each experiment), one can observe a transient period of under one second where the run-time manager adapts the window sizes. (c) Comparison of maximum blocking time with and without window management. Displaying the maximum blocking time, shows more clearly that after the transient times, all blocking is removed when using communication management. (d) Throughput comparison with and without communication management. Using communication management only marginally degrades the application throughput.*

Figures 5.16(b), 5.16(d), 5.16(a) and 5.16(c) show different performance aspects of the algorithm. Figure 5.16(a) shows that, when the application starts, there is a significant amount of blocking. This causes a large reduction of the high value (H) and thus the size of the window ω. Due to this action, the next message statistics samples do not show any blocking (Figure 5.16(a)). In turn, this results in a few minor increases of the send window size. However, the following message statistics collected by the run-time manager again indicate blocking. This means the run-time manager will continue decreasing the send window size.

Whenever the window size remains stable for three consecutive sampling periods (i.e. $\rho = 3$), t_{period}^{init} is decreased (Figure 5.16(b)) in order to spread the window.

Spreading the window should alleviate blocking caused by bursty message production behavior and increase the overall throughput for non-bursty traffic. However, spreading the window can also cause blocking to re-occur. Although the send window fraction Ω remains equal, there is a new distribution of computation and communication. This means that due to spreading, messages will be produced and stored in the network interface output buffers when the send window is closed. Hence, the actual communication utilization of the send window will rise, which again increases the blocking risk. In order to decrease blocking for the newly set t_{period}^{init}, the send window ω is decreased again.

After maximally spreading the send window (i.e. after reaching the minimal modulo value t_{period}^{min}), the window values ω_{best} and t_{period}^{best} with the best throughput with minimal blocking are selected and reinstated. For Experiment 1, this takes about 16 iterations. This corresponds to about 800 ms when the run-time manager samples the communication values every 50 ms.

In Experiment 2 (i.e. after 5 seconds), the production of messages gets more bursty: the producer creates the same number of messages, but transmits them in only half the amount of time. The previously *stable* send window values are now reset and new stable values are determined using the same procedure: decreasing the send window size until blocking is removed. Decreasing the send window also decreases the throughput. In order to find the optimal throughput without blocking, the modulo value is decreased. Finally, the most optimal window values are selected (around the 5.8s timestamp).

Figure 5.16(d) shows that the throughput with windowing is at most 6.7% lower than the throughput without applying a send window (i.e. best effort), while, at the same time, blocking (and hence inter-application interference) is almost completely eliminated (Figure 5.16(a) and Figure 5.16(c)).

The communication management functionality is obviously a load on the processing element executing the run-time manager. Figure 5.17 details the load of the communication management scheme when executing on a StrongARM processor. It shows that statistics collection requires a small constant load, even in absence of blocking. In the future, this statistics *polling* could be replaced by an interrupt driven mechanism that only informs the run-time manager in case of blocking at a certain node. The time required for executing the heuristic is mostly stable, but depends on the collected statistics. The total time for traffic management includes the mechanism for collecting traffic statistics, executing the heuristic and the mechanism for setting the new window values.

5.4.4 Case Study on the $Gecko^2$ Emulator

In order to support the case study, we first briefly detail the used emulation platform. An in-depth discussion of the MPSoC architecture of our emulation platform and the capabilities of the NoC it contains, including the run-time management controlled traffic monitoring mechanisms [162], is presented in Chapter 6 .

Our emulated packet-switched multiprocessor MPSoC is implemented using a 3x3 bidirectional mesh NoC linking the StrongARM SA1110 processor (206 MHz) of a

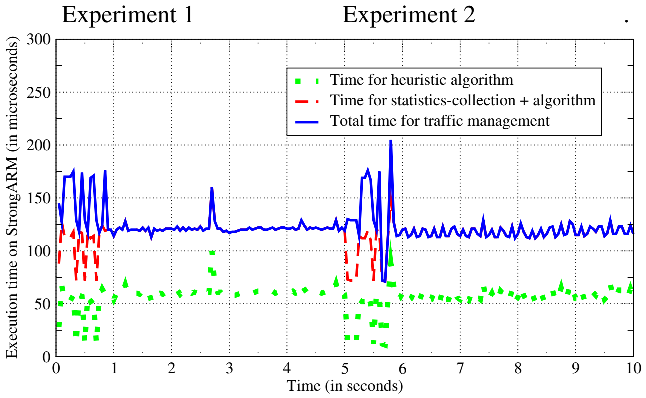


Figure 5.17: *Algorithm execution time measured on a StrongARM ISS (processor load $< 1\%$).*

handheld device to an FPGA containing eight slave processing elements and the NoC (33 MHz clock).

Our packet-switched NoC actually consists of two independent NoCs. The *data NoC* is responsible for delivering data packets, while the *control NoC* is used for transferring run-time management control messages. This separation is vital to our communication management scheme since the control NoC provides a way to control the traffic even in case of data NoC congestion.

We measured on our NoC emulation platform (running at 33 MHz clock-speed) that at every sampling time, the run-time manager takes about 60μs to gather communication statistics from a tile network interface. In total, incorporating such a traffic management inside our NoC platform operating system takes on average 182μs per tile at every sample time (set to 50ms in our system).

Our case study application is an MJPEG video decoder, detailed in Figure 6.17 of Section 6.4.2. Initially, the setup is run without any traffic management. However, the MJPEG decoder throughput can be significantly reduced due to heavy blocking on the shared link caused by a perturbing application (between tiles 6 and 7 in Figure 6.17).

When starting the window management, we first notice a negligible additional load caused by activating the algorithm on the StrongARM of less than 1%. We also notice that the algorithm converges to optimal window values, i.e. values that allow the application to reach the required throughput for the message generator, and that effectively reduce the interference with the MJPEG application.

The simple producer-consumer communication management model used to develop the windowing algorithm converges within about 16 iterations (i.e. 800 ms). The case study takes about 20 iterations (i.e. about 1 second) to find good window values for the IDCT and the generator task that share a communication link. This minor increase can be attributed to the (more than expected) bursty nature of the MJPEG communication. The number of iterations can be reduced by providing more detailed application communication characteristics, like e.g. burst periodicity, burst width and burst magnitude. Another solution would be to extend the t_{period} size range. This would allow to control burst at a finer granularity. However, one should avoid increasing the run-time of the algorithm.

Concluding Remarks

The presented run-time communication management algorithm is suited to handle user-induced changes, like e.g. user changes in the application quality settings or inter-application interference caused by starting a new application.

5.4.5 Algorithm Improvements

This section details a set of improvements for the windowing algorithm. The first improvement deals with algorithm convergence speed and accuracy. The second improvement deals with window placement in the send spectrum. The final improvement considers the concept of task pipelines (previously introduced in Chapter 4), which extends the simple producer-consumer communication management model.

Improving Algorithm Convergence and Granularity

The base algorithm typically uses a t_{period} of 16 timeslots in order to find new window values in a timely manner. This restricts the granularity at which bursts can be spread over the send window. However, decreasing the time-slot size and increasing the t_{period} range to find better window values also dramatically increases the run-time of the algorithm.

This section changes the t_{period} range to $2^{19} - 1$ and reduces the timeslot size w_{min} to 30ns. In order to ensure the algorithm execution time does not explode, we replaced the convergence of window size ω and t_{period} (Table 5.3) by a variable step size δ, as detailed by Table 5.4. This exponential convergence assumes a relatively stable optimal window size, i.e. one that only changes due to, for example, user interaction.

Table 5.4: Variable changes (δ) to window size (ω) and t_{period}.

SmallReduction	$\begin{cases} \omega = \omega - \delta \\ \delta = \delta/2 \end{cases}$
LargeReduction	$\begin{cases} \omega = \omega/2 \\ \delta = \delta/2 \end{cases}$
Throughput $< 80\%$	$\begin{cases} \omega = \omega + \delta \\ \delta = \delta/2 \end{cases}$
Spreading	$\begin{cases} \omega = \omega/2 \\ t_{period} = t_{period}/2 \\ \delta = t_{period}/2 \end{cases}$

Assume that during the first iteration ($i = 0$), the algorithm takes an initial window ω_0 that is completely opened during the modulo t_{period}^{init}, that is $\omega_0 = t_{period}^{init}$. Assume for now we keep t_{period} constant, i.e. $t_{period} = t_{period}^{init}$. At iteration n, ω_n writes as:

$$\omega_n = \omega_0 - \sum_{i=0}^{n} \frac{\delta}{2^i} \tag{5.5}$$

How can we determine the initial value of δ (called δ_0) so that all window values from ω_0 down to 0 are covered? This means that for $n \rightarrow +\infty$, $\omega_n = 0$, in other words:

$$\omega_0 = \sum_{i=0}^{+\infty} \frac{\delta}{2^i} \tag{5.6}$$

The geometric series of 5.6 converges absolutely to the value 2δ. Hence, if we want to exploit the maximum window range and reach every possible window value, then initially δ is $\delta_0 = t_{period}^{init}/2$ (remember $\omega_0 = t_{period}^{init}$). When at iteration i, t_{period} changes, the value of δ is reset (Spreading in Table 5.4). One could equally consider a smaller δ_0 value, like e.g. $\delta_0 = t_{period}^{init}/4$. The larger δ_0, the more aggressively the algorithm will react to close the window upon blocking conditions.

Window Co-Placement

The base algorithm essentially changes the send window size $\omega = (H - L)$ by keeping the window low value (L) at zero and by altering the window high value (H). This essentially means that, in order to minimize interference, we have to make the windows of two producers that share a communication link as disjoint as possible. Although it might seem trivial, one has to consider that communication paths shared by several producer-consumer pairs can be quite complex.

Let's first deal with the issue of placing two windows. In case both t_{period} are equal, one can place the second window after the first one (Figure 5.18(a)). If this would result in crossing the t_{period} boundary, the window is shifted until this is no longer the case (Figure 5.18(b)). As the t_{period} are equal, the first send window will be representative for all consecutive periods.

In case of different t_{period} this is no longer the case. So, there are essentially two options. First, if the t_{period} values are prime numbers with respect to one another, then the windows will continuously shift with respect to each other. This means that all window placements are equivalent (Figure 5.18(c)). Second, if the t_{period} are confined to being a power of two, we can again find an optimal configuration. In the setup of Figure 5.18(d), we can minimize the overlap by stating that $L_2 = H_1$ modulo $t_{period-2}$. Note that, if t_{period} values are orders of magnitude apart, window placement will have little effect (Figure 5.18(e)).

For handling multiple windows in an efficient way, we make some assumptions. First, we rely on the fact that the run-time manager will attempt to cluster applications and by consequence minimize inter-application communication interference during task assignment. Second, the remaining dependencies will be for a relatively small amount of producer-consumer pairs. Consider the following experiment: exploring all possible assignments for two to four producer-consumer pairs in a 3-by-3 mesh network and examining the amount of shared links. As Table 5.5 illustrates, most cases will have to deal with combining two or three windows when placing 4 producer-consumer pairs. Hence, the suggested approach for multiple windows

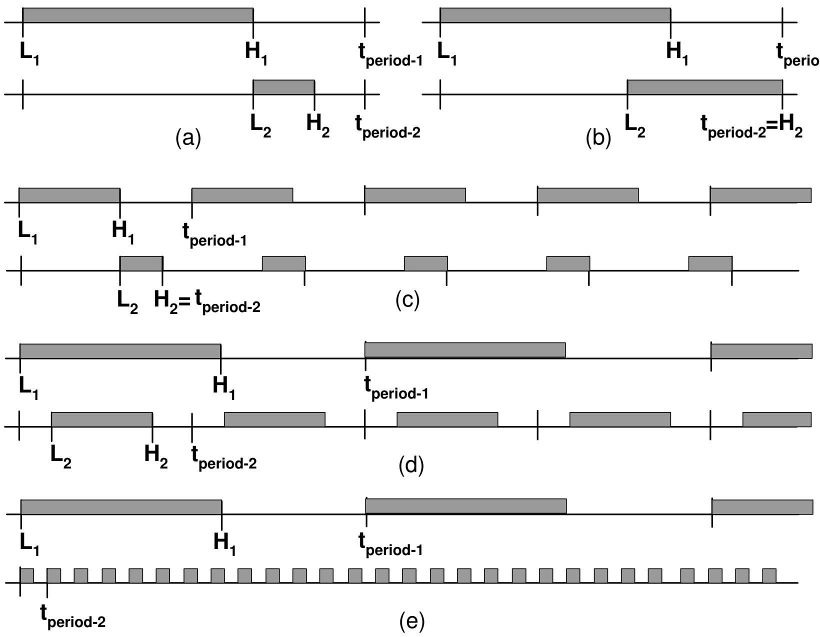


Figure 5.18: Relative placement for two windows. (a) Equal t_{period}, non-overlapping windows. (b) Equal t_{period}, window shift minimizing overlap. (c) t_{period} values are prime numbers with respect to each other, overlapping always occurs. (d) t_{period} values are powers of two, windows of comparable sizes can be placed to reduce overlapping. (e) t_{period} values are powers of two, windows sizes differ too much for placement to significantly reduce overlap.

is to choose a reference window that is relevant to every window that needs to be placed in the pipeline of producer-consumer pairs.

# Pairs	2 pairs share a link	3 pairs share a link	4 pairs share a link
2	41%	-	-
3	80%	8%	-
4	96%	28%	1%

Table 5.5: Percentage of pairs that share a link for every possible assignment in a 3x3 mesh network.

Dealing with Task Communication Pipelines

Up till now, the algorithm considered an application as a set of producer-consumer pairs (Figure 5.15). This, however, is a simplification of reality. Consider a pipeline of tasks like the one depicted in Figure 5.15. If the pipeline stalls at the last task T_D, messages will be blocked. As a result, the send window of T_C will be decreased. This might cause T_C to slow down and, hence, consume fewer messages. In turn, blocking will appear for task T_C. This will affect the send window of T_B, and so on. Eventually, the send window of T_A could be reduced. It is clear that during this type of transient periods the throughput is strongly impacted in other stages of the pipeline, like e.g. for T_C. This example shows that the algorithm requires support for handling communication pipelines. However, considering that calculating new window values happens at run-time, we require a lightweight approach for handling window sizes of pipelines.

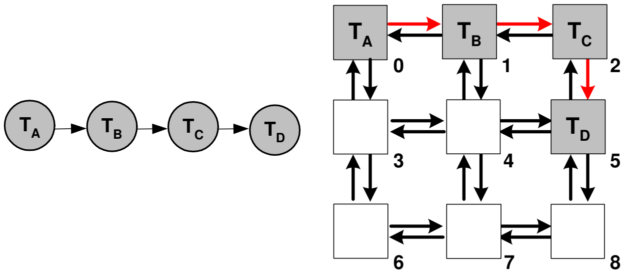


Figure 5.19: Task graph of a pipelined application and its platform mapping.

The proposed solution is to consider the two end points of the pipeline in order to converge faster[10]. Still, a new window value is determined for every producer-consumer pair (if necessary). However, instead of communicating this new window to the window-setting mechanism, the new window values are stored until all producer-consumer pairs have been covered. For every producer that is about to receive new window values, the run-time manager checks whether there is a producer earlier in the pipeline that is also going to receive new window values. If so, only the earlier producer(s) will receive a window update. This solution only works for directed task graphs. In cyclic task graphs, one cannot determine which task is first.

Example 5.2: Updating window values for a task pipeline (Figure 5.20)
In case of a pipelined application, not all producers receive a window update in case of message blocking conditions. Only the message producers without updated predecessor producer tasks receive new window values. As the example shows, T_2, T_3, T_4 and T_5 all require updated window values. As T_4 and T_5 have T_3 and T_2 as predecessors, their window values will not change.

In order to verify this improved algorithm, consider the set-up depicted in Figure 5.19. Initially, this setup is completely balanced with respect to the producer-consumer pairs, i.e. there is no blocking and there is no significant slack-time. When cutting the consumption of task T_D by half after 100 ms, blocking will occur and the effect should propagate down to task T_A.

[10] Because of the sampling period of the monitoring, it is not possible to guarantee that for multiple stages in the pipeline the assignment algorithm would not get into an oscillation loop. By considering only the two end points in the task communication pipeline we also ensure that the algorithm will converge.

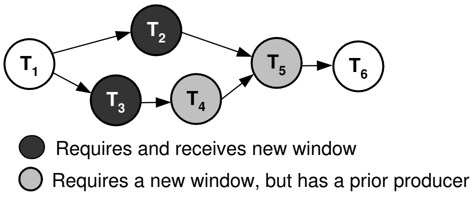

T1
T2
T3
T4
T5
T6
● Requires and receives new window
● Requires a new window, but has a prior producer

Figure 5.20: *Setting windows for task pipelines: only the first pipeline producers receive an update.*

The base windowing algorithm reacts by altering the send window sizes of tasks T_A, T_B and T_C (Figure 5.21(a)). However, due to the window interaction, the blocking with respect to T_D is not at all resolved (Figure 5.21(b)).

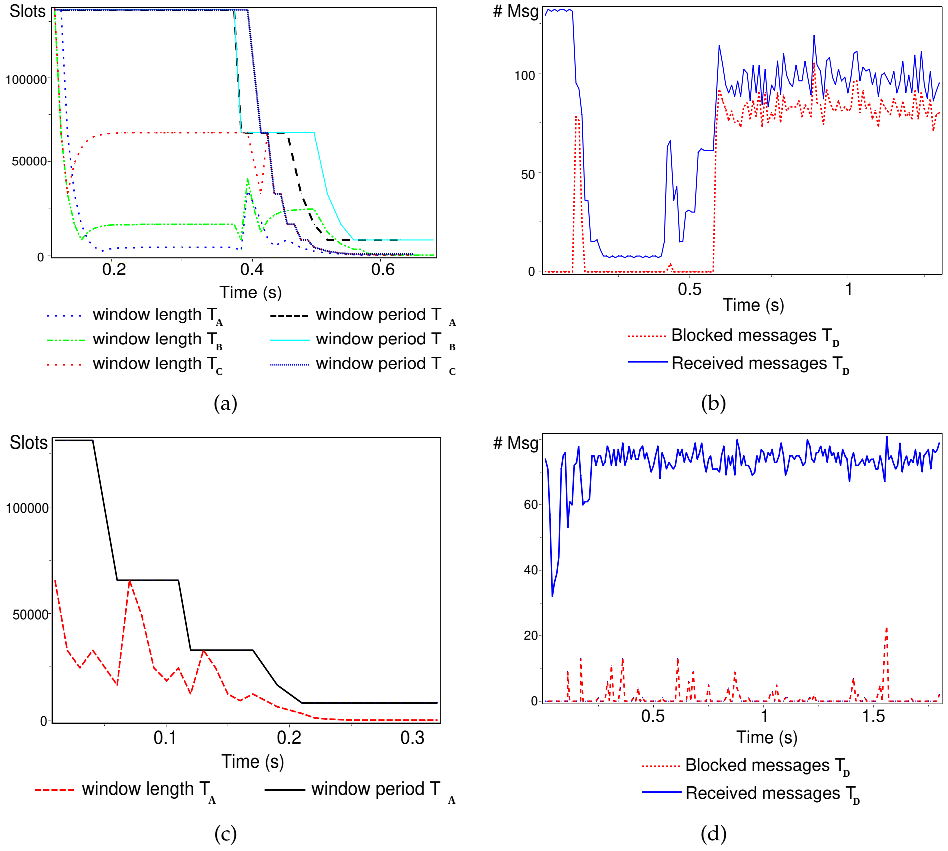
By analyzing the task-graph of the application, only the send window of a T_A is altered (Figure 5.21(c)). Because this window change ripples through the pipeline, blocking of task T_D can be handled (Figure 5.21(d)).

5.5 Related Work

Advanced NoCs such as Æthereal provide hardware support for end-to-end credit-based flow-control at the level of network interfaces [189]. The purpose of this end-to-end flow control is to ensure that guaranteed packets can be timely removed from the network layer. Failing this, the flow-control mechanisms at the network and data-link layers would be used to block packets and congestion would potentially build-up disrupting guarantees. Our approach to global connection-level management provides a similar flow-control but at coarse granularity. On the one hand the coarse granularity of flow-control cannot guarantee that individual packets do not get blocked at the network interface layer, so this flow-control should not be used to provide hard real-time guarantees. But, on the other hand the same mechanisms (monitoring, actuating and part of decision making) can be reused to also perform congestion-control (this requires enhancing the decision-making mechanism to correlate statistics of several connections).

Congestion control is a topic that has been extensively studied for computer networks, but it is only very recently that is has started to be addressed in networks-on-chip. In the Internet, one of the most known congestion-control mechanisms is the sliding-window algorithm implemented in TCP [11]. In terms of traffic shaping, two very known algorithms extensively used in computer networks are *leaky bucket* and *token bucket* [218]. These two algorithms correspond to classes of actuators in our definition.

The token bucket is a non-work-conserving algorithm and as such allows a certain level of traffic burstiness. The algorithm simply controls when packets can be injected in the network. At a rate of $1/r$ a token is added to a bucket (a simple counter of maximum size b). Before sending a packet of size $n < b$, there must be at least n tokens available in the bucket (and they are consumed when the packet is sent).



(a) (b) (c) (d)

Figure 5.21: *Pipeline window management experimental results. (a) Setting windows for all tasks in the pipeline and (b) the resulting throughput and blocking for task T_D. (c) Only setting the window for task T_A. and (d) the respective resulting throughput and blocking for task T_D.*

When not enough tokens are available, the packet is either buffered or discarded, depending on the policy implemented.

The leaky bucket algorithm is a related traffic-shaping algorithm, that is typically used to smooth traffic bursts. The algorithm can be understood as a bucket (of size B bytes) in which different streams converge. The bottom of the bucket has a hole that permits a constant flow rate to get out of the bucket (thereby smoothing traffic bursts). Leaky bucket belongs to the work-conserving class of algorithms because it always serves packets when there are any available.

In network-on-chips, most related work pertains to congestion avoidance rather than to congestion control. To avoid congestion, packets either take different routes (adaptive routing) to spread the traffic and reduce the creation of hot-spots, or they may even be dropped. The Nostrum NoC implements a form of adaptive routing, called *hot-potato* routing [145] and allows packets to be dropped [153]. In hot-potato routing, packets can be deflected (onto random output ports) from the optimal path if it is deemed congested. To avoid creation of hot-spots, routers send backpressure signals to notify their neighbors of congestion ahead of sending packets. This technique is called Proximity Congestion Awareness [152]. The creation of hot-spots is thus diminished, but some packets take longer paths than the deterministic shortest-path and thus arrive out-of-order at destination. The DyAD algorithm is another known form of congestion avoidance algorithm [100]. An NoC implementing the DyAD routing algorithm performs deterministic shortest-path routing. Upon detection of congestion, the routing switches to an adaptive form to take less congested paths. Though congestion avoidance algorithms, do reduce congestion on an average, they cannot guarantee it is removed. Moreover out-of-order delivery or packet dropping are expensive congestion control techniques for NoCs because they require large reordering or re-transmission buffers.

In terms of monitoring mechanisms for NoC, Ciordas has recently proposed to extend the Æthereal NoC with an event-based monitoring service [42]. The original purpose of this mechanism is to offer run-time observability of NoC behavior mainly to support system-level debugging, but can also be used for performance analysis. Hardware probes are attached to NoC components with a target area overhead for monitoring and control of 15 to 20%. The monitoring system requires an increase in the arity of the router as monitoring signals are sent to an additional port onto the local network interface. The router area is accordingly increased from $0.11mm^2$ to $0.13mm^2$ ($0.13\mu m$ standard cell technology). Monitoring packets can be sent onto the NoC either as BE or as GT packets. The GT QoS class is preferred for this purpose as it permits a deterministic rate of monitoring information, but it may require increasing the number of time-slots of the NoC for the additional GT connections and thus create an additional area penalty.

Only very recently have other researchers addressed congestion-control in NoCs. Ogras and Marculescu propose a novel data-link level flow-control technique to reduce congestion in NoCs [164]. The authors build a model of the neighboring routers to predict whether congestion could occur. When congestion is predicted, the flow-control is used to limit the injection of packets into the next router and thereby reduce its congestion. Another very interesting approach is described by Van den Brand in [223]. Building on top of the link-level monitoring mechanism proposed by Ciordas [42], the authors build a model predictive control of the NoC as a decision-making process. Though the actuators in the system are not explicitly mentioned in the article, we presume that the network interfaces are reconfigured to control how much traffic is injected into the network layer. The latency of the congestion-control system is reported to be in the order of magnitude of microseconds. Monitoring information uses GT connections and is reported to have a low bandwidth usage (0.3 MBytes/s) when sampling at 1000 ns.

5.6 Conclusions and Future Work

This chapter covers techniques for reactive communication control so as to answer two questions. On the one hand the communication of applications designed independently of the platform may have to be dynamically matched to the platform resources. On the other hand, platform virtualization that perfectly isolates applications from one another can be ineffective. It can be too expensive to implement, and/or the requirements of the application are only imperfect worst-case estimates resulting into over-allocated (and thus wasted) communication resources.

To perform reactive communication control, three components are required: monitoring of NoC usage, actuators to shape the traffic and decision-making processes to control the actuators. A control loop is built feeding the output of the monitoring into the decision-making process and using its output to control the traffic shaping. All three components can be defined at either of the layers in the network protocol stack, creating a large design space. Using the $Gecko^2$ MPSoC platform emulator as an experimental platform, we fix an actuator (injection rate control) at the network interface layer because it is the source of network traffic.

In this chapter, we study monitoring at the connection-level combined to a global decision making under the control of a run-time manager. We show how a global decision making algorithm implemented on a StrongARM processor can efficiently manage the injection rate controller actuators in order to re-shape traffic at run-time to reduce congestion on an NoC. Taking a global decision-making approach permits an efficient reactive control of communication of a pipeline of tasks. In contrast, Marescaux [132] studies a system, based on the same experimental setup, where monitoring is performed at the link-level, combined with distributed decision making.

We show that reactive communication management is a beneficial addition to MPSoC platforms and demonstrate how it can be used by exploring two extreme points of the design space (i.e. one point is detailed in this thesis, another point is studied by Marescaux [132]). The centralized algorithm should be used to optimally adapt the parameters of the actuators for slowly changing traffic characteristics, whereas the distributed control of traffic shaping should be used to cope with fast transient changes in traffic. Nevertheless, as many combinations and trade-offs are possible, the design space of reactive communication control is very large and worth exploring. Future work could start by combining both central and distributed approaches. For instance it would be very interesting to have the global controller fine-tune the parameters of the window size algorithm (such as the values of k and l controlling the aggressiveness of the window) and let the distributed mechanism control the actuators for faster reaction time.

CHAPTER 6

Gecko Proof-of-Concept Platforms

The work on the Gecko series of platform emulators started at IMEC in 2001 as an ambitious engineering project to explore, in real-time, NoC-enabled multiprocessor SoCs with heterogeneous, run-time reconfigurable, computing resources. It was only natural to include run-time management of the platform, with a particular focus on the NoC, on dynamic partial reconfiguration of the computing resources, and on task migration between heterogeneous computing resources.

The first demonstrator, later denoted as *Gecko I* , has been first publicly shown at the 2002 Design Automation Conference (DAC). By October 2003, the second (and final) generation of Gecko emulators, dubbed *Gecko*^{2} (read: Gecko square) has been publicly introduced at the IMEC Annual Research Review Meeting (ARRM) and later demonstrated at the occasion of major international conferences. *Gecko*^{2} is an elaborate NoC-enabled MPSoC platform emulator, with hierarchically reconfigurable computing resources, under the control of a run-time manager.

This chapter describes the hardware of the *Gecko* platform, with the main focus on the *Gecko*^{2} instantiation. It brings a concrete demonstration of the reactive communication control concepts developed in Chapter 5 and extends them with a proof-of-concept of a centrally controlled congestion-control and of dynamic re-routing. It furthermore introduces the exotic concept of hierarchical hardware reconfiguration (Chapter 3) and demonstrates it on a realistic emulation platform. Though, the main contribution of *Gecko* is maybe to have served as an initiatory journey through technical and sometimes less technical matters, earning it a warm place in the middle of the other chapters, rather than in an appendix. This chapter is also part of the PhD thesis of Théodore Marescaux [132].

Finally, the reconfigurable computing demonstrator developed within the IWT RESUME project is briefly detailed, especially focusing on run-time resource management aspect. It shows how the developed run-time resource management concepts incorporated into the Gecko demonstrator series are easily portable onto another embedded platform and base operating system.

The chapter is organized as follows. Section 6.1 describes the emulator platform and details the *Gecko I* and the *Gecko*^{2} hardware instantiation. Section 6.2 discusses the driver applications and the envisaged usage-scenario of a run-time reconfigurable MPSoC platform. Section 6.3 discusses the run-time manager controlling the MPSoC platform and briefly refreshes the hierarchical reconfiguration concept. Section 6.4 is a textual transcript of the proof-of-concept demonstrations running on the *Gecko I* and the *Gecko*^{2} emulator. It explains the demonstration of heterogeneous task migration, hierarchical reconfiguration, and management of inter-task communication interferences (including reactive communication control of the data NoC). We also briefly discuss the RESUME demonstrator run-time manager and its differences with respect to the Gecko run-time manager (Section 6.4.3). Finally section 6.5 concludes.

6.1 *Gecko* Platform Hardware

The *Gecko* series of platforms are based on the same hardware that connects a StrongARM processor, present inside a Compaq iPAQ PDA, to an FPGA (Xilinx Virtex-II 6000) by means of the iPAQ expansion port (direct access to the StrongARM processor bus). The FPGA is clocked at 33 MHz, and the StrongARM SA-1110 processor, present in the PDA, is clocked at 206 MHz. This setup is illustrated by Figure 6.1.

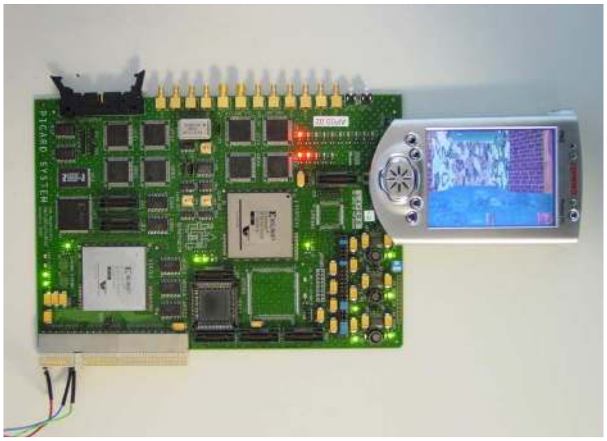


Figure 6.1: *Gecko is built by linking a Compaq iPAQ PDA to an IMEC in-house FPGA board containing a Virtex-II 6000 FPGA.*

For *Gecko I* , this setup represents a MPSoC platform containing a StrongARM tile and two large run-time reconfigurable FPGA fabric tiles interconnected by a simple NoC in a torus topology (Figure 6.2). The routers of this NoC support source routing and wormhole packet switching. Marescaux et al. [134] provide an in-depth description of the Gecko hardware concepts.

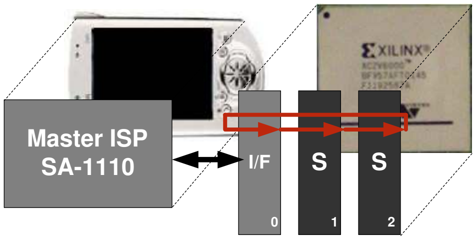
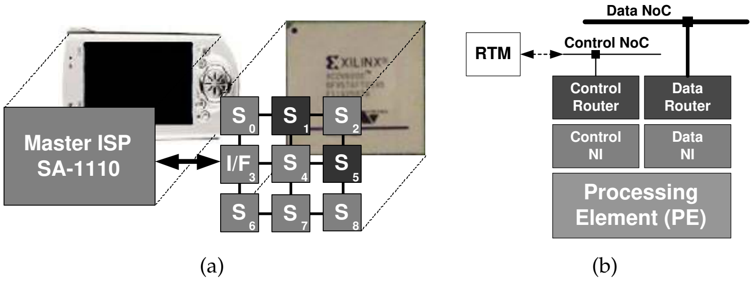


Figure 6.2: *The Gecko I MPSoC contains a StrongARM tile and two large FPGA tiles interconnected by a wormhole-switched, input buffered, unidirectional torus NoC.*

The *Gecko*^{2} multiprocessor system is a more elaborate MPSoC emulated on the same hardware setup (Figure 6.3(a)). In this configuration, the FPGA is divided into eight slave processor tiles. It furthermore contains a master ISP tile that interfaces to the external StrongARM processor. All nine tiles are interconnected by two networks-on-chip also emulated on the FPGA. One NoC is used for application data and the other for platform monitoring and control. Slave tiles 1 and 5 are fine-grain reconfigurable hardware slave processors, meaning that they expose the FPGA fabric. The other slave tiles either contain an accelerator or a simple 16-bit RISC processor (see Figure 6.9 and Table 6.1).



(a) (b)

Figure 6.3: *(a) Our heterogeneous MPSoC is emulated by coupling an ISP (master) through an interface (I/F) with the slave processors (S), instantiated inside an FPGA. (b) One Tile: the Data NI and the Control NI connect the computing resource to the data NoC router and the control NoC router respectively.*

Our packet-switched NoC, called *data NoC*, is instantiated as a 3x3 bidirectional mesh and is responsible for delivering data packets for tasks executing on the PEs. A second NoC, the *control NoC*, is used for run-time control messages[1] [135] (Figure

[1] The control NoC is implemented as a single-master central shared bus. There are two reasons to this technical choice. On the one hand we needed visibility in the platform while debugging the data NoC

6.3(b)). Separation of data and control NoCs ensures that application data circulating on the *data NoC* does not interfere with run-time control messages. Some commercial NoCs have since taken a similar approach. For instance, the *configuration bus* of the Arteris NoC has a very similar purpose as our control network.

The following sections describe in more detail the most important platform hardware components of the *Gecko*^{2} architecture, i.e. the NoC data router, the data NI and the control NI (Figure 6.3(b)).

6.1.1 NoC Data Router Design

The data NoC [21] is a packet-switched network. The data routers in the network use *virtual cut-through* (VCT) switching and output buffering.

Virtual cut-through switching ensures a low communication latency: pieces of packets, called *flits*, are forwarded as soon as the output channel is free, as happens in wormhole switching. Upon congestion, VCT, unlike wormhole switching, allows the buffering of complete packets inside the routers and thus ensures a low latency by leaving the crossbar free for other packets.

Virtual cut-through achieves, at low traffic, the same low latency as wormhole switching, paired with the same high throughput at high loads as store and forward.Because in our system the size of the payload is relatively large (limited to 544 bytes for run-time management efficiency reasons [144]) a wormhole switching network would saturate much faster than the virtual cut-through one. The drawback is the large buffer size required to store the packets. On an FPGA implementation, buffer size is less of a problem thanks to the richness of the device in embedded Block RAMs.

Packets need to be stored (i.e. buffered) in case of blocking. There are several different buffering strategies. The buffers can be placed before the crossbar switch, denoted as *input queuing*, *centrally*, or after the switch, denoted as *output queuing*. Our routers use output queuing and there is one buffer per output block (Figure 6.4). Thanks to output queuing, we avoid the *head of line blocking* effect which occurs in input queued routers [66]. As our routers are output buffered, packets are stored on the outputs, leaving (upon blocking) the input port free to serve other packets with different output destinations and hence reduce overall NoC latency.

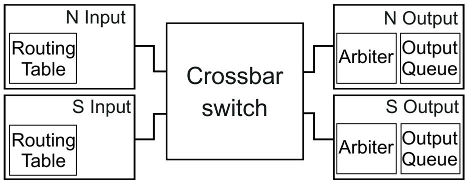


Figure 6.4: *Structure of a 2-input, 2-output Gecko*^{2} *router.*

(probably one of the very first complete NoC implementations ever) and on the other hand it fitted the low-latency requirements of the short control messages of the run-time manager running on the StrongARM. Nevertheless, the communication scheme on this bus is packet-based message-passing, making it easy to change to a packet-switched fabric with only minor changes to the Control NI.

We implement a deterministic routing algorithm based on a run-time management configurable lookup table. The table has an entry for every tile in the network, offering the possibility to customize the routing for each tile at the network level. Deterministic routing on the $Gecko^2$ platform guarantees in-order packet delivery, hence avoiding the packet reordering overhead required by adaptive routing NoCs.

The design is optimized for performance: every input block (buffer and link controller) has a routing table (Figure 6.4), and every output block an arbiter and a buffer. This solution provides the shortest possible latency at the cost of silicon area (minimally 3 cycles are required to transmit the first flit of a packet across a hop). A packet traveling 5 hops has a base latency of 15 cycles plus a number of cycles equal to the number of packet flits. For a large packet the delay to receive the first flit is only a small fraction of the time needed to receive the entire packet. Once the first flit arrives the IP can start processing the data right away, the next flits will follow every clock cycle (assuming no waiting in the network). The overhead delay incurred by the routing through the network is kept to a minimum.

To save area the routing table and the arbiter could be shared by introducing a mechanism to serialize the input or output accesses to them. However, for routers with a small number of ports, duplicating the routing table provides a better performance/area trade-off. On Virtex FPGAs this table is very efficiently implemented using distributed dual-port selectRAMs. A router requires $2 * No_{InPorts} * No_{OutPorts}$ LUTs (every input-block requires an entry for every output port; and two LUTs per entry).

6.1.2 Data Network Interfaces

The computing resources of our MPSoC are interfaced to the packet-switched data NoC by means of a data *Network Interface* (NI). From the computing resource viewpoint the main role of the data NI is to buffer input and output messages and to provide a high-level interface to the data router (Figure 6.5). From a run-time management viewpoint, the data NI is responsible for collecting message statistics (i.e. number of messages received, sent and blocked) and for enforcing the injection rate window. All this information is passed to the run-time manager through the control NI (explained in Section 6.1.3).

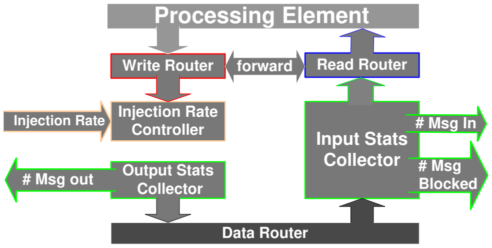


Figure 6.5: *$Gecko^2$ data NI.*

At the system layer, inter-task communication is done by message passing on a socket input/output port basis. Figure 6.6 shows an example of an application task graph with the input/output port connections between tasks. Each application registers its task graph with the run-time manager upon initialization [134, 156].

For each task in the application, the run-time manager assigns a system-wide unique logic address and places the task on the platform, which determines its physical address (Figure 6.7). For every output port of a task the run-time manager defines a triplet *(destination input port, destination logic address, destination physical address)*. The *destination physical address* denotes the tile of the communication peer for this output port. The *destination logic address* enables the destination tile to determine the destination task in case multiple tasks are present on that tile. Finally, the *destination input port* denotes the input port at which the message should be stored.

For instance, task C in Figure 6.6 communicates with task D on output port 0 and with task E on output port 1. Hence task C is assigned two triplets, which compose its Destination Lookup Table (DLT) (Figure 6.7). In our system a task may have up to 16 output ports, thus there are 16 entries in a DLT. The run-time manager can change the DLT at run-time, by sending a control message on the Control Network.

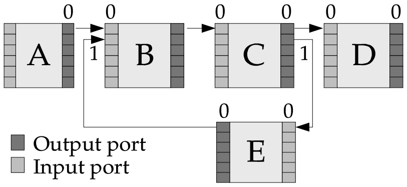
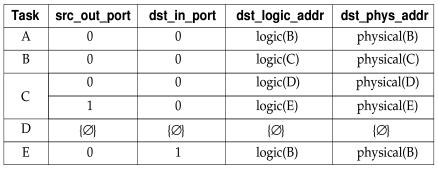


Figure 6.6: *Application Task Graph showing Input-Output port connections.*



Task	src_out_port	dst_in_port	dst_logic_addr	dst_phys_addr
A	0	0	logic(B)	physical(B)
B	0	0	logic(C)	physical(C)
C	0	0	logic(D)	physical(D)
	1	0	logic(E)	physical(E)
D	{∅}	{∅}	{∅}	{∅}
E	0	1	logic(B)	physical(B)

Figure 6.7: *Destination Look-up Tables for every task in the graph.*

The data NI is also responsible for collecting the local computing resource message statistics. This involves keeping track of the number of messages sent, received and blocked. The *blocked message count* denotes the number of received messages, that were blocked in the data router buffer while waiting for the computing resource input buffer to be released. Moreover, as chapter 5 details the data NI implements an injection rate control mechanism to perform traffic shaping by allowing control of the amount of messages the attached computing resource injects in the data NoC per unit of time [135, 162].

6.1.3 Control Network Interfaces

The control network is used by the operating system to control the behavior of the complete system. It allows data monitoring, debugging, control of the IP block, exception handling, etc. Run-time management control messages are short, but must be delivered fast. We therefore need a low bandwidth, low latency control network.

To limit resource usage and minimize latency we decided to implement the control network as a shared bus, where the run-time manager running on the ISP is the only master and all control network NIs of tiles are slaves. The communication on this bus is message-based and can therefore be replaced by any type of NoC.

The control NI of every tile is memory-mapped in the ISP. To send a control message to a tile, the run-time manager first writes the payload data, such as the contents of a DLT (Figure 6.7) and finishes by writing a command code on the control network, in this case an *UPDATE_DLT* command. The control NI reads the command opcode and processes it. When done, it writes a status opcode in the NI to NoC memory, to indicate whether the command was successfully processed and posts an interrupt. The run-time manager retrieves this data and clears the interrupt to acknowledge the end of command processing.

It is in the control NI, that statistics and exceptions originating in the data NI are processed and communicated to the run-time manager. It is also through the control NI that the run-time manager sends destination look-up tables or injection-rate windows to the data NI. The control NI is also responsible for handling the data synchronization mechanism used for e.g. task migration.

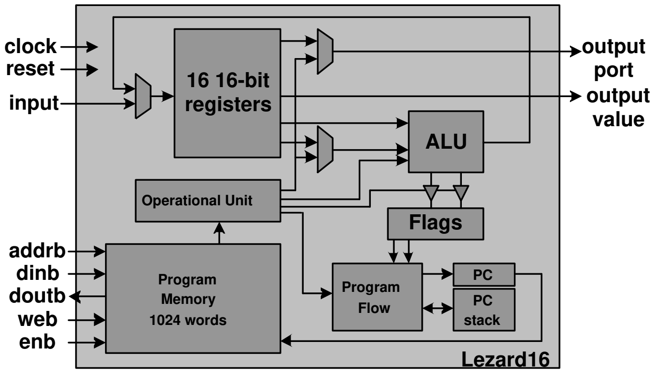


Figure 6.8: *The Lezard16 processor is at the heart of the Gecko^2 NI.*

The heart of the control NI contains a Lezard16 ISP (Figure 6.8). This in-house developed softcore ISP is based on the instruction set of the Xilinx PicoBlaze 8-bit processor [97]. The Lezard16 is a 16-bit processor with 18-bit instruction words. The Lezard16 instruction set is similar to the one of the PicoBlaze, except that the Lezard16 is not able to handle interrupts. Furthermore, it features a 1024 instruction word deep program memory, as opposed to the 256 instruction word memory of the PicoBlaze. The program memory is implemented as a dual-port memory, allowing the program code to be updated through the second access port. The size of the 18-bit instructions perfectly match the width of the Virtex2's Block RAMs. By extending the Picoblaze's data-path of 8-bits to 16-bits while only adding 2 more bits for instructions, the loading of immediate constants is limited to 10-bit numbers. An in-house assembler, developed using the GNU Bison and GNU Flex tools, facilitates software development. The Lezard16 firmware provided with the FPGA configuration bitstream contains bootstrap code to load the code to run in the control NI at platform boot-time. This way, the control NI functionality can easily be updated.

6.1.4 Hardware Implementation Results

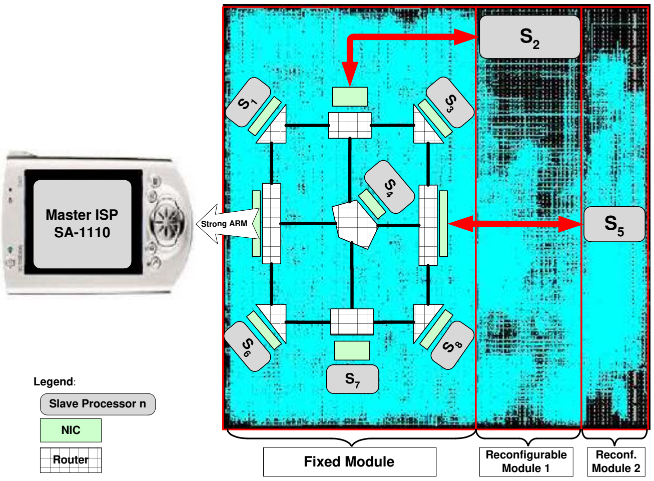


Figure 6.9: *Gecko^2 emulation floorplan on a Xilinx XC2V6000 FPGA.*

The second generation of Gecko demonstrators (*Gecko*^{2}) emulates a control bus and a 3x3 data bidirectional mesh network that connects 9 heterogeneous processors. The StrongARM processor on the iPaq is used as the master CPU of our system, whereas the 8 slave processors and the data and control NoCs are emulated on the FPGA (Fig. 6.9). Various configurations have been generated; the one typically demonstrated contains: a simple 16-bit RISC processor (denoted *Lezard16*) on tiles S_1, S_3, S_4, S_6 and S_7; fine-grain reconfigurable processors on tiles S_2 and S_5, a $2_D IDCT$ accelerator on tile S_8. As Table 6.1 explains, various tasks have been implemented to run on one or more of these heterogeneous processors. Some task implementations, such as the Lezard16 and the convolution filter are programmable machines. They are therefore also considered as processors when instantiated in reconfigurable hardware. These processors support demonstrating the hierarchical configuration concept [158, 161].

The driver applications (Section 6.2.2) can concurrently run on the platform and HW/SW trade-offs can be explored at run-time by dynamically changing task locations on the heterogeneous processor system.

	Task Implementations					
Processors	*Huffman Decoder*	$2_D IDCT$	*3D Text. Mapper*	*Lezard16 Processor*	*Conv. Filter*	*Edge Detect*
SA-1110	√	√	√		√	√
Reconf. S_2, S_5	√	√	√	√	√	√
Lezard16					√	√
Conv. Filter					√	√
Fixed HW		√		√		

Table 6.1: *Implementations of various HW/SW tasks on Gecko^2 heterogeneous processors*

The floorplanning of the *Gecko*^{2} system follows Xilinx' Modular Design [125] technique to isolate reconfigurable computing modules from communication modules through NoC interfaces [134] so as to enable dynamic partial reconfiguration. The FPGA is divided into 3 independently reconfigurable modules (Figure 6.9).

Fixed Module contains the data and control NoCs, their interfaces, as well as all fixed slave processors ($S_1, S_3, S_4, S_6, S_7, S_8$) and the fixed interface to the external Master processor.

Reconfigurable Module 1 contains raw Virtex 2 logic, block-RAMs and multipliers. It connects to the Fixed Module on data and control NoCs as slave processor S_2 and contains route-through wires to connect the Fixed Node to Reconfigurable Module 2.

Reconfigurable Module 2 contains raw Virtex 2 logic, block-RAMs and multipliers. It connects to the Fixed Module on data and control NoCs as slave processor S_5.

6.2 *Gecko* Applications

This section details the envisioned *Gecko* application scenario and details the applications that have driven *Gecko* research and development.

6.2.1 Envisioned Application Scenario

Run-time task migration can be exploited to optimize resource usage on an heterogeneous reconfigurable platform. The following use case scenario, depicted in Figure 6.10 demonstrates the need of task migration in the case of mobile multimedia terminals.

A user is watching a movie on the handheld multimedia terminal (1). As the movie originates e.g. from a broadcast public TV stream [177], it is occasionally interrupted for commercials. When this occurs, the user wants to take advantage of that time to execute another application, while keeping an eye on the video window in order to resume watching the movie whenever the advertisements are finished. The video window is thus downsized in the corner of the screen (2), while another application (e.g. a 3D game) is started on the terminal. Figure 6.10 also shows the behavior of the platform corresponding to the use case scenario. The platform is composed of a set of flexible computing resources (ISPs, reconfigurable hardware, etc.). Initially, the movie player is the only application requiring the platform resources. Hence the video decoder can use all needed resources, effectively enabling it to run at full resolution and at full frame rate (1). When the user downsizes the video window, both resolution and frame rate can be reduced. Consequently, the video decoder tasks can be migrated to fewer (cheaper) computing resources[2] (2). The resources

[2] The computing resources for scenario (2) can differ from those used in scenario (1). For instance in a full resolution, full frame-rate scenario, hardware acceleration such as IDCT could be performed on a specialized module running on FPGA fabric. Whereas in scenario (2) the IDCT functionality could be migrated to a software block running on an ISP.

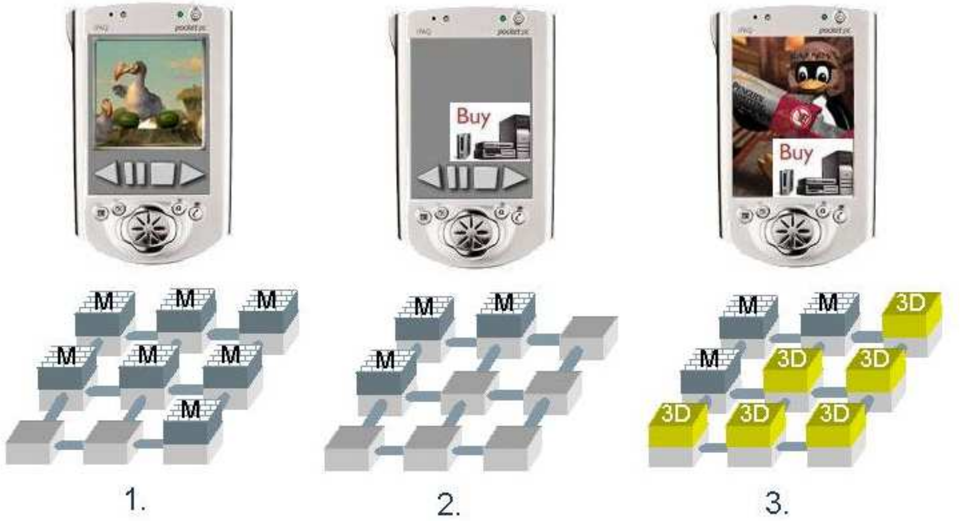


Figure 6.10: *Multimedia applications scenario. 1. The user is watching a movie - the video decoder (M) executes using as much hardware resources as it can; 2. During advertisements, the user downsizes the video window - the decoder is therefore relocated on fewer resources; 3. The user starts a 3D game (3D) - the 3D engine uses the hardware resources made available.*

that are no longer used can be made available to the 3D engine (3) that is required by the game.

The user interaction on the terminal thus creates dynamism that affects the mapping of the applications on the available resources. This dynamism is one of the reasons for having flexible, yet computing intensive (multimedia application are computation hungry) resources on the platform, i.e. reconfigurable hardware. It also clearly justifies the opportunities of run-time task migration capabilities.

6.2.2 Driver Applications

This section details the three *Gecko* driver applications: an edge detection application, an MJPEG video decoding application and a 3D game.

Edge Detection

The *Laplace edge detection* application is a simple, but compute-intensive filtering application. The algorithm applied is a 2D Laplace edge detector that is based on the convolution sum with a (4-neighborhood) 3×3 convolution kernel with the following coefficients:

$$\nabla^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The application is composed of three tasks. The first one cuts the image into smaller 10 by 10 pixel blocks and sends them to the second task for a 2D filtering operation. Consequently, the second task returns the filtered blocks for the third task to insert into the completed filtered image.

Several pure hardware, pure software or hardware/software versions of the application have been implemented (Table 6.1) and serve both as test applications and as a proof-of-concept for the hierarchical configuration concept [158, 161].

MJPEG Video

The *Motion JPEG* (MJPEG) video decoder [144, 162] is composed of four pipelined tasks (Figure 6.11(a)). Task T_1 is responsible for reading the video-stream from disk and to send it to task T_2. T_2 performs *Huffman decoding* and *dequantization*. Consequently, T_3 performs a 2D IDCT function and a YUV to RGB color conversion. Finally, task T_4 constructs and displays the output images. T_1 and T_4 are software only tasks, while T_2 and T_3 are HW/SW tasks designed using the OCAPI-XL design flow [144, 169, 196, 199, 227]. These HW/SW tasks can be migrated from a high performance FPGA tile to an ISP tile and vice versa.

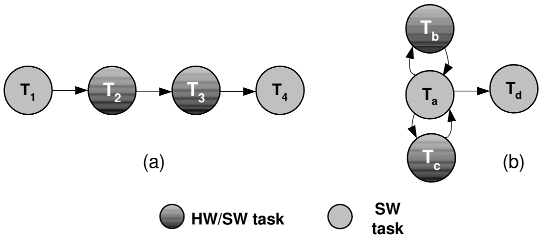


Figure 6.11: Driver application task graphs containing both HW and HW/SW tasks. (a) Task graph of the MJPEG video application. (b) Task graph of the 3D Game application.

Shoot'em Up 3D Game

The 3D game is a simplified *first-person shoot 'em up* game [151]. The aim of the game is to shoot the targets on the walls that compose the 3D scene. The wet bottom of the 3D scene contains a water rippling effect. This rippling effect increases the scene realism and increases computational requirements.

The application task graph contains four tasks (Figure 6.11(b)). Task T_a is responsible for 3D scene transformations, while task T_b and task T_c are responsible for the texture mapping and water rippling effect respectively. Finally, task T_d is responsible for displaying the 3D scene. Task T_a and task T_d are software only tasks implemented in

C, while task T_b and task T_c are HW/SW tasks. Similar to the MJPEG HW/SW tasks, these tasks have both a C and a VHDL implementation. Both implementations are generated from a unified OCAPI-XL application representation [150].

6.3 Run-Time Resource Management

This section covers all run-time management components of the *Gecko I* and *Gecko*^{2} demonstrator and illustrates how the run-time management algorithms are functioning inside real-life proof-of-concept demonstrators. In that sense, Section 6.3.1 details how the run-time resource manager extends an existing RTOS and how this fits into the run-time management implementation space. Section 6.3.2 details the different steps the Gecko run-time manager takes when the user starts an application. This includes showing the role of the quality manager and the resource manager. Section 6.3.3 details the mechanism responsible for handling a configuration hierarchy.

6.3.1 Extending an Existing RTOS

Instead of starting from scratch, we decided to extend an existing RTOS. For *Gecko I* we used RTLinux as base RTOS, while for *Gecko*^{2}, we used its *GNU General Public License* (GPL) counterpart RTAI.[3] The main reasons for this approach is that support for regular software tasks is already present and that it enables finalizing the demonstrator in a short amount of time with only limited resources. It also illustrates how to extend an existing operating system, which improves portability of run-time manager components to other platforms/operating systems.

According to the run-time management implementation space, detailed in Chapter 2, the created operating system can be classified as a *Master-Slave configuration* (Figure 6.12). This implies that one processor unit (the master) is responsible for monitoring the status of the system and for assigning work to all the other processor units (the slaves). The run-time library functionality is provided by a Lezard16 processor (Section 6.1.3) instantiated in every tile control NI. In order to avoid the master becoming a bottleneck, the platform monitoring and the enforcement of decisions are handled by the run-time libraries.

The operating system keeps track of the applications by maintaining a list of task graphs. For every task, the operating system creates a *task information structure*. The task information structure retains all information needed for the run-time manager to make its decisions.

The operating system manages its computing resources by using a *processor information structure* for every PE in the system. In addition, the operating system keeps track of the interconnection between these PEs, as this will be important information when making resource assignment decisions. In addition to hard (i.e. fixed) PEs, the operating system also employs a processor information structure for every *softcore*.

The operating system also maintains a *communication structure* to keep track of the platform communication information. This includes the topology of the on-chip in-

[3] RTAI stands for Real-Time Application Interface.

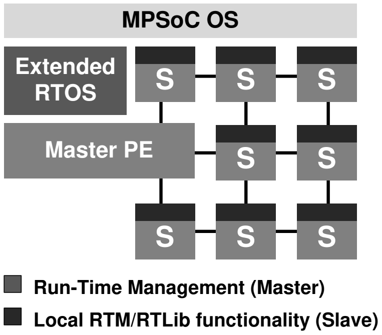

MPSoC OS
Extended RTOS
Master PE
S
S
S
S
S
S
S
S
▨ Run-Time Management (Master)
■ Local RTM/RTLib functionality (Slave)

Figure 6.12: *The overall MPSoC operating system can be classified as a master-slave operating system. The MPSoC operating system is created by extending an existing RTOS that executes on the master PE. Every slave PE has run-time library functionality.*

terconnect, the routing tables of every router, the destination lookup table of every tile as well as the state of the tile injection rate controllers. The close interaction between the master run-time manager and the (slave) run-time management, i.e. the run-time library functionality, resembles classic *remote procedure calling* (RPC). The master maintains for each control NI a structure that describes its functionality and that allows the master to remotely execute a function on a slave node. So the control NI structure in the master run-time manager can be seen as the RPC stub.

Figure 6.13 illustrates how the slave run-time library functionality is used. First of all, the master makes a function call to the respective control network interface component (control NI) stub (1). This stub translates the call into a control message containing the desired function number and its required parameters. Consequently, this message is sent to the slave node (2). Once the message is received on the slave node (3), its function number and parameters are unwrapped and the respective local run-time manager executes the required function at the slave node (4). The return value (5) is packed into a message (6), sent over the control network to the control NI stub, where it is unpacked (7). Finally, the original master run-time management function call returns with the respective return value (8).

Certain network events (such as synchronization upon flushing the buffers in the network layer) require action from the master run-time manager. In such a case, the slave node triggers an interrupt to initiate a function call to the master using the same mechanism.

6.3.2 Flow for Starting an Application

Figure 6.14 details the operating system flow used for starting an application on the $Gecko^2$ platform. The flow contains three distinct parts: application registration (1), the actual run-time management (2) and the management mechanisms (3).

When the user starts an application (1), the operating system parses the application binary and extracts the application task graph and its properties. Consequently, a set of task structures is created and the binaries for every task's supported PEs are

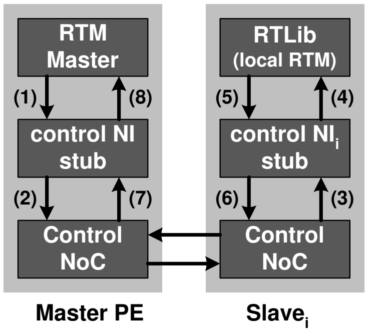


Figure 6.13: *Communication between master run-time manager and the RTLib (i.e. tile-local run-time manager) functionality.*

registered. The next step is to start the run-time manager in order to find and allocate the required resources to execute this application.

The run-time management components of the second part (2) represents the situation where an application consists of a *single task graph*, i.e. contains only a single application operating point. This means that the quality management is, in this case, not responsible for selecting the right operating point, but merely for determining the required PE and communication resources with respect to the task graph and the user requirements. After determining the load properties of the task graph, the resource manager can start assigning resources taking the current resource usage of the platform (i.e. the already executing application) into account. Once resources assignment decisions have been taken, they need to be enforced. This is the responsibility of the resource management mechanisms.

The resource management mechanism (3) is responsible for setting up the task on its assigned processing element and for setting up and initializing the inter-task communication structures. In that sense, the resource assignment mechanism starts by loading the task binaries to their assigned processing elements and, consequently, initializes the tasks with e.g. initial parameter values. Then, the communication structures are initialized. This involves setting up a *DLT* on every tile according to the task graph and the location of the communication peers. This also entails setting up the right task *send windows*. As Chapter 5 details, these send windows enable bandwidth management to reduce inter-task communication interference. Finally, all tasks belonging to the application are started (i.e. set as runnable).

6.3.3 Supporting a Configuration Hierarchy

When setting up a task on a softcore, allocating resources to the assigned task involves a little more work. First, the PE management structures of the softcore(s) and the respective host PE(s) need to be linked. This linking allows a softcore PE to use the services provided by the lower layer, like message-passing over the NoC. This successive linking associated with *allocate PE* is detailed in Figure 6.15(a). This results in a configuration hierarchy (Figure 6.15(b)).

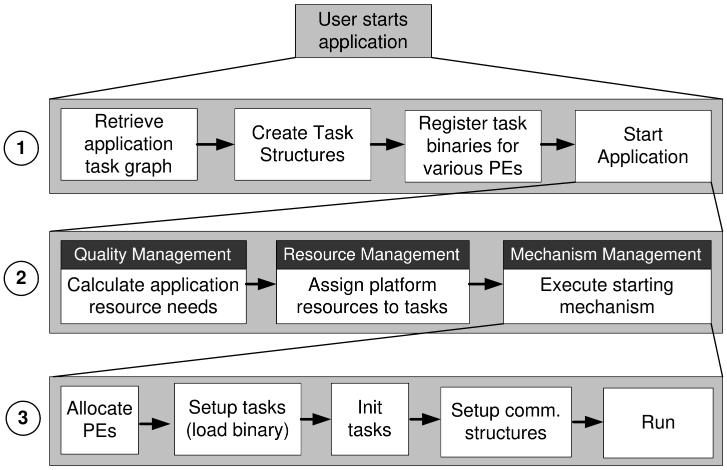

User starts application
1
Retrieve application task graph
Create Task Structures
Register task binaries for various PEs
Start Application
2
Quality Management
Calculate application resource needs
Resource Management
Assign platform resources to tasks
Mechanism Management
Execute starting mechanism
3
Allocate PEs
Setup tasks (load binary)
Init tasks
Setup comm. structures
Run

Figure 6.14: *Operating System flow with run-time management components for starting an application on the Gecko^2 platform. (1) The application task-graph is loaded from the application binary, task structures are created to represent the application threads and the representations of the various binaries for one given task. (2) The run-time manager assigns communication, computation and storage resources and executes the application starting mechanism. (3) Binaries are loaded on their respective tasks (hardware binaries are sent through the FPGA reconfiguration bus and software binaries are sent over the data NoC), tasks are initialized with eventual data, the communication infrastructure is reconfigured and the application is started.*

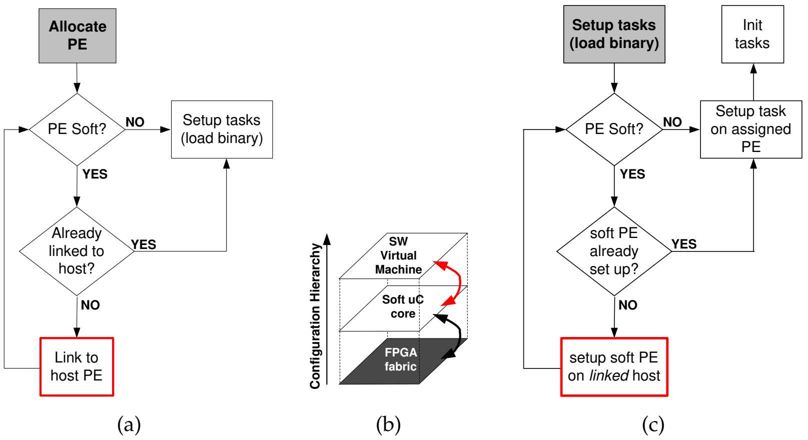
When setting up a task on a soft PE, one first needs to set up the underlying softcore PE. Only after setting up the configuration hierarchy, the task can be setup on top of the softcore. If the softcore is already instantiated, it does not need to be reinstantiated. In that sense, a soft IP core is just *a task* to the underlying host. A similar approach is taken when initializing the task.

6.4 Proof-of-Concept Demos

This section first details the demonstrations running on the *Gecko I* and the *Gecko*^{2} emulator. It shows the heterogeneous task migration concept (Section 6.4.1). It also illustrates the hierarchical reconfiguration and the management of inter-task communication interferences (Section 6.4.2). Secondly, this section briefly introduces the RESUME demonstrator.

6.4.1 *Gecko I*

The *Gecko I* demonstrator illustrates the feasibility of the envisioned application scenario detailed in Section 6.2.1. Both the MJPEG Video application and the 3D game are simultaneously executing on the MPSoC platform and, depending on the needs of the user, it is possible to reconfigure the FPGA tiles to accelerate one or the other application (Figure 6.16). Tasks are seamlessly migrated from the FPGA tile (i.e. the



(a) (b) (c)

Figure 6.15: *Flow (a) for linking the softcores to their host, which results (b) in a configuration hierarchy. Setting up a task on a softcore hierarchy (c).*

hardware implementation) to the ARM processor (i.e. the software implementation). During the migration process, one has to ensure that inter-task communication is kept consistent.

6.4.2 *Gecko*^{2}

The *Gecko*^{2} demonstration contains two distinct parts. The first part illustrates the dynamic creation and use of a configuration hierarchy. The use of a configuration hierarchy is also demonstrated in combination with task migration. The second part illustrates how the run-time manager can manage the communication of a NoC enabled MPSoC platform. Inter-task communication interference is handled in three different ways: by using the injection rate control mechanism described in Chapter 5, by rerouting communication and, finally, by migrating interfering tasks to another tile. Note that in this demonstration, though the decision mechanism is a simple script, the injection rate control actuator is used to perform congestion-control. In Chapter 5 the same actuating mechanism is used to perform flow-control under the supervision of a decision-making algorithm in the run-time manager.

Handling a Configuration Hierarchy

Using the edge detection application, we illustrate the use of hierarchical configuration[4]. First, we assign all edge detection tasks to execute in software on the StrongARM processor. An animated sequence of images (320x240 pixels, 16-bit colors) is edge-detected and displayed on the screen of the iPaq PDA.

[4] In our experience, the speed of hierarchical reconfiguration is constrained by the FPGA reconfiguration (proportional to the area to reconfigure). We expect to use hierarchical reconfiguration upon user interaction (application start-up or upon a change in quality requirements)

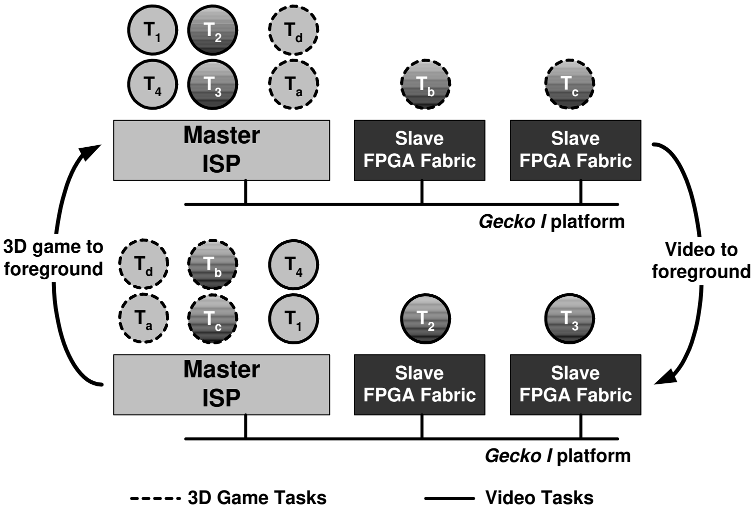


Figure 6.16: *Gecko I demonstration. This illustrates the envisioned application scenario. Depending on the user preference, the 3D game or the video application is put in the foreground. This affects the assignment of the HW/SW tasks.*

The software implementation of edge detection is very compute intensive and does not meet real-time constraints. Upon user request the run-time manager can migrate the edge-detect task to a hardware accelerator on the FPGA. As table 6.1 shows, the edge-detect application has registered four binaries targeting different architectures: a StrongARM binary, the FPGA bitstream for a dedicated reconfigurable hardware module, a binary for the Lezard16 and coefficients for a generic hardware convolution filter. Whereas the dedicated reconfigurable hardware task only requires the run-time manager to partially reconfigure the FPGA with its bitstream, the Lezard16 and convolution filter implementations illustrate hierarchical configuration. Indeed, if the (virtual) machines that execute these binaries are not currently instantiated on the platform, the run-time manager first reconfigures an FPGA module with the hardware to run the virtual machine and then configures it with the registered binary. Finally, it redirects messages to decode to the newly created task.

Managing Inter-task Communication Interference

After assigning the MJPEG video application tasks to tiles 1,3 and 8, we have purposely mapped a *message generator* task and a *message sink* task as a perturbing application on tiles 7 and 6 respectively (Figure 6.17). This way, the perturbing application will congest the communication channel (7 → 6) it shares with the video decoding application (Figure 6.17). Measurements have been performed for both bandwidth allocation techniques detailed in Chapter 5: window-spreading and block-allocation windows (Figure 6.18).

The effect of diminishing window size is clear on the *message sink* task in the case of the continuous-window allocation: the amount of messages sent (Figure 6.18(a)-

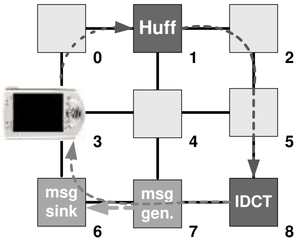


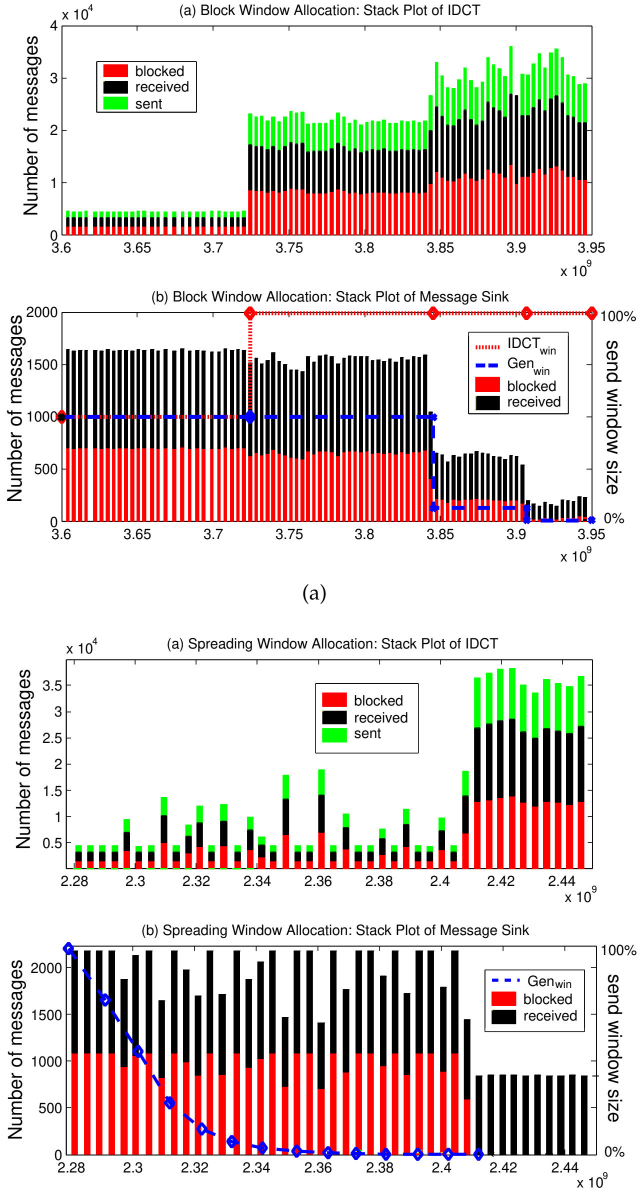
Figure 6.17: *Assignment of the video application and the perturbing application. Tiles 1,3 and 8 run the video decoder application and tiles 6 and 7 run a synthetic application to create congestion on the link* $7 \rightarrow 6$.

bottom). Optimal Video Decoder performance is obtained when less than 1% of the total bandwidth is allocated to the message generator (Figure 6.18(a)-top, time interval [3.91e9;3.95e9]). The run-time manager can trade-off performance between both applications by changing their respective injection rates.

When using the window-spreading technique, the effect of diminishing the total window size is not directly proportional to the bandwidth allocated and the trade-offs obtained in the previous case are not possible (Figure 6.18(b)-bottom). However, using window-spreading has other advantages: jitter is greatly reduced because communications are evenly spread over time. Moreover, a proper window setting can hide the latency of the receiver side and completely suppress message blocking. In Figure 6.18(b)-bottom at the time-stamp $2.41e^9$, the *message sink* task no longer causes message blocking in the NoC. This happens when the window of the message generator is less than 0.02% of the total bandwidth. Note that the message sink, is not disturbed by this window reduction: it still consumes 40000 messages per second. The run-time manager has simply matched the window size to the optimal sending rate in the perturbing application. As a consequence, thanks to the bandwidth saved by the run-time manager, the video decoder reaches its optimal frame-rate.

Besides exploiting the injection rate control mechanism, the run-time manager can also solve interference issues between applications in other ways. First of all, as Section 6.1.1 explains, it is possible to avoid the congested link by rerouting the video application stream around the hotspot, providing a form of operating-system controlled adaptive routing on the NoC. This technique allows maintaining the in-order packet delivery guarantee.

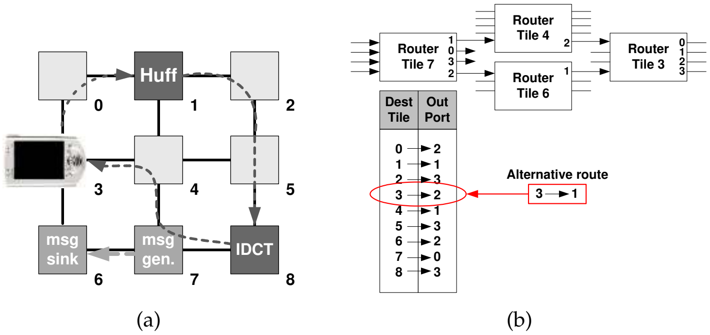
This concept is illustrated by Figure 6.19. Rerouting communication involves a few steps. First, the communication link between tile 8 and tile 3 (Figure 6.17) needs to be flushed in order to avoid out-of-order delivery of messages. The flushing is achieved by interaction between the run-time manager and the control NI. Upon request from the run-time manager, the initiator NI tags the last message it sends. Upon reception of a tagged message, the target NI notifies the run-time manager thereby acknowledging the virtual path between imitator and target network interfaces is empty. Once the link is flushed, it is safe for the run-time manager to alter the routing table of tile 7 (Figure 6.19(b)). This change ensures that messages for tile 3 are no longer



(a)

(b)

Figure 6.18: *Messages sent by Tile 8 (top) and messages received by Tile 6 (bottom). Time axis indicates the clock timestamp. The send windows of Tiles 8 and 7 are non-overlapping; their sum equals the send period. (a) Communication management experiment 1: Traffic shaping with continuous window allocation reduces blocking on messages received at Tile 6 (perturbing application) but also accordingly decreases the throughput. (b) Communication management experiment 2: Traffic shaping with window spreading better matches the perturbing traffic source. When blocking is removed, the throughput of the perturbing application is not diminished.*



Dest Tile	Out Port
0	2
1	1
2	3
3	2
4	1
5	3
6	2
7	0
8	3

(a)

(b)

Figure 6.19: *(a) Rerouting the video application stream between tile 8 and tile 3. (b) This is achieved by changing an entry in the routing table of the tile 7 router.*

routed through tile 6, but through tile 4 (Figure 6.19(a)). This route avoids the congested link. Based on our platform configuration from [135] (NoC clock at 22MHz and StrongARM access to control bus at 50MHz), we estimate the reconfiguration of a routing table entry to take about 450 NoC cycles and a link flush to take between 750 and 2800 NoC cycles depending on the amount of data to flush (and neglecting congestion).

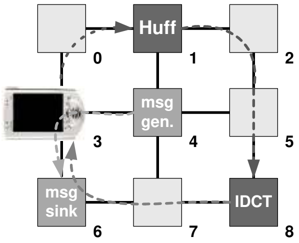


Figure 6.20: *Using task migration to resolve inter-application interference.*

The third possibility to avoid creation of a hotspot is for the run-time manager to dynamically migrate the message generator task to another node in the NoC. It is moved from tile 7 (in Figure 6.17) to tile 4 (in Figure 6.20). Messages between the generator task and the sink task are now routed through tile 3. This technique avoids the traffic generator to interfere with the video decoding application.

6.4.3 RESUME Run-Time Manager

The aim of the RESUME project[5] was to illustrate scalability in the context of multimedia terminals. This includes both content scalability (i.e. adapting the video stream according to the capabilities of the receiver) and multimedia processing scalability [68–70].

[5] IWT project funded by the Flemish Government (GBOU-RESUME project IWT-020174-RESUME).

Consequently, a heterogeneous multiprocessor platform containing an ISP and multiple FPGA fabric PEs was created[6]. Our contribution to the project was to provide a suitable operating system. This means an operating system that provides scalability support. As Section 6.3.1 details, the *Gecko*^{2} run-time manager was embedded into an existing RTOS (i.e. RTAI). For the RESUME project, we created a set of extensions containing the same run-time management components, further denoted as *Heterogeneous System Extensions* (HSE), for the Linux 2.4 kernel. This illustrates that the run-time management components and their algorithms can also be integrated into general purpose operating systems.

In contrast to an RTOS, Linux maintains a boundary between the kernel (i.e. kernel-space) and the applications (i.e. user space). Moving the HSE applications to *user-space* is the most important difference with respect to the *Gecko*^{2} demonstrators. This means that HSE tasks no longer pose a (potential) threat to the stability of the entire system and that more standard software libraries are available. Hence, the *Gecko*^{2} run-time manager functionality is split into a user-space run-time library (RTLib) and a set of kernel-space HSE run-time management components. The HSE run-time library is in fact the interface between the application developer and the HSE kernel. This happens much in the same way that the C library sits in between the application programmer and the Linux kernel. The application programmer, for example, rarely uses a kernel system call, instead a library function call is used that acts as a wrapper for the system call.

The purpose of the HSE kernel-space components is to allow the HSE core a way to control the hardware components and enable communication with the tasks assigned to different PEs. Obviously, these HSE drivers need to provide HSE with a fixed interface. The inverse communication (i.e. from hardware driver to HSE core) also happens through a fixed interface. The most important functions that need to be addressed in a PE driver are: creating and removing a task from the PE, starting a task, retrieving or restoring a certain task state (i.e. for task migration), suspending or resuming a task and providing communication functionality (i.e. sending messages, receiving messages, specifying communication parameters, blocking communication). Finally, HSE PE drivers can be registered or deregistered on the fly.

6.5 Conclusion

The Gecko series of demonstrators are emulators of NoC-enabled MPSoC platforms that execute real-life applications. They feature a complete MPSoC stack that spans from the real-time operating system layer down to the network link layer. The application programming model is message passing, so the system layer is very thin as it simply provides direct access to the message passing API of the network interface layer. In addition to being a convincing proof-of-concept of MPSoC architectures controlled by advanced real-time operating-systems, they are platforms that allow exploration of the design-space of (multi-processor) reconfigurable computing, of network-on-chip and of MPSoC run-time management.

[6] This platform was created by inserting multiple FPGA boards in the PCI slot of a host PC.

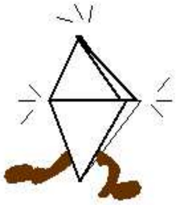
To the best of our knowledge, the *Gecko* platform series were the first to demonstrate NoCs emulated in a realistic real-time MPSoC environment. Quite ahead of their time, they were also the first world demonstration of (realistic) partial reconfiguration of Xilinx FPGAs, and later on introduced the (exotic) concept of hierarchical reconfiguration. Finally, they demonstrate the concept of reactive communication control of an NoC, both in terms of flow-control (Chapter 5) and congestion-control. They open the way to exploring close interaction between a run-time manager and the underlying MPSoC platform architecture.

Many lessons have been drawn from the Gecko experience. Some exotic concepts, such as HW/SW migration, dynamic partial reconfiguration and hierarchical reconfiguration versions thereof, are still ahead of their time. We expect partial reconfiguration to remain exotic[7] or to remain confined to some niche markets, such as network-processing. Though it is not impossible that FPGA fabric may be embedded into MPSoC ASICs on a larger scale, opening a possibility for hierarchical reconfiguration. Other concepts such as NoCs have since made their way into more mainstream MPSoCs (though they are still on the bleeding edge of technology).

Other lessons concern the message-passing programming model that directly uses the API offered by the network interfaces. Though usable (and natural to hardware designers and engineers), software designers often prefer the shared-memory paradigm, so the system-layer is required to extend the network stack to provide bus-like load/stores (or DMA-like) support. It is interesting to note that the commercial NoCs (like e.g. Arteris and Sonics) that have appeared since are marketed as *bus-replacements*, confirming this conclusion.

The lesson learned concerning emulators, is that they are convincing, fun to use and are great for software exploration at the system-level, but they also cost blood, sweat and tears to design and they are more difficult to exploit for research purposes, where high-level simulators are to be preferred. We have also found that it is sometimes uneasy to be too ahead of time and that it is important to have one particular component to sell. Finally, we also concluded that though daring engineering is a step to technical success, recognized achievements require the engineers to furthermore qualify in the art of marketing. At the same time we have found that being naive and over-optimistic about the complexity of some engineering projects can sometimes be a way of solving problems.

[7] Until a killer application is discovered that forces FPGA vendors to provide more hardware and design-tool support for this concept than they are doing today.



CHAPTER 7

Run-Time Quality Management

In order to address the time-to-market issue, an application developer should create applications that are easily deployable over multiple MPSoC platform generations (e.g. with increasing resource capabilities). The designer could provide the run-time manager with only a single application implementation and expect the run-time manager to deal with contention of critical platform resources or with changing user requirements through e.g. run-time task migration. However, the designer could choose to provide a scalable version of the application. This means an application with multiple operating points that each represent a trade-off between provided user quality and the required platform resources. Such applications can adapt to varying user quality requirements.

Indeed, assisted by application design tools, the designer could provide a scalable application implementation where each application operating point represents a trade-off between quality and required resources. There would even be multiple operating points when only considering a single application quality. Each such operating point would use a different amount of platform resources to provide the same end-user quality. In general, these applications are denoted as adaptive applications.

Such scalable *adaptive* applications allow the run-time manager, for example, to select the right operating point for every active application in order to optimize the total user value, while considering the platform capabilities. In essence, run-time management support for adaptive applications enables both platform scalability and application scalability. The run-time management component that deals with the application quality levels, their associated user values, and their resource needs is the *quality manager*.

The quality manager essentially relies on the application quality information, the available platform resources and the user requirements to make its decisions. This means the quality manager requires an efficient and fast algorithm to select the best operating point for every active application given the platform resource constraints. Furthermore, if an application needs to be deployable on multiple generations of an MPSoC platform, this operating point information should be provided in a way that is not platform specific. This means that the quality manager will have to reconcile a platform independent description of resource needs with a platform specific description of available resources. Finally, as the quality manager relies on the resource manager to assign the platform resources, both run-time management components have to closely cooperate.

The rest of the chapter is organized as follows. Section 7.1 provides an overview of quality management for adaptive applications. This includes a description of the quality management concept and the quality manager components, an adaptive application case study with a motivation for having multiple implementations for a single application quality level and, finally, a description of the quality management issues. Section 7.2 details an efficient and fast algorithm to solving the quality manager operating point selection problem. Section 7.3 proposes a quality manager interface with both the application design-time analysis information and the resource manager. This includes proposing a novel way for the quality manager to make a high-level assessment of the mapping feasibility of an application operating point. Section 7.4 details the interaction between the quality manager and the resource manager in the presence of multiple starting and stopping applications. In addition, it proposes several new quality manager algorithm optimization alternatives. Section 7.5 presents the related work with respect to quality management and its algorithms. Finally, Section 7.6 presents the conclusions.

7.1 Adaptive Quality Management Overview

As Section 2.2 explains, our system manager accepts application quality and implementation information that is generated at design-time. This means the system manager is aware about the capabilities and quality levels supported by the (adaptive) application and their respective properties.

This section details the quality management concept (Section 7.1.1), provides an adaptive application example (Section 7.1.2), motivates having multiple implementations for a single application quality level (Section 7.1.3) and, finally, provides an overview of the issues tackled in the rest of this chapter (Section 7.1.4).

7.1.1 Concept

As Section 2.3.3 already explains, adaptive applications can be defined as applications that support multiple modes of operation along one or more resource and/or quality dimensions [30,48,51,52,105,112,185]. At design-time, both the application quality required by the user and the available platform resources could be unknown (e.g. other applications could be running). By providing the run-time manager with the quality and resource trade-offs for every application, it can select the right quality and resource usage in order to (1) maximize the user application value and (2) optimize the usage of the platform resources.

Figure 7.1(a) illustrates our quality management concept.[1] The system manager contains a quality manager and a resource manager. The quality manager contains a QoE manager and an operating point selection manager. Every application i comes with a set of quality options q_{ij} with their respective required platform resources $\vec{r_{ij}}$ and implementation details d_{ij}. The QoE manager is responsible for assigning a user value v_{ij} to every application quality q_{ij} according to a *session utility function* $f_u(\cdots)$. As Figure 7.1(b) shows, the highest quality (i.e. VGA, 30 frames/s) should not always provide the highest user value. Consequently, the operating point selection manager is responsible for selecting an operating point $(v_{ij}, \vec{r_{ij}}, d_{ij})$ for every application i in order to maximize the value of a *system utility function* while ensuring that the sum of every resource type k of the selected operating point j remains within the boundary of what is available on the platform R^k, i.e. $\sum_{i,k} r_{ij}^k \leq R^k$. Finally, for every application i, the resource manager has to perform the actual platform resource allocation based on the selected operating point.

The QoE manager is responsible for assigning, at run-time, a value to every application quality level that reflects its user appreciation versus its cost. Figure 7.1(b) shows an adaptive video application example featuring four different application implementations providing three quality levels with a different resolution and framerate. In Figure 7.1(b) the QoE manager assigns a user value (high, medium, low) to the different application quality levels. In this case, the user prefers a lower framerate or even a lower resolution rather than top quality, because (1) high quality video is not needed (e.g. for video-conferencing) and (2) the cost (e.g. energy cost) of high quality video is too high. As Jingwen et al. [105] describes, multimedia users need a simple way to communicate with the QoE manager, i.e. to control and customize the quality of their multimedia applications. A common way is to provide a graphical user interface with a limited number of quality trade-off options. Two features need to be present in such a user interface. First, users need a way to score a quality perception (e.g. excellent, good, fair and bad) and to select other user related specifications such as e.g. window size. Secondly, users should be able to specify the price or price range they want to pay for the desired service. Without notion of cost, users have no reason to choose anything besides the highest quality. An in-depth look at the QoE manager is out of the scope of this thesis. For more information, we refer to related work like e.g. Jingwen et al. [105].

Consequently, the operating point selection manager uses the user values and the resource vectors to select, for every active application, an operating point in order to maximize the system utility function, while making sure that the total amount of required platform resources (i.e. amount of PE, Mem and BW) does not exceed the available platform resources. In essence, the operating point selection manager has to solve a *Knapsack Problem*, more specifically a *Multiple-choice Multi-dimensional Knapsack Problem*.

[1] Here we assume type 1 run-time adaptivity according to the classification specified in Section 2.3.3

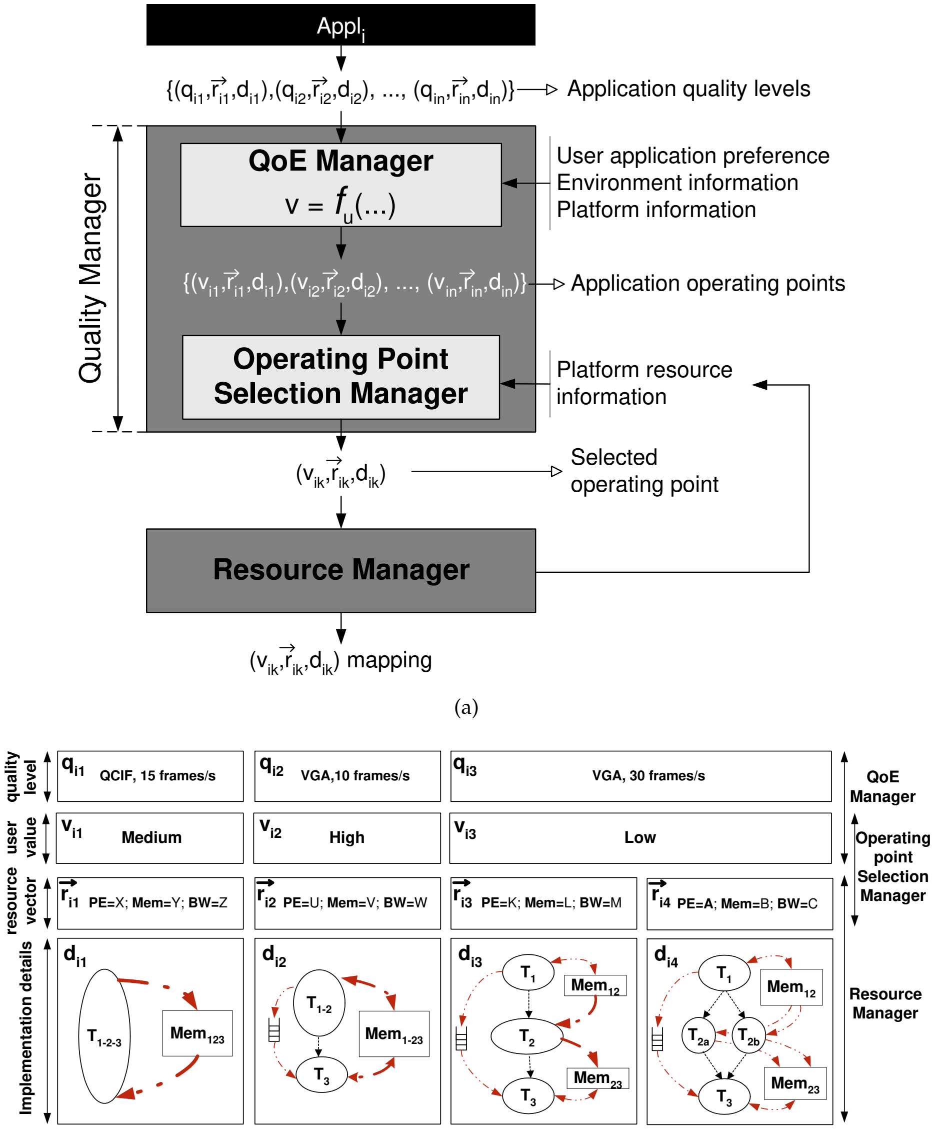

Appl$_i$
$\{(q_{i1},\vec{r}_{i1},d_{i1}),(q_{i2},\vec{r}_{i2},d_{i2}), ..., (q_{in},\vec{r}_{in},d_{in})\}$
Application quality levels
Quality Manager
QoE Manager
$v = f_u(...)$
User application preference
Environment information
Platform information
$\{(v_{i1},\vec{r}_{i1},d_{i1}),(v_{i2},\vec{r}_{i2},d_{i2}), ..., (v_{in},\vec{r}_{in},d_{in})\}$
Application operating points
Operating Point Selection Manager
Platform resource information
$(v_{ik},\vec{r}_{ik},d_{ik})$
Selected operating point
Resource Manager
$(v_{ik},\vec{r}_{ik},d_{ik})$ mapping
(a)
quality level
user value
resource vector
Implementation details
q_{i1} QCIF, 15 frames/s
q_{i2} VGA,10 frames/s
q_{i3} VGA, 30 frames/s
v_{i1} Medium
v_{i2} High
v_{i3} Low
$\vec{r}_{i1}$ PE=X; Mem=Y; BW=Z
$\vec{r}_{i2}$ PE=U; Mem=V; BW=W
$\vec{r}_{i3}$ PE=K; Mem=L; BW=M
$\vec{r}_{i4}$ PE=A; Mem=B; BW=C
d_{i1}
$T_{1\text{-}2\text{-}3}$
Mem_{123}
d_{i2}
$T_{1\text{-}2}$
T_3
$Mem_{1\text{-}23}$
d_{i3}
T_1
T_2
T_3
Mem_{12}
Mem_{23}
d_{i4}
T_1
T_{2a}
T_{2b}
T_3
Mem_{12}
Mem_{23}
QoE Manager
Operating point Selection Manager
Resource Manager
(b)

Figure 7.1: Our quality management concept. (a) The quality manager consists of a QoE manager, that captures the user preference, and an operating point selection manager, that selects the best application operating point considering the current state of the platform. (b) Example video decoding application. The QoE manager transforms the application qualities into application user values. The operating point selection manager selects the best operating point given the user value and the resource vectors, the resource manager assigns resources based on its implementation details.

The classic 0-1 *Knapsack Problem* (KP) can be explained as a puzzle [120]: a hitchhiker needs to fill his knapsack with various objects. Each object has a weight and provides a particular value. The knapsack itself can be filled up to a maximum weight. How should the hitchhiker fill his knapsack in order to maximize value while staying within the maximum weight boundary? In more general terms: how to maximize value when picking a number of value items for a resource-constrained knapsack? The *Multiple-Choice Knapsack Problem* (MCKP) deals with *groups of items*. This adds additional constraints as one has to choose one item from every group. In the *Multi-Dimensional Knapsack Problem* (MDKP), the items have multiple value dimensions, while the knapsack contains multiple constrained resource dimensions. The *Multiple-choice Multi-dimensional Knapsack Problem* (MMKP) combines the above [9], i.e. groups of multi-dimensional items, where one has to choose a single item for every group to be added to the multi-dimensional knapsack. Again, the goal is to optimize value, while remaining within the knapsack bounds. We consider applications with multiple, multi-dimensional operating points. The problem of choosing a single multi-dimensional operating point for every active application in order to optimize the usage of the multi-resource MPSoC platform (i.e. a multi-dimensional knapsack) is, hence, a MMKP.

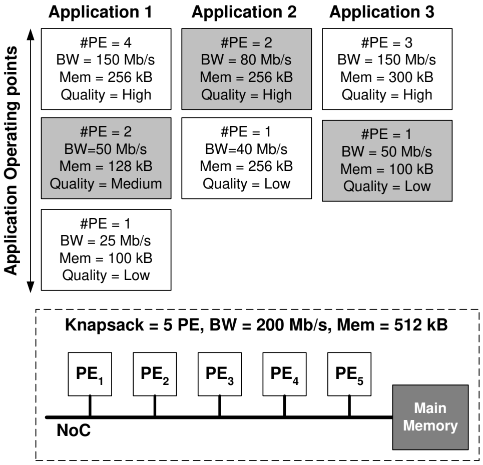


Figure 7.2: *The MMKP problem in the MPSoC context: selecting a application operating point for every active application in order to reach an optimized value. However, the sum of all resources used by the applications should not exceed the available MPSoC resources.*

Example 7.1: Simple MPSoC MMKP illustration (Figure 7.2).
In this simplified example, the MPSoC platform is a three-dimension knapsack of processing elements (5 PEs), an on-chip communication resource (BW=200 Mb/s) and a main memory (Mem=512 kB). For a certain amount of each of these three resources, an application provides several quality levels. The goal is to choose an operating point for every application such that the overall provided quality is maximized while the sum of used resources never exceeds the available resources. The selected points fill the knapsack almost completely as they require 5 PEs, 180Mb/s BW and 484 Kb Mem.

The resource manager is responsible for communicating platform information to the quality manager. The quality manager requires two types of information. First, the resource manager has to provide the operating point selection manager with detailed platform resource usage information. This allows the operating point selection manager to verify if the selected operating points are respecting the platform resource boundaries. Secondly, the resource manager has to indicate a priority for searching and selecting feasible solutions. This means, for example, that the resource manager could indicate that using a scarce or fragmented resource should be avoided. In the end, the resource manager uses the resource vector and the respective implementation details (i.e. task graph and other designer specified details) of the selected operating point to perform the actual resource assignment. This also means that the resource manager can resort to task and/or data migration to make the assignment a success.

Example 7.2: Interaction of resource assignment policy and operating point selection (Figure 7.3).
In this example, the designer provides, for the same application, two implementations with the same quality and value to the user. The first implementation uses a lot of memory but only requires two processors, while the second uses four processors but only requires a small amount of memory. Both implementation options can produce a valid resource assignment for both platform 1 and platform 2. However, the first implementation would be preferred for platform 2 (where processor tiles are a scarce resource), while the second implementation would be preferred for platform 1 (where memory is a scarce resource). Avoiding a scarce resource implies that the resource manager expects that other applications might be started in the future. The resource manager could employ a different policy that attempts to shut down as many processors as possible in order to save energy. This would mean that the first implementation would also be preferred in the situation of platform 1.

The combination of a QoE manager, an operating point selection manager and a resource manager corresponds to the three quality layers for adaptive applications detailed by Jingwen et al. [105]. These levels include a *user quality layer* that details what the user perceives (i.e. resolution and framerate) and the price or value that the user attributes to the different qualities, an *application quality layer*, provided by the designer, that is platform independent but that does detail the application specific quality attributes and, finally, a *resource quality layer* that contains platform specific resourcing details and allows a run-time manager to perform the resource assignment

7.1.2 Application Adaptivity: QSDPCM Case Study

In this section, we illustrate application implementation trade-offs using the QSDPCM real-life video encoding application. As Marescaux describes [132,133], the *Quad-Tree Structured Differential Pulse Code Modulation* (QSDPCM) algorithm is an inter-frame compression technique for video images [214]. It consists of a hierarchical motion

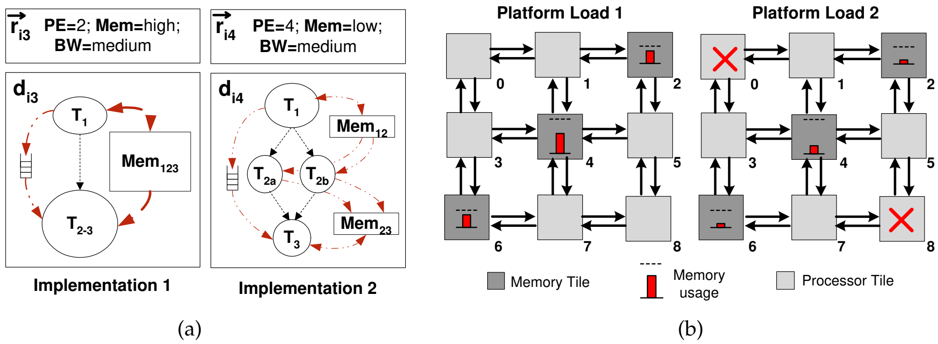


Figure 7.3: *Influence of resource assignment policy on selecting the operating point. (a) Two operating points provide equal value to the user but have different implementation trade-offs. (b) Two platforms with little memory and a lot of processing elements and vice versa. Depending on the platform resource usage and the resource assignment policy, one of the implementations is preferred.*

estimation, quad-tree quantization, a Huffman encoder and image reconstruction (Figure 7.4(a)).

The hierarchical motion estimation consists of 3 steps: first at quarter resolution (ME4), then half resolution (ME2), and finally at full resolution (ME1) for every macro-block (16x16 pixels) in the image. It determines the motion vector with which the difference between the current and reconstructed previous image can be encoded using the shortest code word. For this purpose the input image is sub-sampled (SS) by a factor four and two. Quad-tree quantization (QC) works by recursively splitting the motion compensated predicted error frame from the ME1 step into four quadrants until, within a certain threshold, such a quadrant can be approximated by its quantized mean value. The image reconstruction decodes the frame by dequantization and motion compensation (MC). The output stream is compressed using a Huffman encoder. Brockmeyer et al. [32] provides an in-depth analysis of the QSDPCM kernel optimization steps and the multiprocessor mapping options and trade-offs.

Figure 7.4(b) details the processing requirements of every kernel for encoding a QCIF resolution image. In order to get a more balanced set of *mapping kernels*, the basic QSDPCM kernels are combined. This means that the subsampling (SS) and the ME4 and ME2 kernels are merged into a single ME42 mapping kernel, while the QC and MC kernel are combined to form the single QC mapping kernel. These mapping kernels communicate through a shared memory and they are synchronized at the slice[2] level. Consequently, one can use these mapping kernels to create application tasks (Figure 7.5(a)). First, all mapping kernels are combined to create a single application task. Then a functional split into two and three applications tasks is considered. As ME1 requires far more cycles than ME42 and QC, a set of ME1 data-splits are considered. In this case, each ME1(x) kernel processes parts of the image.

The task graphs of Figure 7.5(a) are mapped onto a multiprocessor virtual platform (provided by [132]) containing seven TIC62 processing element tiles and an L2 mem-

[2] a set of macro-blocks that span the width of the image

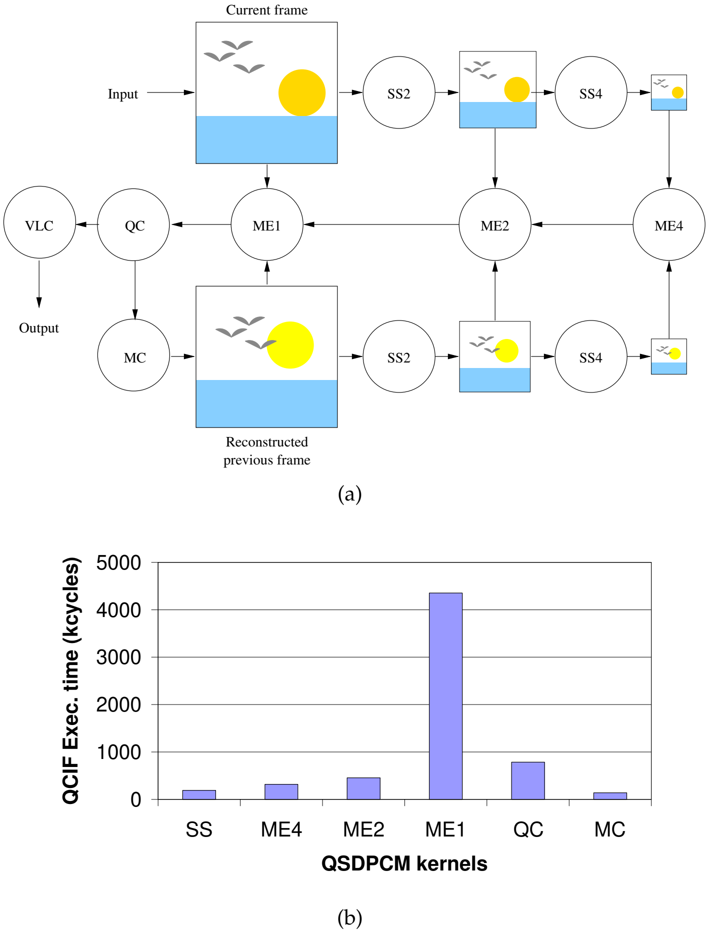
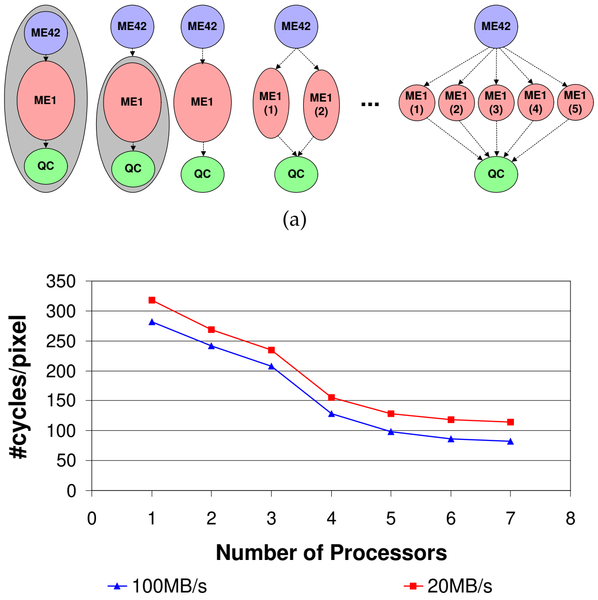


Figure 7.4: *QSDPCM application. (a) Description of the algorithm. (b) Execution time (kcycles) of the different kernels for QCIF resolution.*

ory tile interconnected by a NoC. We measured the overall application execution time as a function of the parallelization (one task per processor) and the assigned L2 memory bandwidth (Figure 7.5(b)). A similar graph can be constructed for resolutions other than QCIF. This experiment shows that for a certain perceived user quality, one can select different implementation trade-offs.

7.1.3 Exploiting Multiple Application Implementations

There are multiple ways to implement a single application (Figure 7.3(a)). A designer can make various implementation decisions, like e.g. the number of parallel tasks and their synchronization granularity, that affect the resource usage and their trade-offs. This section motivates the use of multiple application implementations and reflects on its benefits and drawbacks.



(a)

(b)

Figure 7.5: QSDPCM mapping results. (a) QSDPCM implementation options ranging from a single task to seven parallel tasks. (b) Performance of the QSDPCM application as a function of the parallelization and the used L2 bandwidth.

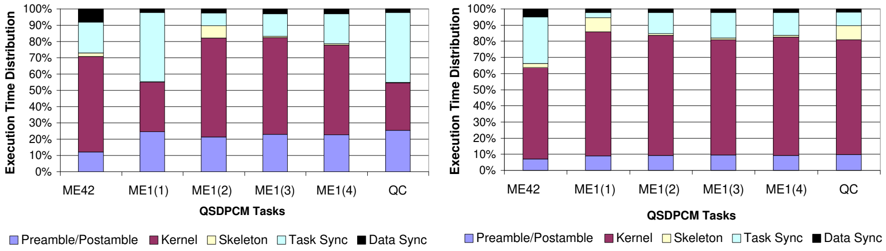
Motivation

An application designer typically provides a single application implementation with associated application quality properties. Indeed, it is often a single task graph with different resource requirements depending on the quality needs. Such a designer expects the run-time (resource) manager to flexibly allocate the required resources according to the desired quality. In case of the QSDPCM example (Figure 7.5(a)), the designer would most likely provide the seven-task implementation as this is the most flexible and balanced implementation. The task graph will exhibit different processing, memory and communication requirements depending on the requested resolution and framerate.

In contrast, just like for the QSDPCM example, a designer could create multiple implementations with different task graphs featuring e.g. different parallelizations (Figure 7.5(a)). In this case, the run-time manager could select the best implementation given the quality requirements, the state of the platform and the current resource allocation policy (e.g. shut down as many PEs as possible). However, there are other benefits to having multiple implementations. In case of low user quality requirements, it is beneficial to have a single task instead of multiple communicating tasks. Besides the fact that a single task has no synchronization overhead, it also avoids scheduler overhead and the scheduling side-effects like e.g. cache trashing.

Brockmeyer et al. [32] has analyzed the QSDPCM implementation using six tasks for both QCIF (Figure 7.6(a)) and VGA (Figure 7.6(b)) resolution. These figures show

how much time is relatively spent in every task. In both cases, a significant amount of time is spent in executing the kernels, while the time required for executing the control code surrounding the kernels, further denoted as the task *skeleton*, is quite small. But, for VGA resolution, the relative amount of kernel execution time is a lot higher. As a consequence, for QCIF resolution, a significant larger share of time being *wasted* in task synchronization and in the task preamble/postamble. This task preamble/postamble represents the time needed to respectively initiate and terminate a task run for a set of data. Furthermore, we notice that the data-split for VGA resolution is a lot more balanced than for QCIF resolution. This is caused by the fact that a QCIF frame has 11 columns that need to be (unevenly) split over four tasks. In contrast, a VGA frame that has 40 columns that split nicely over four tasks. The fact that, for VGA resolution, a task has larger chunks to process also explains the smaller preamble, postamble and task synchronization



(a) (b)

Figure 7.6: *QSDPCM mapping results for QCIF and VGA resolution using the six-task implementation. (a) QCIF resolution (b) VGA resolution.*

The overall conclusion of this study [32] is that this six-task implementation becomes more efficient with increasing frame resolution. This also means that, although this implementation is perfectly capable of handling QCIF resolution, it would be more appropriate to use another implementation with fewer tasks in order to reduce the amount of wasted computing resources.

Feasibility?

Is it feasible to expect multiple implementations for a single application quality? Indeed, creating multiple implementations obviously comes at a cost both at design-time and at run-time.

The design-time cost is mainly caused by having the designer implement multiple versions. However, by using an evolution of today's application design and exploration tools such as SPRINT [44], OpenMP [56] and the ATOMIUM tool suite (see Appendix C), a future designer should be able to create multiple implementations starting from a single sequential application code with minimal effort.

Having multiple implementations also requires managing them at run-time. In order to avoid duplication of application code for every task graph, we have shown that it is possible to integrate different operating points into a single code-base with minimal overhead [51,159]. For the QSDPCM example, the code size overhead of the integrated version with respect to the sequential stand-alone version is less than 5%.

The additional configuration data needed to distinguish between different modes of operation requires an additional 404 bytes. The per task performance penalty (QCIF resolution) due to the introduction of additional control code is less than 0.17%. These overhead figures allow us to be confident that having multiple implementations is feasible. Furthermore, in the future, we would expect the application design tools to be able to provide integrated output code.

7.1.4 Quality Management Issues

Although the quality management concept is clear, there are a few issues that remain to be solved when adding a quality manager to the system run-time manager. First, the quality manager requires a flexible and fast way for solving the Multiple-choice Multi-dimensional Knapsack Problem (MMKP) at run-time. Secondly, this means determining the interaction between quality manager and resource manager.

These issues are addressed in the rest of this chapter. First, Section 7.2 describes a generic way to solve the MMKP in a fast and efficient way. Then, Section 7.3 analyzes the quality manager interface with both the application and the resource manager. Finally, Section 7.4 investigates the interaction between the quality manager and the resource manager when a user starts and stops multiple applications. This also involves switching between operating points.

7.2 Solving the MMKP

Having multiple active applications, each with their proper set of operating points, requires a run-time manager to select for every application the most appropriate operating point given the available platform resources and the user constraints. This section details this operating point selection problem and proposes a fast heuristic algorithm to solve it. In addition, the algorithm is characterized by a set of experiments.

7.2.1 Problem Definition

Consider that every application i of a set of active applications S, contains a set s_i of N_i operating points (for the algorithm complexity analysis N_i is constant for every i, i.e. is equal to N).

The platform architecture contains a set of m resource types $\vec{R} = (R_0, \cdots, R_k, \cdots, R_m)$, where R_k denotes the available resources of type k.

Every operating point j of set s_i, further denoted as pt_{ij}, is characterized by a cost c_{ij} and an execution time t_{ij}. Furthermore, r_{ijk} denotes the amount of k-type platform resources needed for operating point pt_{ij}. Hence, $\vec{r_{ij}} = (r_{ij0}, \cdots, r_{ijk}, \cdots, r_{ijm})$ denotes the resource usage vector for pt_{ij}.

To translate this MMKP into a mathematical formulation, let x_{ij} denote whether the point j of set s_i is selected ($x_{ij} = 1$) or not ($x_{ij} = 0$). Equation 7.1 denotes that, for each active set, exactly one point must be selected. The total resource usage of all active applications cannot exceed the available platform resources (Equation 7.2).

$$\forall i, \sum_j x_{ij} = 1 \tag{7.1}$$

$$\forall k, \sum_i \sum_j x_{ij} r_{ijk} \leq R_k \tag{7.2}$$

The goal of the operating point selection is to minimize the total cost, i.e. minimizing $\sum_i \sum_j x_{ij} c_{ij}$. This constrained minimization problem can be transformed into a different form [139]. First, consider each set as an ordered set with respect to the cost axis, i.e.: $j \leq j' \implies c_{ij} \geq c_{ij'}$. Then, one can substitute c_{ij} by the *value* v_{ij} as follows: $v_{ij} = (c_{i0} - c_{ij}),\ v_{ij} \geq 0$. Hence, the minimization problem becomes the new maximization problem detailed by Equation 7.3.

$$Maximize(\sum_i \sum_j x_{ij} v_{ij}) \tag{7.3}$$

This maximization problem is a classical MMKP. The MMKP goal is to create a solution $f \in F$ by picking exactly one operating point from each set in order to maximize the total value of the pick, subject to the resource constraints. Hence, a *solution* $f \in F$ of the MMKP is a combination of points, one per set, that satisfies all resource constraints. All additional symbols are summarized in Table 7.1.

7.2.2 Operating Point Selection Heuristic

Low algorithm complexity is crucial for a good and fast operating point selection algorithm. A reasonably good and feasible solution with little computational run-time effort is preferred over an exact solution with a longer computation time. Hence, a fast and effective heuristic is needed.

Solving the MMKP with our heuristic [49, 50] involves an off-line and two on-line steps. The first, (off-line) step filters out all non-Pareto operating points of every set. This step that can be performed independently from the selected MMKP heuristic. It allows us to reduce the number of points in each initial MMKP benchmark set used to measure the heuristic performance, It also reduces the execution time of the heuristic without sacrificing solution quality [79]. This Pareto filtering step is performed by an implementation of the *Simple Cull* (SC) algorithm [241]. When integrating this operating point selection algorithm into our run-time manager, the Pareto filtering step will also play an important role. Indeed, only after the QoE manager has assigned values to every quality level, one can derive the Pareto operating points.

Algorithm 11 details the two on-line steps. The second step (lines 1-6) performs multi-dimensional resource reduction. This reduction starts with finding an initial solution that includes the lowest-value point[3] pt^0_{ij} from each set s_i. For the multi-

[3] Starting with the highest-value points can dramatically exceed some available resources and does not perform well to find a feasible solution. Selecting the lowest value points to represent the penalty vector holds two assumptions. First, it assumes that the lowest value point also consumes the least resources. Secondly, it assumes that the resource usage of the lowest value points is indicative for the higher value points of the same set.

Symbol		**Definition**
Sets	S	Set of active applications
	s_i	Operating point set of application i, $s_i \in S$
	$N_i = N$	Number of operating points in s_i
Points	pt_{ij}	Point j of set s_i
	pt^0_{ij}	Lowest-value point in the set s_i
	v_{ij}	Value of the point pt_{ij}
	c_{ij}	Cost of the point pt_{ij}
	x_{ij}	1 or 0, if pt_{ij} is respectively selected or not
	r_{ijk}	Amount of resource type k used by point pt_{ij}
	r^0_{ijk}	Amount of resource type k used by point pt^0_{ij}
Resources	m	Number of resource types
	$\vec{R}$	Resource vector for the m resource types
	R_k	Available resources of type k, component of $\vec{R}$
	R^0_k	Type k resources used by the initial MMKP solution
	$\vec{p}$	Resource penalty vector
	p_k	Penalty vector component for resource type k
	R_{ij}	Single resource for the point pt_{ij}
Solution	F	Set of all feasible solutions
	f	Solution of the MMKP $f = \{pt_{1i}, ..., pt_{Sj}\} \in F$
	$f_{current}$	Current feasible solution $f_{current} \in F$
	f_{new}	New (potentially feasible) solution
	f^{value}	Total (system) value for a solution

Table 7.1: *Additional symbols used for solving the MMKP.*

choice knapsack problem, an initial feasible solution, if one exists, can always be obtained by choosing the lowest-cost point of each set. This no longer holds for the MMKP. So, in the worst case, every possible operating point combination must be tested in order to find a feasible solution, which amounts to solving the MMKP itself. Hence, using a heuristic does not guarantee finding a feasible solution. The amount of resources used by this initial solution is given by Equation 7.4

$$\forall k, R^0_k = \sum_i r^0_{ijk} \tag{7.4}$$

$$\forall k, p_k = R^0_k / R_k \tag{7.5}$$

$$R_{ij} = \vec{p} \cdot \vec{r_{ij}} = \sum_k p_k \times r_{ijk} \tag{7.6}$$

Then, as in [10], the multi-dimensional resource vector $\vec{r_{ij}}$ of every point pt_{ij} is reduced into a single resource representing the price of the resource combination.

To that end, a *penalty vector* $\vec{p} = (p_0, \cdots, p_k, \cdots, p_m)$ is defined (Equation 7.5) to give a high penalty to any highly used resource. For any resource type k, the more resources are used by the initial solution, the larger the penalty component p_k.

The single resource R_{ij} of any point pt_{ij} is then derived as defined by Equation 7.6. It is the projection of the resource usage vector $\vec{r_{ij}}$ of pt_{ij} with the penalty vector

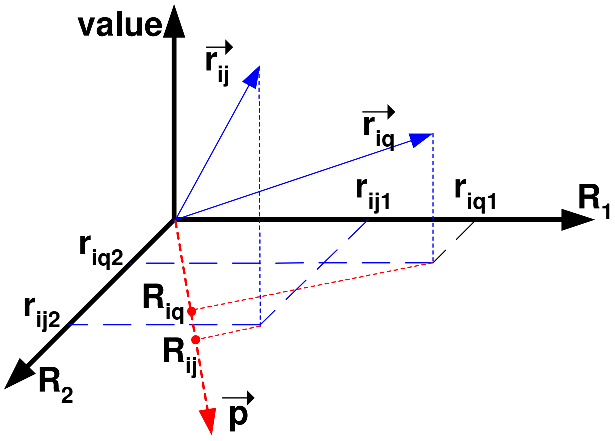


Figure 7.7: *The single resource is obtained by projecting the resource vectors $\vec{r_{ij}}$ onto the penalty vector $\vec{p}$.*

$\vec{p}$. This is illustrated by Figure 7.7. It can be obtained by making the sum of all resources used by the point weighted by their respective penalty vector component. The complexity of this step is $O(m + sN)$.

Where Akbar et al. [10] use a reduction to construct a convex hull of each set separately, we use this reduction to be able to sort all points of all sets together in a single two-dimension search space.

Consequently, all points of all sets in the previously derived two-dimension search space are sorted according to the value v_{ij} over single-resource R_{ij} ratio. Sorting is done in descending order according to this angular coefficient (i.e. v_{ij}/R_{ij}). This means that points with a high value combined with a small resource usage are preferred. The worst-case complexity of this step is $O(sNlog(sN))$.

Algorithm 11: *MMKP solver.*

```
Input: $\vec{R}, S$
Output: $f \in F$
MMKPSolve($\vec{R}, S$)
$f_{current} = FindInitialSolution(S)$
$f_{current}^{value} = \sum v_{ij}, \forall pt_{ij}\ in f_{current}$
$\vec{p} = DeterminePenaltyVector(f_{current})$
foreach $pt_{ij} \in S$
    $R_{ij} = DeriveSingleResource(pt_{ij}, \vec{p})$
foreach Unused $pt_{ij}$ with highest $v_{ij}/R_{ij}$
    $f_{new} = Exchange(f_{current}, pt_{ij})$
    if $((Feasible(f_{new})$ and $(v_{ij} > v_{ik}))$
        ExchangePoint($f_{current}, pt_{ij}$)
        $f_{current} = f_{new}$
```

The third and final step solves the MMKP, i.e. selects a point for every set while maximizing the system utility function. In this case, this means maximizing the total value. Starting from the initial solution, the algorithm visits every point in descending v_{ij}/R_{ij}. If replacing point pt_{ik} with pt_{ij} of set i produces a feasible solution $f_{new} \in F$ and results in an increased overall value (i.e. $f_{new}^{value} > f_{current}^{value}$), the new solution is adopted.

The time complexity of the third step is $O(sN)$. This implies that the overall worst-case complexity of the heuristic is only $O(m + 2sN + sNlog(sN))$, in contrast to the ones of [10,113].

After visiting all sorted points, one could consider updating the penalty vector and perform another algorithm iteration in order to improve the found solution. This step was omitted because experiments showed that the solution improvement is marginal for the additional execution time.

7.2.3 Experimental Setup

The experiments provide results for all benchmark sets (denoted I01, I02, ..., I13) provided by [1]. These benchmarks are representative of adaptive multimedia systems and are generally used to benchmark a MMKP heuristic. As Table 7.2 details, they are of different sizes, with $s \leq 400$, $N_i \leq 10$, and $m \leq 10$.

Table 7.2: *MMKP Benchmark parameters*

Benchmark	s	N	m
I01	5	5	5
I02	10	5	5
I03	15	10	10
I04	20	10	10
I05	25	10	10
I06	30	10	10
I07	100	10	10
I08	150	10	10
I09	200	10	10
I10	250	10	10
I11	300	10	10
I12	350	10	10
I13	400	10	10

In order to benchmark our solution, we have implemented the heuristics detailed in [10,113] on the cycle-accurate SimIt-ARM simulator [2] for the StrongARM architecture running at 206 MHz.

We have similarly optimized the C code of these heuristics for StrongARM. Each heuristic has been restricted to a single iteration to reduce the execution time as much as possible, without relevant penalty on the solution quality.

7.2.4 Experimental Results

Pareto Filtering

The off-line (i.e. design-time) filtering of points removes all non-Pareto points of the multidimensional space. With respect to the used benchmarks sets, up to 26% of all points are eliminated. This step increases the heuristic speed performance up to 25.8%, while the total value reduction is limited to maximally 3.7%.[4] However, it also reduces the resource usage up to 4.4%. A similar trend is observed when other algorithms [10, 113] are used.

MMKP Heuristic

The sorting of points in descending order according to this angular coefficient is illustrated by Figure 7.8a for $s = 5, N = 5, m = 5$, i.e. benchmark I01 [1]. Consequently, Figure 7.8b shows both the initial and final solution.

The execution times for the considered benchmarks and heuristics are reported by Figure 7.9(b). The results reflect the very low complexity of our heuristic (denoted *IMEC*). The IMEC heuristic shows more than 97.5% gain for the execution time on a StrongARM processor, compared to two other candidate heuristics [10, 113]. Our heuristic is also the only one to run in less than 1ms on an embedded processor for problems with $s \leq 30$, $N_i \leq 10$, $m \leq 10$, which represent realistic MPSoC MMKP problem sizes.

Figure 7.9(a) shows that the total value obtained with our heuristic is comparable with respect to the other candidate heuristics. The dramatically high execution time of [10] for small problems is due to the trigonometric function usage (e.g. arc tangent) which is expensive especially on embedded processors[5].

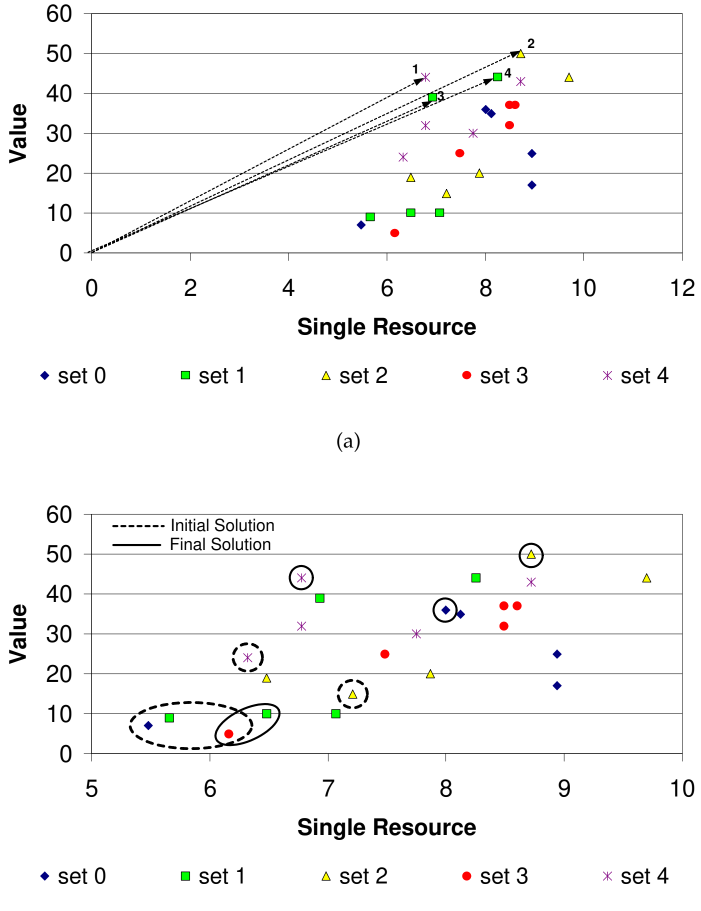
As in [10, 113], one can iterate the algorithm by updating the penalty vector after each iteration and sorting Pareto points accordingly in order to further improve the solution. But, as already mentioned this is not considered for performance reasons.

7.3 Quality Management Interfaces

Section 7.2 solves the MMKP problem for MPSoC sized problems in an isolated and abstract way: the available platform resources and the application operating point resource needs are expressed in a dimensionless way and the actual resource assignment is neglected. This section investigates the interfacing (i.e. interaction) of the quality manager with both the application (i.e. the operating point description) and the resource manager. To this end, it incorporates the MMKP algorithm into the quality manager and it assesses the impact of this interaction. This includes evaluating the performance in case of platform independent description of application resource needs.

[4] At first glance, it seems odd that removing non-Pareto points results in a value reduction. However, one has to bear in mind that (1) the removed operating points belong to the multi-dimensional resource space and not to the single-resource space and that (2), in the end, the reduced operating points get sorted according to their value over single-resource ratio. This means the heuristic could visit some non-Pareto points earlier than some Pareto ones.

[5] Within SimIt-ARM, the arc tangent function is executed within 230μs, whereas a integer division (resp. multiplication) is executed within only 40 (respectively 10) μs.



(a)

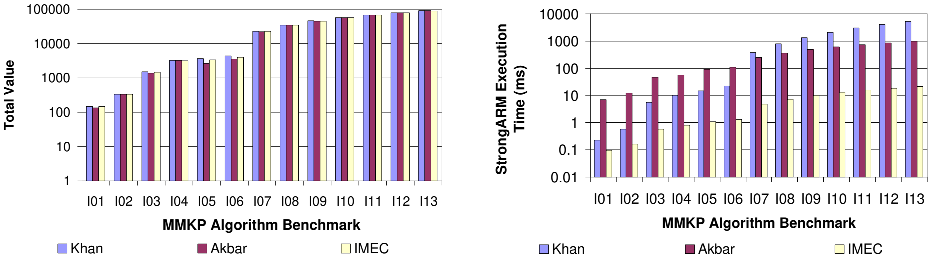
(b)

Figure 7.8: MMKP I01 benchmark results. (a) Sorting the single-resource points by their angular coefficient. (b) Initial and final solution.

7.3.1 Flexibility and Scalability Issues

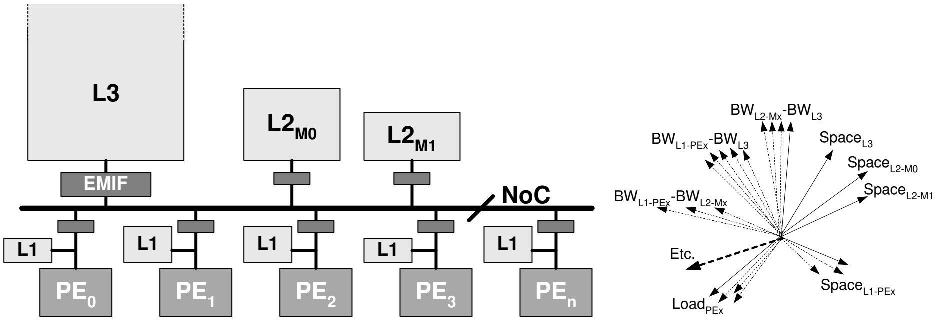
Section 7.2.2 provides a fast and efficient solution for the MMKP problem for MPSoC size problems. However, just applying this approach to a set of adaptive applications that need to be mapped onto an MPSoC platform reveals some scalability and flexibility issues.

Consider the MPSoC platform and its associated resource axis illustrated by Figure 7.10. An operating point described in the multidimensional space of Figure 7.10(b) exactly details how much of every platform resource is used. This way, the operating point selection manager (Algorithm 11) can easily check if a selected point is feasible or not with respect to the already used platform resources. By selecting an operating point, the quality manager also selects the mapping of the task graph onto the archi-



(a) (b)

Figure 7.9: *MMKP benchmarking results: (a) total value of the solution and (b) algorithm execution time on StrongARM.*



(a) (b)

Figure 7.10: *Description of (a) a platform architecture and (b) its associated operating point space. The architecture features multiple processing elements, each with their private level-1 (L1) memory, interconnected with the NoC to the shared level-2 (L2) and level-3 (L3) memory. Every application operating point in that space describes how much of each resource is needed to provide its associated application quality.*

tecture graph. Intuitively, it is easy to understand that if the resource usage for every resource is described by the operating point, the resource assignment is, de-facto, done. So the role of the resource manager is, in this case, reduced to executing the platform dependent resource allocation mechanisms (i.e. executing the mapping).

As a consequence, the quality manager requires a large number of operating points per implementation of every quality level in order to remain flexible, as other applications might already be executing on the platform. Besides the fact that such operating point space representation is in itself not scalable, the scalability problem also appears when moving to a different MPSoC platform instantiation. Even if this other platform contains the same resource types, one requires a different operating point space with, again, a large number of operating points per implementation of every quality level. This approach also requires that all target platforms are known at design-time or that the operating point space can be transformed on the fly to a new target platform. In case all solutions must be provided at design-time, the total number of operating points N_k for every application k is (in general) given by Equation 7.7, where Q_k denotes the number of application quality levels, I_q denotes the number of implementations per quality level, M_i denotes the number of mappings per implementation and P denotes the number of supported platform instances.

$$N_k = Q_k \times I_q \times M_i \times P \tag{7.7}$$

The number of design-time operating points can be reduced primarily by avoiding the M_i and P factor in Equation 7.7. This means avoiding additional operating points for every mapping (flexibility challenge) and for every target platform (scalability challenge). Such a solution requires three components: (1) a description of the design-time operating points in a way which is not tied to a specific platform architecture instance, (2) a quality manager that still makes valid decisions based on this new operating point description and (3) a matching run-time resource manager that can assign the selected operating point in a flexible way to the available resources on a specific platform instance.

The following sections describe the interface between our quality manager and our resource manager (Section 7.3.2) and compare the issues and performance of platform instance independent operating point descriptions (Section 7.3.3).

7.3.2 Interface between Quality Manager and Resource Manager

This section details the interface between the quality manager and the resource manager. For simplicity, the quality manager is limited to just the operating point selection manager. This means that the transformation from operating point quality to operating point user value is already done. Figure 7.11 details the run-time management components and their interface. The quality manager receives as input a set of application operating points for application i, each characterized by their required resources $\vec{r_i}$ and their value v. The quality manager receives two types of input from the resource manager. First, it receives a penalty vector to perform the multi-dimensional resource reduction and sort the operating points. Secondly, it receives a description of the available platform resources to verify the validity of the selected operating point with respect to the described resources. After selecting an operating point, its task graph TG is forwarded to the resource manager for assignment. In case the assignment fails, the selected point is discarded and the quality manager should select a new operating point based on the, potentially updated, penalty vector and available platform resources.

The actual purpose of the penalty vector is to allow the quality manager to sort the operating points based on their platform resource cost. In contrast to the penalty vector described in Equation 7.5 (Section 7.2), this penalty vector has to reflect the current platform resource cost. This means that a scarce resource should have a high penalty. This can be achieved either by Equation 7.8 or by Equation 7.9. Intuitively, Equation 7.8 makes sense as every penalty vector component k represents the percentage of used resource. This scheme works if resources are normalized with respect to their importance and if they are converted into dimensionless values (as was the case for the benchmarks used in Section 7.2). However, in case of an MPSoC

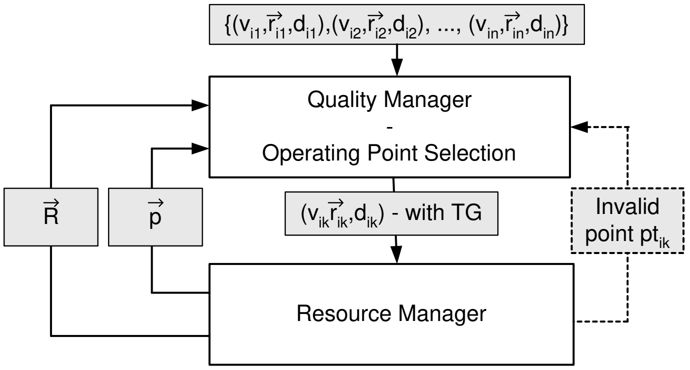

$\{(v_{i1},\vec{r}_{i1},d_{i1}),(v_{i2},\vec{r}_{i2},d_{i2}), ..., (v_{in},\vec{r}_{in},d_{in})\}$
Quality Manager
-
Operating Point Selection
$\vec{R}$
$\vec{p}$
$(v_{ik}\vec{r}_{ik},d_{ik})$ - with TG
Invalid point pt_{ik}
Resource Manager

Figure 7.11: Interface between the operating point selection manager (i.e. quality manager) and the resource manager. The operating point selection manager selects an operating point. The penalty vector $\vec{p}$ allows the quality manager to sort the operating points, while the vector of available platform resources $\vec{R}$ allows the quality manager to check the feasibility of the selected point. The implementation details, including the task graph (TG), of the selected point is forwarded to the resource manager so platform resources can be assigned. In case the assignment fails, the selected point is invalidated.

platform, there are e.g. many more processor tiles than memory tiles. This inherent scarcity is not considered by this penalty vector. In addition, every penalty vector component of a platform without applications would be zero. Hence, Equation 7.9 represents a more suitable penalty vector. Indeed, this penalty vector normalizes the required resources with respect to the available resources.

$$\forall k, p_k = \frac{R_k^{used}}{R_k^{total}} = 1 - \frac{R_k^{available}}{R_k^{total}} \tag{7.8}$$

$$\forall k, p_k = \frac{1}{R_k^{available}} \tag{7.9}$$

The resource manager maps the task graph $TG(T, C, D)$ to the architecture graph $AG(P, L, M)$ according to the heuristic described in Algorithm 12. This is an extended version of the resource assignment algorithm (Algorithm 3) discussed in Chapter 3 (page 56). The extension allows the resource manager to also handle memory blocks $d_u \in D$ (task graph *data*) and platform memory tiles $m_i \in M$. This means that all task graph memory blocks are sorted based on their size (largest first) and the total number of memory accesses (largest first). Consequently, we sort the memory tiles based on their tile size (smallest first), their usage and on their already present communication load (highest usage first). Finally, we assign the most important task graph data to the memory tile with the smallest cost. If needed, the memory assignment component performs backtracking just like Algorithm 3 implements backtracking for assigning tasks to tiles. Once all memory blocks have been assigned, the algorithm performs the resource assignment of tasks and communication links according to Algorithm 3, with the memory assignment as a boundary condition. This effectively means that memory blocks $d_u \in D$ are considered as pre-assigned tasks and platform memory tiles are considered as a special type of processor tiles. This means the task graph $TG(T, C, D)$ is transformed to $TG'(T', C)$ and the architecture graph $AG(P, L, M)$ is transformed to $AG'(P', L)$.

Algorithm 12: *Extended Generic Heuristic for resource assignment.*

```
Input: $TG(T,C,D)$, $AG(P,L,M)$
Output: Assignment $TG(T,C,D) \rightarrow AG(P,L,M)$
ExtendedGenericHeuristic($AG(P,L,M), TG(T,C,D), bt$)
(1)  $PrioritizeDataBlocks(TG(T,C,D), M)$
(2)  foreach unmapped $d_u$ with highest $Prio(d_u)$
(3)      $N(t_i) = PrioritizeMemoryTiles(d_u, TG(T,C,D), AG(P,L,M))$
(4)      if $N(d_u) > 0$
(5)          Assign $d_u$ to $m_i$ with lowest $Cost(m_i)$
(6)      else
(7)          $PerformBactracking(bt)$
(8)  if (All memory blocks assigned)
(9)      $TG'(T',C) = Transform(TG(T,C,D), D \rightarrow M)$
(10)     $AG'(P',L) = Transform(AG(P,L,M), D \rightarrow M)$
(11)     $GenericHeuristic(TG(T,C), AG(P,L), bt)$
(12) else
(13)     No solution found. Exit.
```

7.3.3 Design-Time/Run-Time Interface

This section addresses the issue of defining a scalable operating point space (design-time) that does not already include the platform mapping, and that is not dependent on a specific MPSoC platform instance. This operating point description should still allow the quality manager to select an operating point that (1) represents a good quality given the cost of the platform resources *and* (2) that can be assigned by a resource manager to the actual available platform resources.

The most straightforward way to create such an operating point space is to describe every operating point by its total amount of required processing element resources, its total amount of required memory resources and the amount of required communication bandwidth to/from those memory resources. This holds the assumption that (1) the shared memory tiles bandwidth is the most critical communication resource, while processor-to-processor communication is not a critical resource and that (2) issues with this resource can easily be resolved by the resource manager. Indeed, as there are many more processing element tiles than memory tiles and, in case of a NoC with flexible communication paths, the resource manager could find a different path.

When using this approach, one has to check the performance of two quality management functions. First, can the multidimensional resource reduction be done based on a penalty vector given by the cost of platform resources? Secondly, can the quality manager verify, on a high level, the assignability of the selected point? This means that, although the quality manager does not perform an actual mapping, it should still be able to check that the selected operating point can be mapped by the resource manager onto the platform. This ability should gain a considerable amount of time.

The expected issue when using the sum of resources is taking *fragmentation* into account. Indeed, even if the sum of required resources is smaller than the available platform resources, the mapping could be impossible due to resource fragmentation. This fragmentation is (1) due to resources being split over multiple tiles and links and (2) due to previously assigned applications. Furthermore, using the sum of resources makes it hard in case of multiple implementations for a single quality. These different implementations will require a similar amount of resources, but with a different *distribution* or *fragmentation*. For a single quality level, the more distributed implementations will require more resources (due to additional overhead), resulting in a higher value over single-resource ratio. Hence, the operating points with more fragmented implementations will only be selected by the quality manager after the assignment of the operating points with less fragmented implementation has failed. This results in a series of quality manager/resource manager iterations. Although the resource manager can steer the operating point selection by penalizing the most critical resource, it currently cannot steer towards the right amount of fragmentation.

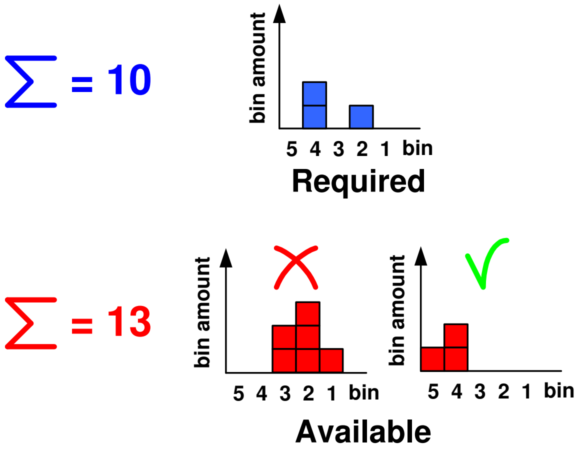


Figure 7.12: *Impact of using a resource histogram for determining operating point feasibility. A certain operating point can seem feasible when only considering the sum of the available and needed resources. Using a resource histogram can reveal that the selected point is, nevertheless, infeasible.*

Example 7.3: Fragmentation and resource assignment (Figure 7.12).
In this example, the application (on top) only requires 10 resource units while 13 platform resource units are available (bottom). So the quality manager might conclude that this application operating point is, obviously, feasible. However, the feasibility actually depends on the fragmentation of the available resources. In this case, the application operating point requires at least two memory resources of size 4 to be feasible. Which means that the more fragmented platform resource situation (bottom, left) is not capable of hosting the operating point.

As it neglects resource fragmentation, relying on the sum of resources obviously creates verification problems for the quality manager. In order to overcome this problem without resorting to analyzing the task graph associated with the operating point, we introduce, for every resource type, a resource histogram for every operating point and for the platform. For every operating point, such a histogram is included as additional design-time information. At run-time, the resource manager can provide the quality manager with the platform resource availability histogram.

Algorithm 13: *Checking feasibility by means of* available *and* needed *resource histograms*

```
Input: $NeededHisto$, $AvailHisto$
Output: $Feasibility$
CHECKHISTOFEASIBILITY($NeededHisto$, $AvailHisto$)
(1)  foreach ($b_{needed} \in [NrBins, 1]$ in decending order)
(2)    $Needed = NeededHisto[b_{needed}]$
(3)    if $(Mode = Restrictive)$
(4)      $b_{avail}^{start} = b_{needed} + 1$
(5)    else
(6)      $b_{avail}^{start} = b_{needed}$
(7)    $b_{avail} = b_{avail}^{start}$
(8)    while $(Needed \neq 0)$ and $(b_{avail} \leq NrBins)$
(9)      if $AvailHisto[b_{avail}] \neq 0$
(10)       $Needed = Needed - 1$
(11)       Reduce $AvailHisto[b_{avail}]$ by 1
(12)       Increase $AvailHisto[b_{avail} - b_{needed}]$ by 1
(13)       $b_{avail} = b_{avail}^{start}$
(14)     else
(15)       $b_{avail} = b_{avail} + 1$
(16)   if $(Needed \neq 0)$
(17)     $Feasibility = 0\ (FALSE)$
(18)   else
(19)     $Feasibility = 1\ (TRUE)$
```

Algorithm 13 details how the quality manager checks the feasibility of an operating point by verifying the histogram of needed ($NeededHisto$) and available ($AvailHisto$) resources. The core of this algorithm resembles a bin packing algorithm using a first fit decreasing strategy [45]. Indeed, the algorithm starts with the highest needed resource bin and attempts to find a suitable element in the first fitting available resource bin. Whenever such an available bin b_{avail} is found, the size of both the needed bin $Needed$ and available bin b_{avail} is decreased by one, while the available bin that represents the size difference (i.e. the leftover resource) between the available bin and the needed bin (i.e. $b_{avail} - b_{needed}$) is increased by one.

There are two ways of matching the required resource histogram with the available resource histogram. The first way, denoted as *optimistic*, compares a required resource histogram bin with an available resource histogram bin of the same size (line 6). However, due to the bin-width, it is still possible that the mapping fails. Indeed, consider a bin b_i with a certain minimum value b_i^{min} and a maximum value b_i^{max}. It will be impossible to make the assignment if the available resource value is equal to b_i^{min} and the required resource value is equal to b_i^{max}. To overcome this problem, we have to compare in a more *conservative* way, i.e. comparing the resource needs histogram bin b_i with the next histogram bin b_{i+1} of the available resources (line 4). Unfortunately, this means some valid solutions might not be considered as feasible by the quality manager.

Example 7.4: Checking feasibility using a resource histogram (Figure 7.13).
This example covers the histogram feasibility checking algorithm for both the *conservative* and the *optimistic* approach. In both cases, the example starts out with the same available and needed resource histograms. The difference between conservative and optimistic is already clear in step 1: while the optimistic approach matches an element of the needed size 4 bin with an element of the available size 4 bin, the conservative approach requires the element of the available histogram to be in a larger bin. In the optimistic case, the matching reduces both the available and needed size 4 bin (step 2), while in the conservative case, the matching results in an additional element for the available size 1 bin. The algorithm continues until it can no longer match the needed histogram elements (i.e. failure in case of the conservative approach in step 5) or until all needed histogram elements are successfully matched (success in case of the optimistic approach in step 6).

7.3.4 Experimental Setup

In order to assess the performance of the collaboration between the quality manager and the resource manager, we use the PSFF tool (Appendix A) to randomly generate 100 sets that each contain between 10 and 40 Pareto operating points (22 on average) distributed over a low, a medium and a high quality level with user values of about 1000, 3000 and 9000 respectively. The input parameters for the PSFF random kernel graph generator tool are given by Table A.2, while the parameters for the PSFF operating point generator are detailed in Table A.3. These parameters have to be considered in the context of the resources available on the evaluation platform detailed in Figure 7.14.

The characteristics of the generated operating points and their associated task graphs are detailed in Section A.4. The task graphs contain up to 7 tasks and up to 3 memory blocks.

Figure 7.14 details the platform layout for the quality manager interface experiments. It consists of a 16 tile MPSoC interconnected by a 4-by-4 mesh using XY routing. Every NoC link has a capacity of 400MB/s. It contains 13 PE tiles, 2 small memory tiles of size 1024kB and 1 large memory tile of size 2048 Kb. For the experiments, the platform is randomly loaded prior to starting a new set. This means that for every resource type a random load of 40%-80% for high load (H), 20%-60% for medium load (M), and 10%-30% for low load (L) is generated.

The goal of the experiment is to assess the performance of adding a new application with a set of operating points onto a loaded MPSoC platform. This involves the quality manager for selecting the operating point with the highest value that can still be assigned by the resource manager to the available platform resources. The

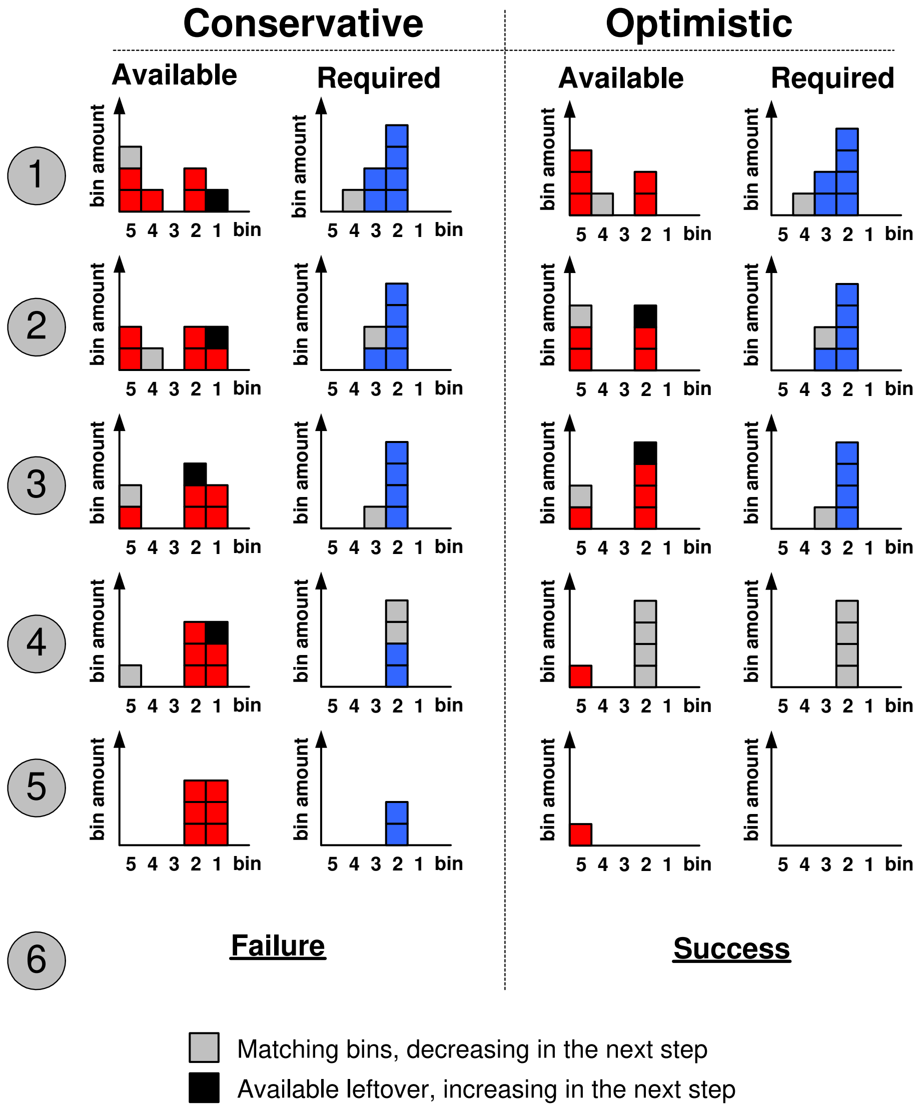


Figure 7.13: *Checking feasibility by means of a resource histogram. For the conservative approach (left), the algorithm uses an available element of a larger bin to accommodate the element of the required bin. For the optimistic approach (right), the algorithm starts matching the required and available elements of the same bin-size. After bin-matching with different bin sizes, an additional element appears in the difference bin that represents the leftover resource. For example, in step 1, an element in bin 1 appears that represents the leftover of matching required bin 4 with available bin 5. The algorithm stops after all required histogram elements have successfully been matched (right) or when the remaining available elements can no longer be matched (left).*

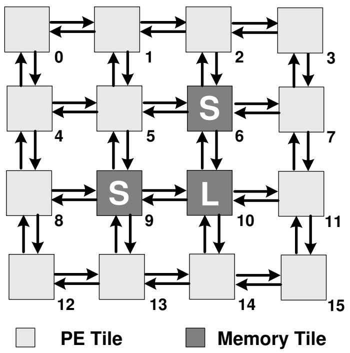


Figure 7.14: Platform for design-time vs. run-time interface experiments. This platform contains 13 PE tiles, 2 small memory tiles (S) and 1 large memory tile (L).

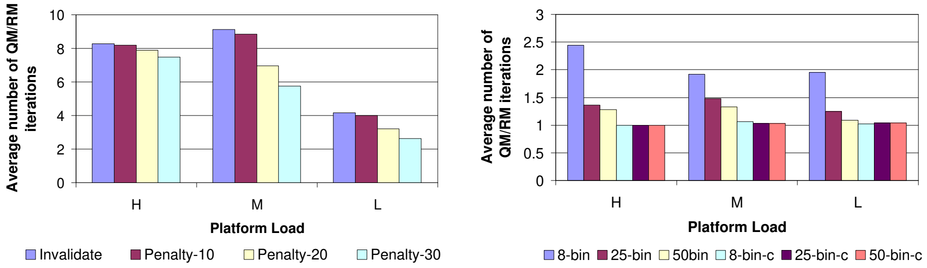
performance will be measured by (1) the amount of *quality manager/resource manager (QM/RM)* iterations i.e. how many times do we have to select an operating point and assign its associated task graph until we reach an assignment, (2) the assignment success rate of a selected operating point, (3) the user value of the selected and assigned operating point and, finally, (4) the operating point selection and resource assignment execution time.

7.3.5 Experimental Results

First we assess the impact of having a platform independent description of operating point resource needs and its impact on the collaboration between quality manager and resource manager. Figure 7.15 details the required amount of QM/RM iterations (Figure 7.11) for selecting the best operating point and assigning the associated task graph to an already loaded platform for 100 application sets.

We can clearly distinguish two performance categories. The first experiment (Figure 7.15(a)) only considers the sum of resource needs with respect to processing power, memory needs and memory bandwidth needs. In case the resource manager is only able to *invalidate* the previously selected point, it takes on average about eight to nine iterations to reach an assignment for high (H) and medium (M) platform load and more than four iterations for low (L) platform load. In case the resource manager can provide feedback by means of the penalty vector, the situation improves. Consequently, we consider the situation when an extra penalty for critical resources (as seen by the resource manager) of respectively 10%, 20% and 30% is applied. This means that every time the resource assignment fails, the critical resource type, i.e. the resource type that causes the assignment failure, receives an extra penalty in the next iteration. Although this improves the situation, it still requires, on average, a little more than two QM/RM iterations.

The second experiment (Figure 7.15(b)) involves the usage of a resource histogram. We investigate the usage of an 8-bin histogram (which fits nicely into a 32-bit word), a 25-bin histogram and a 50-bin histogram in both an optimistic and a conservative way. As expected, introducing a resource histogram significantly increases the ability of the quality manager to assess the mapping feasibility of a selected operating point, which translates into a large decrease of QM/RM iterations. While the optimistic approach (first three bars of Figure 7.15(b)) still requires between 2.4 and 1.1 iterations (depending on the platform load and the number of histogram bins), the conservative approach reduces the number of iterations for all histogram-based approached to less than 1.1 iterations. When investigating the reason why the quality manager occasionally still assesses the feasibility in a wrong way, we notice that this occurs in case a single large memory block with high bandwidth requirements is present in the task graph. The quality manager checks the feasibility of bandwidth and memory separately. However, in case of multiple memory tiles, it might be that one memory tile can satisfy the memory needs, while another satisfies the bandwidth needs. This then results in an assignment failure. Consequently, increasing the number of bins will not further decrease the number of iterations.



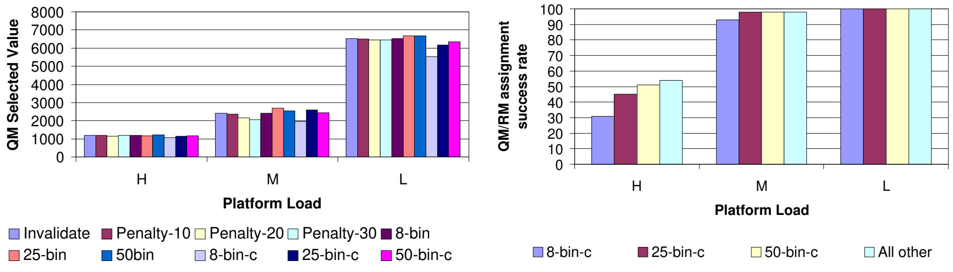
(a) (b)

Figure 7.15: *Number of quality manager/resource manager iterations. (a) The resource manager feedback loop either just invalidates a non-assignable operating point or uses the penalty vector to steer the next operating point selection. (b) Operating point feasibility checking by means of a resource histogram.*

Besides the number of QM/RM iterations, we also have to consider other parameters like the overall value attached to the operating points selected by the quality manager, the assignment success rate of the resource manager, and the execution time of both the quality manager and the resource manager. Indeed, the number of iterations could easily be reduced by having the quality manager select low value operating points that can be assigned without any problem by the resource manager or by stating that no solution can be found without additional platform resources.

With respect to value, Figure 7.16(a) details the average value of the selected operating point for different platform load options. In case of high platform load (H), a low quality operating point (value 1000) is selected. In case of medium platform load (M), the quality manager selects a mixture of low quality (value 1000) and medium quality (value 3000). Similarly, in case of low platform load (L), a mixture of medium quality (value 3000) and high quality (value 9000) is selected. In addition, a few performance observations can be made. First, we notice that the optimistic 25-bin and the 50-bin perform best in terms of selected value. Going to a restricted mode mostly reduces the average value in case of low platform load. This is caused by the fact that some high quality operating point solutions are seen as unfeasible because some of their resource requirements are located in the largest histogram bin.

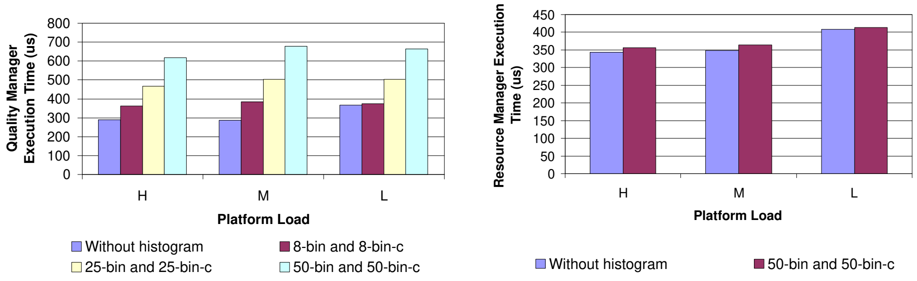
The assignment success rate is detailed in Figure 7.16(b). This shows how many times the collaboration between the quality manager and the resource manager ended up in a successful assignment. We see that all solutions, except the conservative histogram approaches, yield the same result: 100% success in case of low platform load (L), 98% success in case of medium platform load (M) and 54% success in case of high platform load (H). These figures have to be interpreted with respect to the assignment heuristic. As Section 3.7 of Chapter 3 explains (see also Figure 3.5), in some cases the assignment heuristic does not find a valid assignment even when the full search algorithm indicates that a valid assignment exists. Indeed, for high platform load, the resource assignment heuristic finds a solution in on average 84% of the cases with respect to a full search algorithm. The conservative histogram approaches result is fewer successful assignments, again, because the quality manager sometimes discards valid operating points during the verification process in an attempt to minimize the number of QM/RM iterations. However, we see that the conservative 25-bin and the conservative 50-bin approach is as successful as all the other approaches in case of low and medium platform load and is, respectively, 9% and (only) 3% less successful in case of high platform load.



(a) (b)

Figure 7.16: Selected operating point value and assignment success rate. (a) Selected operating point value. (b) Resource manager assignment success rate for the operating point provided by the quality manager.

Finally, we consider the execution time of both the quality manager (Figure 7.17(a)) and the resource manager (Figure 7.17(b)), measured on a SimIt StrongARM ISS with a clock speed of 206 MHz. With respect to the quality manager, we notice that the histogram approaches requires more processing power. When using 50-bins, the histogram approach requires twice as much time as the non-histogram approaches. However, when considering that the conservative 50-bin approach often requires just one execution iteration per operating point, while the non-histogram approaches require, on average, at least four QM/RM iterations. In this context, the conservative 50-bin approach is the best option. The resource manager is responsible for assigning the task graph associated to the selected operating point. Figure 7.17(b) shows that the resource manager execution time is a somewhat larger than the execution time reported in Section 3.7. This is due to a larger platform (4-by-4 tile platform instead of a more heterogeneous 3-by-3 tile platform). We notice that the resource manager execution time is quite independent of the selected quality manager approach. The minor difference is caused by the fact that, on average, the amount of task graph items that need to be assigned is smaller in case of the non-histogram approach. Indeed, in the non-histogram approaches, the operating points with the least fragmented task graphs are selected first as they exhibit a better value over single-resource ratio (i.e. same value, but more resources needed due to overhead).



(a) (b)

Figure 7.17: Execution time for selecting and assigning a single feasible operating point. (a) Quality manager execution time for selecting a feasible operating point. (b) Resource manager execution time for assigning the selected operating point.

Concluding Remarks

Adding resource histograms enables the platform independent quality manager to check the assignment feasibility of a selected operating point in a more reliable way. In turn, this seriously reduces the number of QM/RM iterations and, hence, the elapsed time for selecting and assigning an operating point. Figure 7.18 illustrates the difference between the explored options. Indeed, when determining the time t for selecting and assigning a single point, one has to calculate the quality manager operating point selection time $t^{QM}_{single\ selection}$ and the resource manager assignment time $t^{RM}_{single\ assignment}$ according to Equation 7.10. This equation shows that it is important to minimize the number of QM/RM iterations.

$$t^{total}_{single\ point} = \#\,\text{QM/RM iterations} \times (t^{QM}_{single\ selection} + t^{RM}_{single\ assignment}) \qquad (7.10)$$

One has to consider that there are several cases where this gain will become even more important. First, when the resource manager gets extended with e.g. NoC path finding [131] or task scheduling (see future work in Chapter 8). This will increase the execution time of the resource manager, which makes that the execution time penalty for passing a nonassignable operating point becomes larger. Secondly, being able to select a feasible operating point with a high degree of certainty becomes even more critical when deciding on multiple operating points in a single quality manager run. This occurs, for example, when a new application starts and, as a consequence, already executing applications have to switch. Nevertheless, a feedback-loop between both management entities will always be needed because the proposed operating

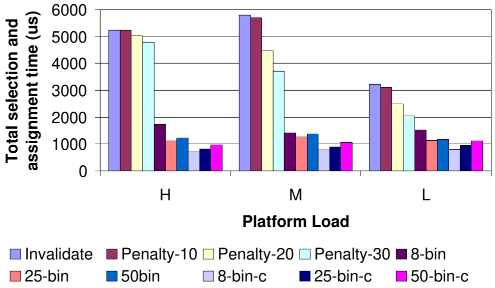


Figure 7.18: *Total time for selecting and assigning an operating point. It is important to minimize the number of QM/RM iterations. in order to minimize the total time.*

point selection feasibility checking mechanism is not infallible, and because the resource assignment heuristic does not always find an assignment solution. This can be caused when e.g. the feasibility checking algorithm verifies multiple operating points at once, while the resource assignment heuristic assigns one set at a time.

The conservative approach sometimes prevents an assignable high value operating point from being selected. One can avoid this by over-dimensioning the fragments of the critical platform resources[6]. In case of using 50 bins, this corresponds to over-dimensioning by 2%, while it amounts to 12.5% for the 8 bin approach (i.e. an extra bin). This should bring the selected value of the conservative approach up to the level of the optimistic approach without sacrificing on the number of QM/RM iterations (Figure 7.16(a)).

In the rest of this chapter, we use a conservative 50-bin approach (50-bin-c) as it provides the best trade-off with respect to the selected value, the assignment success rate and the algorithm execution time.

7.4 Quality Manager and Resource Manager Integration

While Section 7.3 investigates the interfaces of the quality manager by selecting and assigning an operating point of a single set (i.e. application), this section investigates the integration and collaboration of the quality manager (using the 50-bin conservative histogram approach) and the resource manager in a more realistic situation, i.e. a situation where one or more new applications are started or stopped over time.

The rest of this section is organized as follows. The integration of the quality manager and the resource manager and the associated issues are discussed in Section 7.4.1. Then, Section 7.4.2 details how the MMKP algorithm of Section 7.2 is integrated into a quality manager that uses resource histograms for feasibility checking. Consequently, Section 7.4.3 introduces a tuned operating point selection algorithm, aiming

[6] This means that adding an extra resource tile for the critical resource does not improve the operating point assignability. One has to over-dimension the existing tiles in order to make sure that they end up in a larger resource bin.

to provide the same solution quality at a lower cost. Finally, Section 7.4.4 details the experimental setup while Section 7.4.5 details the experimental results.

7.4.1 Integration and Collaboration Description

In case of an ever changing set of applications, each with their operating points and associated user values, there are some additional issues that have to be addressed. As Section 7.3 explains, both the quality manager feasibility checking and the resource manager task graph resource assignment are imperfect. This becomes more complicated when performing operating point selection and resource assignment on multiple points at once. Hence, one has to determine how the quality manager and the resource manager will collaborate and handle such situations. In doing so, we still have to (1) minimize the number of QM/RM iterations, (2) maximize the total application value and (3) minimize the total execution time. In addition, this means we have to consider operating point switching, i.e. selecting a new operating point for an already assigned application. This obviously means that resources will have to be reassigned after such a switch. As operating point switching is a costly operation, one should only use it when necessary.

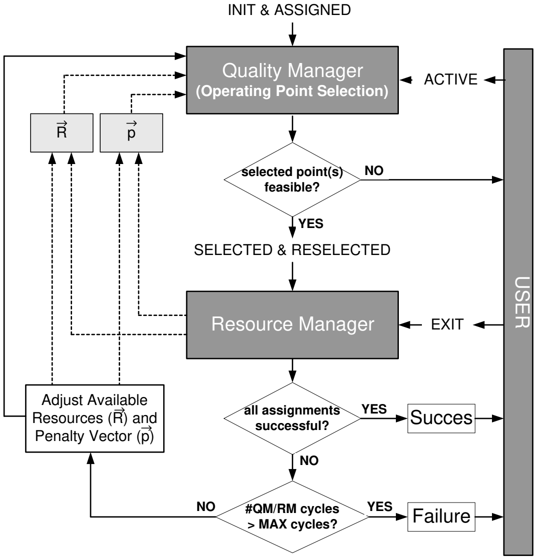

INIT & ASSIGNED
Quality Manager
(Operating Point Selection)
ACTIVE
$\vec{R}$
$\vec{p}$
selected point(s) feasible?
NO
YES
SELECTED & RESELECTED
Resource Manager
EXIT
USER
Adjust Available Resources ($\vec{R}$) and Penalty Vector ($\vec{p}$)
all assignments successful?
YES
Succes
NO
NO
#QM/RM cycles > MAX cycles?
YES
Failure

Figure 7.19: *High-Level collaboration between quality manager, performing operating point selection, and the resource manager.*

Figure 7.19 details how the quality manager and the resource manager collaborate. Every application or set maintains a variable that describes its current state. Three actors are involved in changing the state of any given set. First, the *user* is responsible for activating and terminating an application. Secondly, the quality manager is responsible for selecting an operating point for every ACTIVE and INIT set. The difference between ACTIVE and INIT is that the latter has a previously selected point where resource assignment failed. The quality manager can, if needed, also decide to select a new operating point for an already ASSIGNED set, resulting in an *operating point switch*. Finally, the resource manager is responsible for assigning resources to all sets of state SELECTED and RESELECTED. In case the assignment of an operating point fails, its set state is moved to INIT. The resource manager is also responsible for undoing the resource allocation and for deactivating all applications that were terminated (EXIT) by the user.

As Figure 7.19 additionally shows, that if no feasible operating point(s) can be found by the quality manager, the user again becomes responsible. Consequently, the user can terminate other applications and/or restart the application with another user preference. Similarly, the user becomes involved whenever the assignment fails and the maximum number of QM/RM iterations have been reached. When an assignment fails without reaching the maximum amount of attempts, the penalty vector $\vec{p}$ and the resource vector $\vec{R}$ (as perceived by the quality manager) are modified: with respect to the resource that caused most resource assignment failures during the last resource manager run, the penalty vector reflects an additional 10% penalty, while the resource vector reflects a 10% decrease in availability. In turn, this prompts the quality manager to select a better-assignable operating point.

7.4.2 Operating Point Selection *From Scratch*

The most straightforward way to implement the operating point selection functionality is to take the algorithm for solving the MMKP problem of Section 7.2.2 and to extend it with the histogram feasibility checking approach of Section 7.3.3. In addition, we make minor changes to the algorithm: the multidimensional resource reduction is now based on the actual available platform resources and the initial solution is composed on the points with the lowest *single resource*. These points exhibit the lowest cost with respect to the available platform resources and, hence, have a high probability of producing an initial feasible solution. All these changes will allow the algorithm to find a feasible solution even when the user value is not proportional to the needed resources.

The resulting operating point selection approach is denoted as the *From Scratch* (FS) algorithm as it selects a point for all activated sets at once (i.e. also for the ASSIGNED sets). This improved algorithm is described in Algorithm 14. The *From Scratch* algorithm receives as input (1) the collection of operating points belonging to all ACTIVE, INIT and ASSIGNED sets, (2) an aggregate and histogram-based description of the available platform resources $\vec{R_{avail}} = \{(R_0, R_0^{Histo}), \cdots, (R_k, R_k^{Histo}), \cdots, \}$ and (3) the penalty vector.

After performing a multidimensional resource reduction (line 1), the algorithm selects for every set S_i the point pt_{ij}^0 with the lowest single resource R_{ij}. These points provide the initial solution. Consequently, this initial solution is tested for its feasibility. This includes testing if the solution works from an aggregate resource viewpoint and performing a histogram feasibility check (line 6; for *CheckHistoFeasibility(...)* see Algorithm 13). In case the initial solution is feasible, we visit all points pt_{ij} in descending v_{ij}/R_{ij} order (line 9). For every point, we verify if we can exchange the currently selected point pt_{ik} for a new, feasible point of the same set pt_{ij} with a higher value (line 10). Just like the feasibility checking of the initial solution (line 6), this verification also involves checking the aggregate resource usage and performing a histogram matching. Finally, we end up with a collection of feasible operating points, one point pt_{ij} per set S_i. Depending on the state of the specific set S_i, these points are denoted $pt_{ij}^{SELECTED}$ or $pt_{ij}^{RESELECTED}$.

Algorithm 14: Operating point selection using the From Scratch (FS) approach.

```
Input: $\vec{R_{avail}}, \vec{p}, S_i \in \{S_{ACTIVE} \bigcup S_{INIT} \bigcup S_{ASSIGNED}\}$
Output: $pt_{ij}^{SELECTED,RESELECTED}$
OperatingPointSelectionFromScratch($\vec{R_{avail}}, \vec{p}, S_i$)
foreach $pt_{ij} \in S_i$
  $R_{ij} = DeriveSingleResource(pt_{ij}, \vec{p})$
foreach $S_i$
  $pt_{ij}^0 = pt_{ij}$ with lowest $R_{ij}$
$f_{current} = \{pt_{0j}^0, \cdots, pt_{Nj}^0\}$
if $(\sum_i r_{ijk}^0 > R_k)$ or $(CheckHistoFeasibility(\sum_i r_{ijk}^{0,Histo}, R_k^{Histo}) = 0)$
  $F = \emptyset$, Notify user.
else
  foreach Unused $pt_{ij}$ with highest $v_{ij}/R_{ij}$
    $f_{new} = Exchange(f_{current}, pt_{ij}, \vec{R_{avail}})$
    if $(Feasible(f_{new}, \vec{R_{avail}})$ and $(v_{ij} > v_{ik}))$
      ExchangePoint($f_{current}$, $pt_{ij}$)
      $f_{current} = f_{new}$
      if $(pt_{ij} \in (S_{ACTIVE} \bigcup S_{INIT}))$
        $pt_{ij}^{SELECTED} = pt_{ij}$
      else
        $pt_{ij}^{RESELECTED} = pt_{ij}$
//Selection Succeeded.
```

Although this approach promises to deliver the highest value for the lowest platform resource usage for all user activated applications, some downsides are to be expected.

First, simply consider the case where the assigned operating points each represent the highest value of their respective set and a new application is activated by the user. In case enough platform resources are still available to accommodate this new application's highest value operating point, there is no need to revisit the already assigned operating points. Depending on the ratio of assigned applications versus newly activated applications, there could be a considerable gain in execution time. This approach is explored further in the next section.

Secondly, revisiting previously assigned points in order to optimize the overall value could also lead to a large amount of operating point switches for only minimal overall value gain.

Finally, we have to be aware that there is a difference in approach for checking feasibility and for performing the assignment. Indeed, the quality manager checks the feasibility for multiple operating points at once using the histogram approach, while the resource manager assigns resources on a per set basis (i.e. one application at a time). This phenomenon has been discussed earlier in Section 4.2.2 of Chapter 4.

As Section 4.2.4 shows, assigning two task graphs simultaneously (i.e. denoted as resource co-assignment) improves the assignment success rate for a fixed amount of platform resources.

7.4.3 Single Set Selection with Failure Mechanism

Instead of considering all activated applications for operating point selection, this section describes an approach that only considers selecting and assigning a single ACTIVE or INIT set at a time.

Operating point selection for just a single set should be a lot faster because there are fewer operating points to process and because the amount of feasibility failures are minimized as the feasibility checking and the resource assignment both consider only a single set at a time. In case there are plenty available platform resources, this technique should produce a total value close to that of the *from scratch* approach.

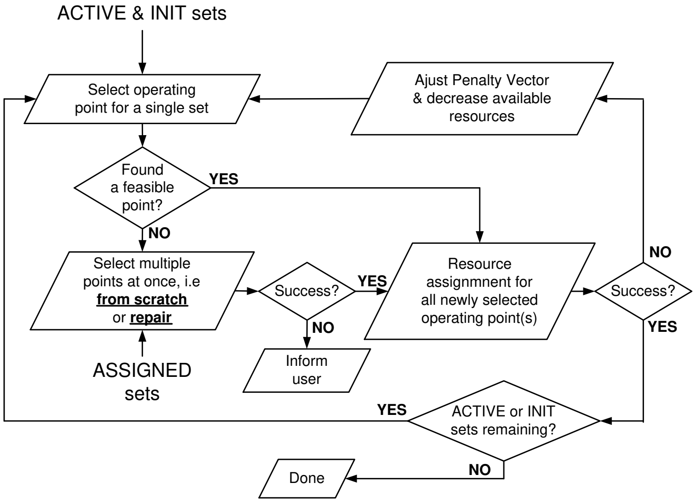

ACTIVE & INIT sets
Select operating point for a single set
Ajust Penalty Vector & decrease available resources
Found a feasible point?
YES
NO
Select multiple points at once, i.e **from scratch** or **repair**
ASSIGNED sets
Success?
YES
NO
Inform user
Resource assignmnent for all newly selected operating point(s)
NO
Success?
YES
YES
ACTIVE or INIT sets remaining?
NO
Done

Figure 7.20: *Flow for single set operating point selection with a failure mechanism. Instead of considering all activated applications from the beginning, we first try to find a feasible operating point for a single application. Only if that fails, we resort to a failure mechanism, which is either selection* from scratch *(Algorithm 14) or using the* repair *selection (Algorithm 15).*

Nevertheless, we still need a way to reconsider the operating point selection of the already assigned applications when it is impossible to find a feasible operating point for the given available platform resources. Here, we explore two options. The most straightforward option is to resort to *from scratch* operating point selection after the single set selection has failed. This option again reconsiders all assigned applications. The second option, denoted as *repair* selection, is to limit the number of already assigned applications that are included in the combined selection process. This effectively means performing a *from scratch* selection with only a limited, carefully chosen number of already assigned applications. This *repair* selection is based on the fact that switching only a few operating points of already assigned applications should be enough to provide the required resources for the newly activated application. In addition, by concentrating on the subset of assigned applications that provide the least value for their respective resource usage, one should still be able to obtain a total value close to the *from scratch* approach. The entire flow is illustrated by Figure 7.20.

The *repair* selection algorithm is detailed in Algorithm 15. *Repair* selection first reduces all selected points of the ASSIGNED sets with the current penalty vector (line 1). The algorithm also creates a set of operating points, denoted S_{REPAIR}, that contains all unassigned sets (line 3). Then, the ASSIGNED set that provides lowest value for the highest resource usage is included in S_{REPAIR} (line 6) and the available resources are recalculated to take the already assigned resources into account. Meaning that repair selection only considers the free platform resources plus the resources assigned to the sets of S_{REPAIR} (line 7). Consequently, we perform a *from scratch* selection with the sets included in S_{REPAIR}. The entire process (line 5-line 8) is repeated until the selection succeeds or all sets were included.

Algorithm 15: Single set selection with failure mechanism

```
Input: $\vec{R_{avail}}, \vec{p}, \{S_{ACTIVE} \bigcup S_{INIT} \bigcup S_{ASSIGNED}\}$
Output: $pt_{ij}^{SELECTED,RESELECTED}$
RepairSelection($\vec{R_{avail}}, \vec{p}, \{S_{ACTIVE} \bigcup S_{INIT} \bigcup S_{ASSIGNED}\}$)
(1)  foreach $pt_{ij}^{ASSIGNED} \in S_{ASSIGNED}$
(2)      $R_{ij}^{ASSIGNED} = DeriveSingleResource(pt_{ij}^{ASSIGNED}, \vec{p})$
(3)  $S_{REPAIR} = \{S_{ACTIVE} \bigcup S_{INIT}\}$
(4)  repeat
(5)      Get unused $pt_{ij}^{ASSIGNED}$ with highest $(R_{ij}^{ASSIGNED}/v_{ij})$
(6)      $S_{REPAIR} = S_{REPAIR} \bigcup S_{ASSIGNED}^{i}$
(7)      Ajust $R_{avail}$ for already assigned resources
(8)      result = OperatingPointSelectionFromScratch($\vec{R_{avail}}, \vec{p}, S_{REPAIR}$)
(9)  until ((Selection Succeeded) or (all $S_{ASSIGNED}$ included))
(10) if (Selection Succeeded)
(11)     Perform Resource Assignment
(12) else
(13)     Inform user
```

Single set selection with a *repair* failure mechanism should work fine in case the user terminates a running application and starts a new one or when a new application is started when plenty of platform resources are available. However, the single set selection does not work in case of a disruptive event, i.e. when the user terminates multiple applications or when the resource manager decides to e.g. shut down some processing resources. In these cases, one needs to revisit all active applications in order to maybe find a better operating point and to optimize the overall system value for this new situation.

Hence, we continue to use the *from scratch* approach in case of disruptive events: i.e. when the number of activated applications either rises or declines drastically or when the amount of platform resources is reduced or increased (caused by a power down or power up of platform resources by the resource manager).

7.4.4 Experimental Setup

The goal of the experimental setup is to assess the collaboration and the integration between the quality manager and the resource manager in a more real-life situation where one or more applications get started and stopped by the user. To this end, we perform three experiments. In all experiments, we take the same 100 randomly generated sets as for the previous experiments (Section 7.3.4). So every set contains between 10 and 40 operating points (22 on average) distributed over three quality levels. For all experiments, we use the platform illustrated in Figure 7.21. This platform contains 6 memory tiles and 10 PE tiles, interconnected by a 4-by-4 mesh NoC using XY routing. The size of the small (S) and large (L) memory tiles is respectively 1024kB and 2048kB, while every NoC link has a capacity of 400MB/s. This platform should contain enough resources so that the quality manager should always find a feasible solution and the resource manager can find an assignment when fitting 15 generated applications. With this basic setup, we perform three experiments.

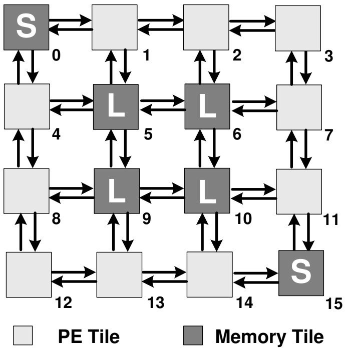
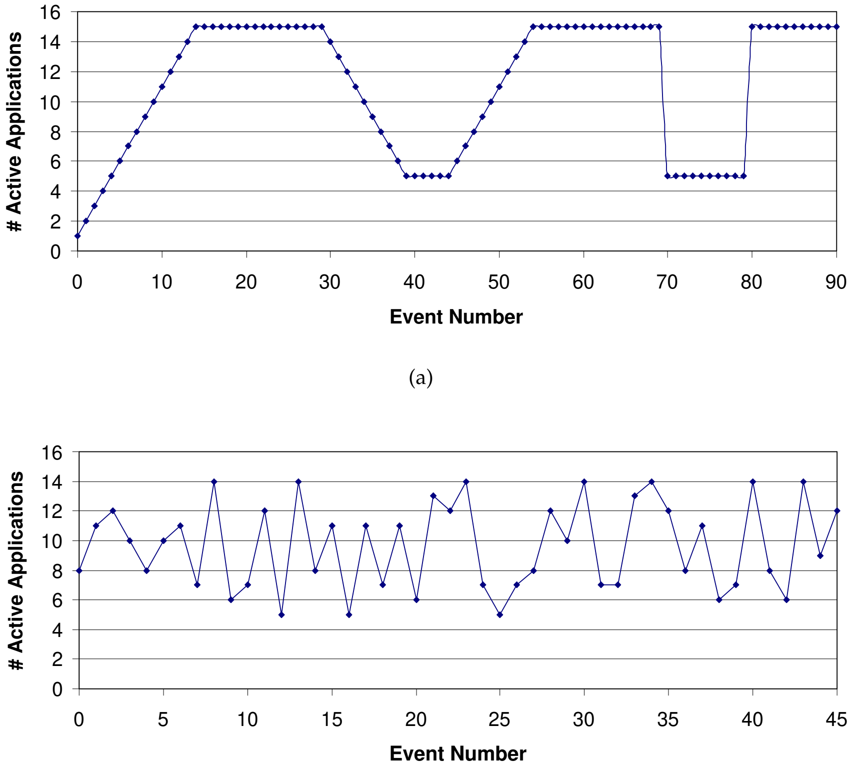


Figure 7.21: Platform quality manager/resource manager experiments. This platform contains 10 PE tiles, 2 small memory tiles (S) and 4 large memory tile (L).

In the first experiment, all user values are proportional with the quality level, meaning that a user value of approximately 1000, 3000 and 9000 is attributed to the respectively low, medium and high quality level. Figure 7.22(a) details the evolution of the amount of activated applications. At every event, a new application is started. In case the total amount of applications remains constant or decreases, a number of already running applications have been terminated. The policy is to terminate the oldest running application first.

The second experiment is similar to the first, except for the application user value with respect to the quality level. All odd applications (i.e. applications starting at an odd event number) have a user value of about 3000, 9000 and 1000 for respectively a low, a medium and a high quality level. So, in this experiment, the user value function sometimes favors a medium or low quality operating point. In those cases, obtaining the highest user value does not require the most platform resources.

The third experiment is similar to the second, except for the evolution of the total amount of activated applications. As Figure 7.22(b) shows, this experiment determines the performance of the operating point selection algorithm in case of contin-



(a)

(b)

Figure 7.22: *Evolution of the total amount of user activated applications. (a) Evolution for Experiment 1 and Experiment 2. (b) Evolution for Experiment 3.*

uous heavy fluctuating amount of applications. In contrast to the previous experiments, a new application is only started to increase the number of active applications (so not at every event).

In every experiment, we will compare the performance of the from scratch selection, denoted as *FS*, with the various forms of single set selection. This includes single set with *from scratch* selection and *repair* selection, denoted as *SingleSet+FS* and *SingleSet+Repair* respectively. This also includes a *SingleSet+Repair* with *from scratch* selection in case of a disruptive event, further denoted as *SingleSet+Repair+FS*. An event is considered disruptive as soon as the difference in number of activated applications from one event to another is larger than 3.

These operating point selection algorithms will be compared based on (1) the provided total user value, (2) the provided value per platform resource usage, (3) their assignment failures, (4) the amount of operating point switches required and, finally (5) the total execution time.

7.4.5 Experimental Results

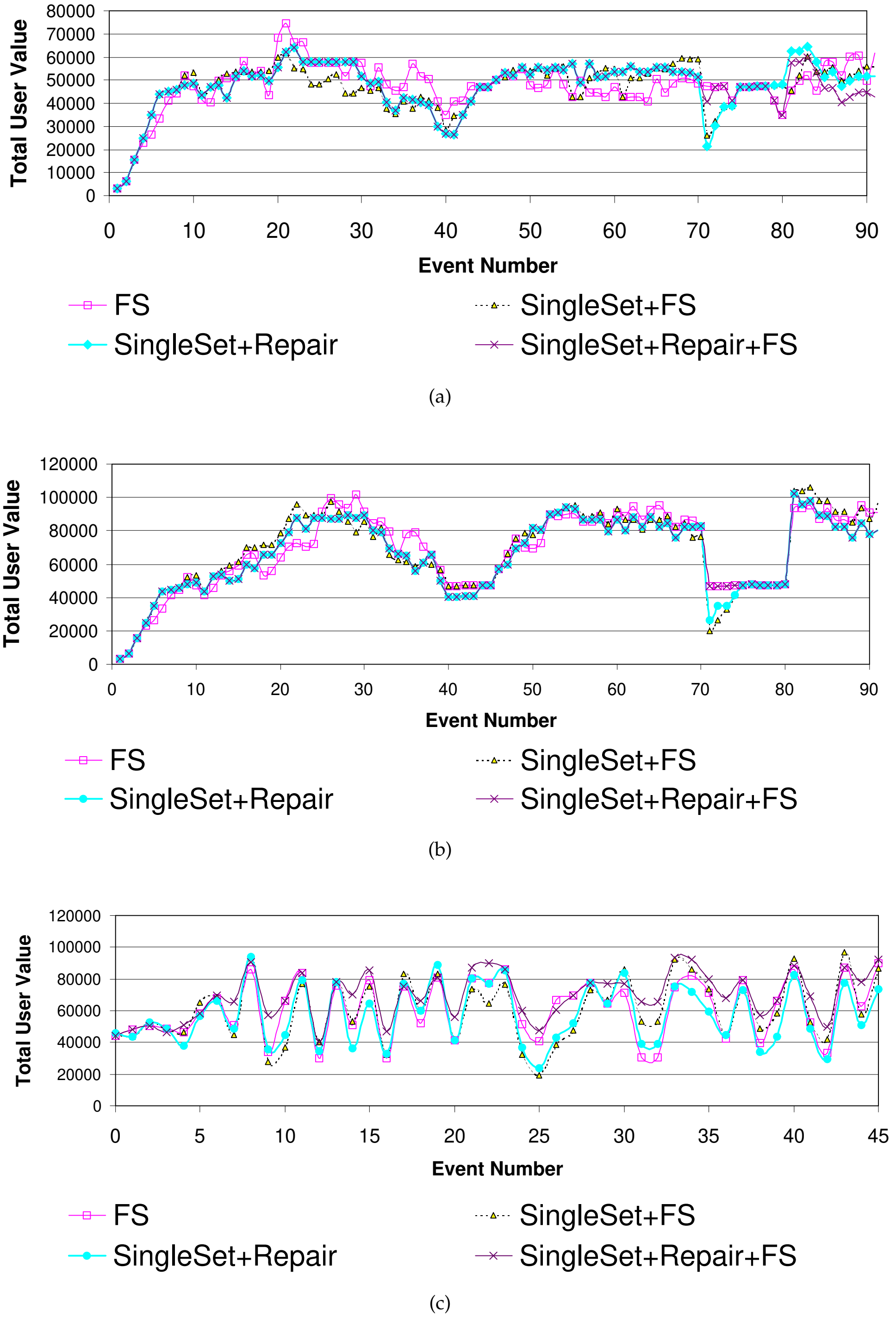
The experiments generate a large amount of data. Figure 7.23 details the evolution of the total value with respect to the evolution of activated applications for all three experiments. Furthermore, we have a look at the average total value in Figure 7.24(a), while Figure 7.24(b) provides the average value per platform load. Figure 7.25(a) details the number of assignment failures, i.e. the number of QM/RM iterations required to get all activated applications assigned. Figure 7.25(b) details the number of operating point switches. Finally, Figure 7.26 details the average execution time for getting all activated applications assigned.

Figure 7.23(a) and Figure 7.23(b) detail the evolution with respect to the application activation of Figure 7.22(a). First of all, we notice that the total value provided by the *from scratch* (FS) selection does not rise as fast as the *single set* selection variants (up until event number 7). Similarly, *from scratch* (FS) sometimes yields a total value below that of the *single set* selection. This occurs between event 50 and 70 for Experiment 1 and around event 20 for Experiment 2. This is striking as the *from scratch* (FS) should be able to find a better total value since it can re-select all activated applications. A closer look reveals that this is caused by the fact that the feasibility checking is done for multiple operating points at once (i.e. resource requirements are mixed), while resource assignment is done per operating point. This means that, as could be expected (see Section 7.4.2), the quality manager overestimates the assignment feasibility, which consequently results in a larger amount of assignment failures (Experiment 1 in Figure 7.25(a)).

As Figure 7.24(a) shows, the average total value for Experiment 1 is the same for both *from scratch* and *single set* selection. For Experiment 2, *from scratch* does provide a higher average total value. This can be explained by the fact that, for half of the Experiment 2 applications, the highest application value does *not* require the highest amount of platform resources. Hence, the resource assignment algorithm is able to handle the feasibility overestimation of the quality manager which, in turn, results in a lower assignment failure rate for *from scratch* selection (Figure 7.25(a)).

One obvious way to solve this feasibility overestimation issue is to perform resource co-assignment, i.e. having the resource manager co-assign all newly selected operating points. This is not an option for two reasons. First, as the results of Section 4.2.4 indicate, co-assignment would increase the execution time of the resource assignment algorithm to an unacceptable level. Secondly, it would almost certainly lead to a unacceptable amount of task *and* data migrations, which would further disrupt the system, while other, less disruptive solutions might still be feasible. The claim that *from scratch* (FS) needs a complete reassignment anyway, does not hold. Often, a few operating points with good v_{ij}/R_{ij} are reselected. This means the resource manager does not have to perform a (re)assignment for these points. In case of co-assignment, this no longer holds.

The second (expected) phenomenon of Figure 7.23(a) and Figure 7.23(b), is that the total value offered by *single set* selection *without* disruptive FS handler is a lot lower (event number 70). Indeed, after terminating a number of applications at once, the *from scratch* selection is able to revisit (and reselect) all remaining applications. By combining *single set* selection with *from scratch* selection in case of disruptive events (i.e. SingleSet+Repair**+FS**), one can combine the best of both worlds. The true value



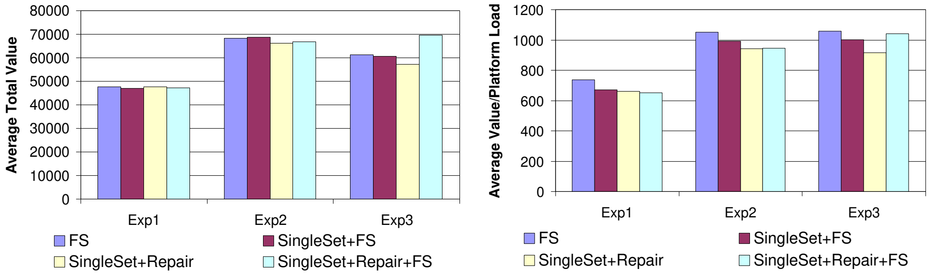
(a)

(b)

(c)

Figure 7.23: *Evolution of total value for operating point selection* from scratch *(FS) and different forms of* single set *selection with respect to the evolution of activated applications (Figure 7.22) for (a) Experiment 1, (b) Experiment 2 and (c) Experiment 3.*

of the *from scratch* selection in case of disruptive events is best seen in the total value evolution of Experiment 3 (Figure 7.23(c)). Indeed, here we see that SingleSet+Repair+FS provides the highest value in most cases. This is confirmed when comparing the average total value for Experiment 3 (Figure 7.24(a)).



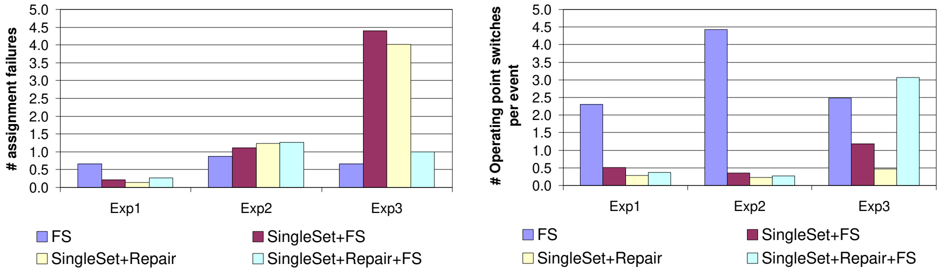
(a) (b)

Figure 7.24: *Performance in terms of (a) average total value and (b) average total value per platform resource usage.*

When it comes to providing the most value per occupied platform resource, we see that *from scratch* (FS) selection outperforms all other operating point selection algorithms. This is obvious as FS re-evaluates all activated applications to find the highest value for the lowest resource usage. Single set selection with the *repair mechanism* yields a total value which is about 10% lower for Experiment 1 and Experiment 2 and about 14% lower for Experiment 3. However, in case of Experiment 3, it is possible to counteract by combining SingleSet+Repair with *from scratch* selection (SingleSet+Repair+FS) as this yields a total value per platform resource usage which is only 2% lower.

Figure 7.25(a) details the number of assignment failures per event, i.e. the number of QM/RM iterations required to select an operating point and perform a resource assignment for every activated set. For Experiment 1, we see that the assignment failures are quite low. As we previously explained, the *from scratch* (FS) selection failure is relatively high due to an overestimation of its assignment feasibility. For *single set* selection in Experiment 2, the assignment failures are a lot higher. This can be explained as follows. In Experiment 1, a newly started application can provide a similar value for a similar amount of resources as a recently terminated application. In Experiment 2, this is no longer the case as some applications provide a high value but only require a medium or a low amount of resources. Experiment 3 shows a very high assignment failure rate for SingleSet+FS and SingleSet+Repair. As Figure 7.22(b) shows, in case of an application increasing disruptive event, on average 3.64 new applications are started. This means that we can consider this as three separate single set selections that each incur an assignment failure rate similar to Experiment 2. The SingleSet+Repair+FS algorithm does not have that problem, because as soon as 3 or more new applications get started, the algorithm resorts to using selection *from scratch*.

Figure 7.25(b) details the number of operating point switches used by the different operating point selection algorithms in order to achieve their value. It is quite obvi-



(a) (b)

Figure 7.25: *Operating point selection and resource assignment result per event with respect to (a) the resource assignment failures and (b) the number of operating point switches.*

ous that the *from scratch* selection introduces the most operating point switches as it re-evaluates all sets at every event. Using *single set* selection (SingleSet) only encounters this problem when it resorts to its failure mechanism. In this case we see that SingleSet+Repair has fewer operating point switches than SingleSet+FS. Intuitively, this is easy to understand as SingleSet+Repair only reconsiders a minimal subset of already assigned applications. For Experiment 3, SingleSet+Repair+FS features more operating point switches than FS, because when the *from scratch* selection kicks in after a disruptive event, it reselects some of the suboptimal previous selections done by the SingleSet+Repair part of the algorithm. This also explains why SingleSet+FS scores relatively high compared to SingleSet+Repair.

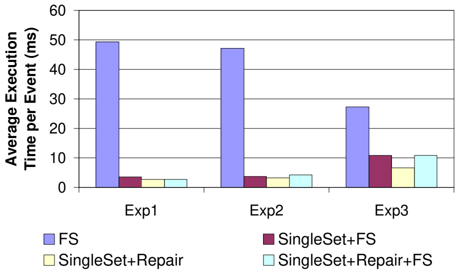


Figure 7.26: *Average total execution time per event for performing operating point selection and resource assignment.*

Finally, Figure 7.26 details the average total execution time per event for performing operating point selection and resource assignment. The number of operating point switches obviously has an effect on the total execution time as it requires to re-assign the switched applications. Two conclusions can be drawn in the given context[7]. First, *from scratch* operating point selection is not an option for a run-time manager in this context as it takes too much time per event. Secondly, the SingleSet+Repair is the

[7] The context being: (1) executing on a StrongARM running at 206 MHz and (2) having to select operating points for up to 15 applications with each application having on average 22 operating points.

only option that, on average, remains below 10 ms for selecting an operating point and performing the resource assignment. Indeed, the time required to start a new application (i.e. creating a new application process) in the Linux operating is in the order of magnitude of 1 ms to 10 ms [138] depending on the hardware platform.

Concluding Remarks
A few conclusions can be drawn from these experiments. First, it is important that the way of checking feasibility in the quality manager is aligned with the way resource assignment is done in the resource manager in order to produce the highest user value. Secondly, using *single selection* with a *repair* failure mechanism as operating point selection algorithm produces good results in terms of user value. Although it provides a lower value per used platform resource (up to 14% lower value than FS), it is up to 5 times faster than the *from scratch* (FS) selection. If needed (depending on the real-life situation), one can always resort to *from scratch* selection in case of a disruptive event. This then guarantees a user value per used platform resource comparable to the *from scratch* selection with an execution time which is still about 2.5 times lower.

7.5 Related Work

The concept of using *Pareto operating points* finds its origin in welfare economics. A conventional definition of Pareto optimality would be: ”A given economic arrangement is efficient if there can be no arrangement which will leave someone better off without worsening the position of others.” [34]. Today, the Pareto concept is used in the industry, as a basis for sociological studies and for setting political policies [34, 75, 116, 205]. Obviously, the Pareto concept is also useful in engineering. In this case, the Pareto operating point set describes a set of system parameterizations that are all Pareto optimal. Hence, a system designer can make trade-offs between all of these solutions instead of having to consider the full range of all the system parameters. This section details related work with respect to using the (Pareto) operating points approach for run-time quality management.

7.5.1 Solving the MMKP

Finding optimal solutions for the MMKP is NP-hard [168]. Any general algorithm that solves the MMKP exactly has a computational complexity which is exponential in the number of sets. Obviously, the constraints for solving the MMKP (e.g. such as compute time, optimality, number of sets, etc.) are given by the context in which the model is used.

Various *Local Search* heuristics, like Tabu search [58] and simulated annealing [65], and genetic algorithms [114] have also been applied to solve the MMKP. However, these and some other proposed algorithms [14, 116] are so computationally expensive that they cannot be used to solve the MMKP at run-time in an embedded system.

Other algorithms [111] are based on branch and bound with linear programming technique. Although the use of linear programming to determine the feasibility of selecting any point from any set reduces the execution time in the average case, it is not feasible to apply these solutions in all practical real-time systems. The execution time of these algorithms increases dramatically with the number of sets [9]. Hernandez's heuristic [168] relaxes the MMKP to a multi-dimension knapsack problem. The failure rate for finding a feasible solution is lower than in [113]. While its solution quality is better, its computational complexity is higher.

Khan's heuristic [113] applies the concept of aggregate resource consumption as measurement to select one operating point for each set. It has a worst-case complexity of $O(ms^2(N-1)^2)$, and it finds solutions with a total value on average equal to 94% of the optimum. Akbar's heuristic [10] first reduces the multi-dimension search space into a two-dimensional one, and then constructs the convex hull of each set to reduce the search space. Its worst-case complexity is only $O(msN + sNlog(sN) + sNlog(s))$. However, this algorithm finds less optimal solutions than [113]. Fast greedy heuristics for solving multi-choice knapsack problems (with several sets, but one resource) also exist for run-time task scheduling on embedded systems [239] and run-time quality-of-service management in wireless networks [28].

An in-depth survey of several MMKP algorithms, comparing their relative complexity and performance is given by Couvreur and Nollet [53]. As Section 7.2 explains, our MMKP heuristic is a variant of the heuristic presented by Khan [113].

7.5.2 Operating Point Selection and Application Quality

Khan et al. [112] investigate how to maximize the *system utility* or total system value when mapping multiple adaptive multimedia applications onto a multimedia system. Their approach is to translate the application quality levels into value for the user by means of a *session utility function*. Their final goal, however, is to optimize the system utility which is a function of the different session utilities. In contrast to our approach, Khan et al. only consider a single (unique) resource vector (i.e. application implementation) per application quality level (although the authors acknowledge that the quality to implementation mapping does not have to be unique). The application resource vectors and the platform resource constraints detail both the required and available resources with respect to the processing elements, memory and bandwidth in an abstract way. In addition, the actual mapping is (implicitly) present in the resource vector. This means that they do not consider the interaction between resource manager and quality manager and, although they abstract the resource needs, they do not consider allocation issues such as fragmentation. When executing on a Pentium 120 MHz workstation with 32MB of RAM, their greedy heuristic provides near optimal solutions in less than 1ms for a problem with 10 applications and 5 resources.

Lee et al. [121] also consider the problem of selecting the right quality operating point for every application within a set of applications. Every application quality has a number of associated resource usage vectors. So different resource usage vectors (i.e. application implementations) can provide the same quality by using a different resource trade-off. However, the actual resource allocation is already included within this resource vector. This means the resource vector describes how much of every platform resource is used. Application quality dimensions are used as input for a *utility function*. This utility function translates the application quality levels into a user value. The overall goal is to optimize the *system utility*, which is defined as a weighted sum of the application utilities. Lee et al. compare three approaches for solving this MMKP. Their most promising run-time approach also uses a penalty vector to reduce the multiple resources associated with an operating point into a single virtual resource. Consequently, the single-resource operating points are sorted according to the utility (value) they provide and the sorted lists of different applications are merged. Their results show that the *local search* heuristic requires tens of milliseconds (Unfortunately, the authors do not provide information concerning the computing architecture used to make these measurements) to provide a near optimal solution for about ten applications with three resources and three quality dimensions. Similar to our approach, this work considers multiple implementations with different resource trade-offs for a single application quality level. However, in contrast to our approach, their solution requires the resource allocation to be included into the operating point, i.e. it does not use a resource manager to flexibly allocate the required resources to obtain a certain quality. This obviously results in a large amount of points. In addition, the authors do not specify how the penalty vector should be obtained as they make an abstraction of the underlying platform.

Pastrnak et al. [170–172] also manages the quality of adaptive applications by means of a quality manager and a resource manager. Each application operating point, generated at design-time, corresponds to a quality setting and a mapping to a *virtual platform* (i.e. an annotated task graph). Hence, the authors do not consider multiple implementations with different trade-offs for a single quality level. Given such virtual platform, it is up to the resource manager to find physical resources with sufficient free capacity. To solve the quality selection problem, the authors reduce the multidimensional resource needs to a single resource by a process denoted as *clustering*. By combining this single resource with a *benefit* figure, provided by the quality manager, the run-time heuristic is able to select the best quality given the available resources. Unfortunately, the authors only provide a high-level description of both the process that verifies if there are enough resources for a certain quality level and the process of actually assigning platform resources. The authors characterized an MPEG-4 decoder with respect to its resource needs for every quality level [170] and used it as a driver application for their quality management algorithm [171].

7.6 Conclusion

This chapter explains the concept of adaptive applications, i.e. applications with multiple quality levels and even multiple application implementations per quality level. Such flexible and scalable applications are easy deployable over multiple generations (with increasing resource capabilities) of a certain MPSoC platform. In addition, for a given platform, such applications can adapt to varying user quality requirements.

This chapter also introduces a quality manger on top of the resource manager. The role of the quality manager is twofold: translating the application quality levels into user values (performed by the *QoE manager*, which is out of the scope of this thesis) and selecting, for every active user application, a suitable operating point (i.e. quality level and implementation, performed by the operating point selection manager).

In order to optimize the user value for a limited amount of platform resources, the quality manager has to solve a Multi-dimensional, Multiple-choice Knapsack Problem (MMKP). To this end, we introduce an efficient and fast algorithm in order to solve this problem at run-time.

Furthermore, in order to be deployable over multiple platforms, the application operating points will have to be described in a way which is not platform specific. Still, the quality manager needs to be able to verify during its selection process if a certain operating point will be assignable. To this end, we introduce a resource histogram. For the application, such a histogram is included as additional design-time information with every operating point. Ideally such a histogram is generated by application design and mapping tools. For the available platform resources, such a histogram is generated by the resource manager at run-time. These resource histograms allow the quality manager to check, with a high degree of certainty, the feasibility of the application operating point resource needs with the available platform resources. Indeed, without a resource histogram, the chance that a *feasible* point is assignable ranges from 13% to 38% respectively for a high and a low platform load. With a resource histogram, this chance increases to respectively 100% and 96%. The fact that the quality manager can better assess the feasibility of an operating point also improves the operating point selection and resource assignment speed up to a factor 5.4 (again depending on the platform load).

Finally, we evaluate the collaboration and the interaction between the quality manager and the resource manager. Here we conclude that integrating the MMKP algorithm as is does not produce the desired results in terms of collaboration and in terms of execution time. Indeed, we show it is important that the way of checking feasibility in the quality manager is aligned with the way resource assignment is done in the resource manager in order to produce the highest user value. We also show it is important to tune the operating point selection algorithm: only performing operating point selection for a single set at a time, with a selective fall-back mechanism in case of failure, produces good results in terms of total user value and user value per platform resource usage. Indeed, this tuned operating point selection algorithm provides less value per used platform resource (up to 14% less value than FS), it is up to 5 times faster than the *from scratch* (FS) selection. If needed (depending on the real-life situation), one can always resort to *from scratch* selection in case of a disruptive event. This then guarantees a user value per used platform resource comparable to the *from scratch* selection with an execution time which is still about 2.5 times lower.

Chapter 8

Conclusion & Future Work

The MPSoC revolution is essentially driven by the customers' desire for having ever more applications and services supported and running in their mobile phone, PDA, settop box, etc. without paying a premium price. All companies involved in the value chain of these devices consider these services a way to (1) differentiate themselves from the competition and (2) segment the market. By putting a flexibly programmable MPSoC platform that combines a high compute performance with a low power consumption in those everyday embedded devices, manufacturers are able to provide that ever-increasing number of complex user applications at an ever-increasing pace. At the same time, it also allows them to keep the cost of those devices under control. In beating the competition through service or application differentiation, a short time-to-market is essential. This means that MPSoC platform programmability is important as it determines the time needed to create new applications. This also means that combining multiple applications in a predictable way (i.e. without inter-application interference) and using the same (scalable) application for different market segments (with different platform capabilities) is equally important.

This results in a set of key MPSoC requirements that any MPSoC platform should address: flexibility, programmability, scalability, predictability, a high performance and a low power consumption. The run-time manager plays an important role in tackling each of these requirements. Indeed, by providing application quality management and platform resource management, the run-time manager directly addresses the flexibility and scalability issues. This also means the run-time manager is responsible for assigning compute performance and for controlling the power consumption.

By minimizing the inter-application interference and by providing a hardware abstraction layer, the run-time manager respectively ensures predictable behavior and programmability.

Besides providing a comprehensive survey of the contemporary MPSoC run-time management functionality and its implementation space, this thesis introduces several novel algorithms, mechanisms and components for performing run-time management of multiple adaptive applications running on a heterogeneous multiprocessor SoC. This includes run-time FPGA fabric management combined with efficient and fast resource allocation, task migration policies and mechanisms, a NoC communication management policy and mechanism, and a quality management component for handling adaptive applications. In addition, we describe a real-life MPSoC proof-of-concept implementation and demonstrator.

Run-time management research is likely to be a *never-ending story* as the run-time manager interfaces with the applications on the one side and with the platform services on the other side. As a result, there will be a need to provide an adapted run-time manager as long as platforms continue to evolve and as long as new applications and user services keep popping up.

So, this chapter provides a concluding overview of the work covered in this thesis and provides a peek into the future of MPSoC run-time management. It is organized as follows. Section 8.1 briefly summarizes the thesis and details how our run-time management contributions address the key MPSoC requirements. Section 8.2 provides both an overview of the *short term* improvements with respect to the presented work, and a vision on the longer term directions for multicore run-time management research.

8.1 Summary

The run-time manager is logically located between the application and the platform. The main goal of the MPSoC run-time manager is to *match the needs and the properties of the application in a fast, efficient and flexible way with the MPSoC platform services and properties in order to execute multiple applications while minimizing inter-application interference.* Such a run-time manager is typically composed of a quality management component, a resource management component and a run-time library (Figure 8.1). Each of these components play an important role in enabling or fulfilling the identified MPSoC requirements: reduction of application design cost, platform and application scalability, flexibility, predictability and combining high performance with low power operation.

In order to match the needs of the application (and its user) with the available platform services in a fast and efficient way, one requires a quality manager and a resource manager. The quality manager considers the application user requirements to derive the application resource requirements, while the resource manager takes these resource requirements and matches them with the available platform resources.

With respect to our quality manager, we present a heuristic capable of selecting an appropriate application quality level (Figure 8.1a). To work efficiently, this heuristic is closely linked to the resource assignment heuristic of the resource manager (Figure 8.1b). The latter assigns an application task graph with associated properties to an MPSoC platform architecture graph. The efficiency of the heuristic comes close to an algorithm that explores the entire solution space while it only requires a fraction of the execution time (i.e. it is fast). In order to retain the run-time flexibility with respect to the changing application requirements and the availability of platform resources, the run-time manager should be capable of changing both the application quality level as well as the application's resource allocation without causing (too much) inter-application interference.

The resource manager is extended with a task migration policy (Figure 8.1c). Run-time task migration also enables the run-time manager react to changing run-time conditions. The associated run-time task migration mechanisms are tuned both towards the specific platform properties (i.e. heterogeneous processing elements) and limitations (i.e. limited amount of tile memory) as well as towards the specific application properties (i.e. considering an application as a pipeline of tasks).

The resource assignment heuristic is also extended to consider the platform properties. This is illustrated by (1) taking into account the specific properties of on-chip FPGA tiles and (2) by considering the properties of the on-chip interconnect. With respect to managing FPGA fabric, we introduce a novel algorithm that exploit the capability of FPGA fabric to create a configuration hierarchy. This allows the run-time manager to match the needs of the application with the capabilities of the platform (Figure 8.1b). With respect to managing the NoC communication, we introduce a reactive algorithm that matches the communication needs of application with the communication services provided by the target platform architecture (Figure 8.1d). In addition, the algorithm minimizes inter-application interference. This communication management algorithm builds on top of two platform services: a communication statistics collector to monitor NoC traffic and an injection rate control mechanism to execute its decisions.

As there are multiple ways of implementing a run-time manager, we also provide an overview of the run-time management implementation space. In addition, we provide an overview of the contemporary academic and industrial run-time manager implementations.

We now give a brief summary of every chapter.

Chapter 2

As the run-time manager sits at the heart of the operating system, it monitors and controls critical platform resources such as the (heterogeneous) processing elements, the (on-chip) memory hierarchy and the platform communication infrastructure. The run-time manager is composed of multiple interacting components (Figure 8.1): the *quality manager*, responsible for application interaction with respect to the quality needs of the user, the *resource manager*, responsible for efficiently managing the platform resources with respect to the requirements of the different executing applications and the *run-time library component*, that provides the necessary platform abstraction functions used by the designer for creating the application and called by the application at run-time. As there are multiple ways of implementing an MPSoC run-time manager, Chapter 2 introduces a run-time management implementation design space. This chapter also briefly details some contemporary industrial and academic multiprocessor run-time management solutions and takes a peek into the future. Clearly the contemporary industrial MPSoC run-time management is mainly

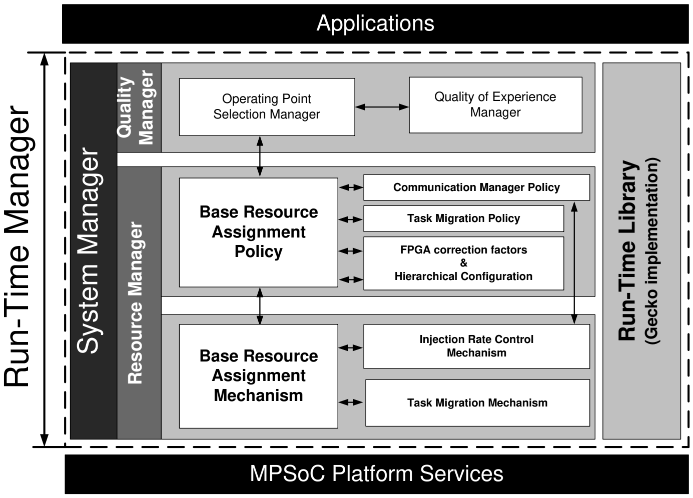

Applications
Run-Time Manager
System Manager
Quality Manager
Operating Point Selection Manager
Quality of Experience Manager
Resource Manager
Base Resource Assignment Policy
Communication Manager Policy
Task Migration Policy
FPGA correction factors & Hierarchical Configuration
Base Resource Assignment Mechanism
Injection Rate Control Mechanism
Task Migration Mechanism
Run-Time Library (Gecko implementation)
MPSoC Platform Services

Figure 8.1: *Detailed view of the contributions, the run-time management components and their algorithms discussed within this thesis.*

focused on providing RTLib functionality, i.e. on providing the application designers with a hardware abstraction layer. The overall MPSoC run-time management trend for the future is quite clear: moderate distribution of the run-time manager over the platform resources, more platform services to support the run-time manager and more configurability or adaptation towards the application. Finally, Chapter 2 also serves as a reference for the rest of the thesis as it puts the run-time manager functionality, introduced in this thesis, in the correct perspective.

Chapter 3

This chapter focuses on the flexibility and performance requirements as it details a fast and efficient run-time task assignment heuristic capable of handling fine-grain reconfigurable hardware (i.e. FPGA fabric) and exploiting its properties. This involves introducing a novel run-time task assignment concept denoted as *hierarchical configuration*. Hierarchical configuration enables easy time-multiplexing of FPGA fabric and it improves the spatial task assignment freedom resulting in a more efficient usage of platform resources. The generic heuristic produces very good results in terms of assignment success rate, quality of the assignment and speed of the algorithm, compared to algorithm that explores the full solution space. Adding support for creating a configuration hierarchy significantly improves the performance of the run-time task assignment algorithm. In some cases, the average heuristic success rate improvements even exceed searching the full solution space without hierarchical configuration, while using only a fraction of the execution time.

Chapter 4

Flexibility requires taking varying run-time conditions (e.g. new user requirements, new incoming applications, etc.) into account for deciding the best allocation of platform resources. Concretely this means the run-time manager needs a way to revise the initial mapping of one or more already executing applications. Hence, this chapter introduces *run-time task migration* capabilities, i.e. a way to move tasks to a different (heterogeneous) processor without the need to completely stop and restart the application. With respect to heterogeneous task migration, we consider four issues: the run-time task migration policy, the HW/SW task migration issue, i.e. design-time and run-time components needed to migrate a task from FPGA hardware to ISP software, the task migration mechanism tailored to the Networks-on-Chip environment and, finally, the migration initiation issue, i.e. how a cooperative migration request from the run-time manager are detected and handled by the application. We show that adding task migration capabilities to the resource assignment heuristic of Chapter 3 clearly improves the assignment success. In most cases, the success rate of a migration-enabled task assignment heuristic exceeds searching the full solution space without considering migration, while using only a fraction of the execution time. With respect to HW/SW task migration, we detail the design-tools and the required run-time environment illustrated by an MJPEG video case study. With respect to the mechanisms, we introduce two novel task migration mechanisms and we detail a novel poll-free task migration initiation technique.

Chapter 5

This chapter focuses on the predictability challenge by providing a *reactive* platform communication management scheme in order to minimize inter-application communication interference. Indeed, as the NoC is a shared communication medium, one has to minimize the impact different unrelated applications have on one another. In case of imperfect platform virtualization or imperfect application characterization, one requires a reactive communication management scheme. Such a scheme requires three components: a way to monitor the on-chip communication, a decision making entity and an actuator to execute and enforce the decisions. A control loop is built feeding the output of the monitoring into the decision-making process and using its output to control the traffic shaping. In this chapter, we study monitoring at the *connection-level* combined to a *global* decision making under the control of a run-time manager. We show how a global decision making algorithm implemented on a StrongARM processor can efficiently manage the injection rate controller actuators in order to re-shape traffic at run-time to reduce NoC congestion. Taking a global decision-making approach permits an efficient reactive control of communication of a pipeline of tasks. In contrast, Marescaux [132] studies a system, based on the same experimental setup, where monitoring is performed at the link-level, combined with distributed decision making.

Chapter 6

This chapter combines the previous concepts into a set of real-life demonstrators, denoted as *Gecko* demonstrators. The work on the MPSoC proof-of-concept Gecko platform emulators started at IMEC in 2001 as an ambitious engineering project. This project included run-time management of the MPSoC platform, with a particular focus on the NoC, on dynamic partial reconfiguration of the FPGA computing resources, and on heterogeneous task migration. This chapter essentially shows how the Gecko run-time manager builds on top of the platform services and how it interacts with the (hardware) RTLib functionality of every slave node and with the applications. This includes a description of the run-time management structures. Finally, we also detail the envisioned user scenario for such a platform and we provide a brief transcript of the *Gecko* demos.

Chapter 7

This chapter focuses on the scalability challenge, as it describes how the quality manager supports scalable applications, i.e. applications with multiple quality operating points. Such scalable applications are easy deployable over multiple generations of a certain MPSoC platform. For a given platform, such applications can adapt to varying user quality requirements. The run-time quality manager is responsible for selecting the right operating point for every active application in order to optimize the total user value, while considering the platform capabilities. This means the quality manager requires an efficient and fast algorithm to select the best operating point for every active application given the platform resource constraints. We introduce an algorithm to solve the Multiple-choice Multi-dimensional Knapsack Problem. As the operating point information is provided in a way that is not platform specific, the quality manager will have to reconcile a *platform independent* description of the application resource needs with a *platform specific* description of available resources. To tackle this problem, we introduce a resource histogram that describes, for every operating point, the distribution of the required resources. At run-time, the resource manager constructs such a histogram for the available resources. By comparing both histograms, the quality manager can verify the operating point assignment feasibility with a very high degree of certainty: without a resource histogram, the chance that a *feasible* point is indeed assignable ranges from 13% to 38% respectively for a high and a low platform load. With a resource histogram, this chance increases to respectively 100% and 96%. The fact that the quality manager can better assess the feasibility of an operating point also improves the operating point selection and resource assignment speed up to a factor 5.4 (again depending on the platform load). Finally, as the quality manager relies of on the resource manager to assign the platform resources, both run-time management components have to closely cooperate. In a situation that simulates a user starting and stopping a set of adaptive applications (generated with the PSFF tool of Appendix A), we investigate the interaction between the quality manager and the resource manager. As a result, we refine the operating point selection algorithm to provide the resource manager an operating point in a faster and more accurate way without either sacrificing too much user value or additional platform resources.

8.2 Future Work

This thesis obviously provides only a small piece of the ever-evolving overall multiprocessor SoC run-time management puzzle. Each chapter comes with its own set of assumptions and limitations. Providing a complete productizable solution requires validating these platform and application assumptions (or solving the remaining limitations).

First, Section 8.2.1 identifies the most prominent lacunae of every chapter. Then, Section 8.2.2 provides a more in-depth look on the task scheduling lacuna, i.e. making decisions on the temporal task ordering. This includes determining the effect of task scheduling on resource assignment.

Finally, we show a broader and more long term view on the issues of MPSoC run-time management. As the run-time manager is squeezed between the application layer and the platform layer, it requires continuous tuning towards new and emerging platforms and their associated platform services and towards the needs of new applications. Section 8.2.3 briefly details the future collaboration of design-time tools and run-time management. Section 8.2.4 provides an outlook of the many-core future and the role of the run-time manager. Finally, Section 8.2.5 details the (potential) role of run-time management for managing SoC platforms in the deep deep sub-micron era.

8.2.1 Lacunae

Because the research focus of the supporting organization or funding authority tends to shift over time and because time, space and money are all scarce resources, every chapter contains one or more lacunae. This section briefly details the most prominent[1] ones.

Chapter 2
The description and understanding of the run-time management implementation space is only the beginning. The next step is to create a methodology for searching this space for a suitable run-time management implementation given a set of MPSoC requirements, boundary conditions and cost functions.

Chapter 3
The applicability and performance of the resource assignment algorithm should be explored for a wider variety of platforms (e.g. platform size, topology, heterogeneity, etc.), applications properties (e.g. size of the task graphs) and mapping cost functions. In addition, the effect of explicitly adding different forms of scheduling should investigated (more details in Section 8.2.2).

Chapter 4
The most prominent lacuna is the coupling of the actual mechanism task migration cost to the task migration policy. Furthermore, one should explore and classify the different forms of task migration mechanisms with respect to NoC-based platforms. This includes e.g. investigating a deeper collaboration between design-time migration analysis and run-time execution.

Chapter 5
Combined with the work of Marescaux [132], this chapter only details two points in the communication management design space. The next step would be to combine and/or compare these approaches. In general, a methodology to traverse this design space in a more systematic way is needed.

Chapter 7
The next step for quality management would be to combine the presented global quality manager and the resource manager with a local, application-specific quality manager. In addition, one should come up with a matching operating point switching mechanism. Also here, there should be a deeper exploration of the collaboration possibilities between, on the one hand, design-time analysis and mapping tools and, on the other hand, the quality manager.

8.2.2 MPSoC Task Scheduling

The way an application, consisting of multiple communicating tasks, is assigned to the different computing resources of an MPSoC platform is quite critical to its

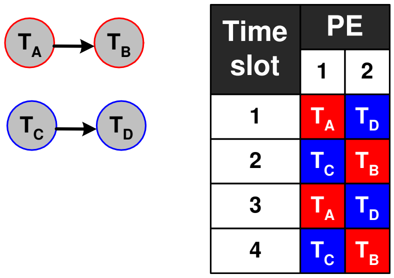
[1] This also means the ones closest to my heart.

performance. However, next to the resource assignment problem there is a PE resource scheduling (or task scheduling) problem. This entails, for example, determining how to interleave tasks executing on the same processing element. This section determines the potential impact of this resource scheduling problem on the various components of the run-time management system.

We briefly discuss the impact of scheduling on resource assignment. Here, we distinguish two phenomena related to assigning multiple tasks to a single PE: the *co-scheduling* issue and *PE cache affinity* issue. In addition, the relation between PE scheduling and FPGA fabric is briefly discussed.

Impact on Resource Assignment: Co-Scheduling

One could argue that PE scheduling in an MPSoC is nothing more than deciding on the assignment of a task set to a set of processors in order to meet a certain criterion like e.g. maximizing processor utilization, minimizing the response time or minimizing the inter-node communication. Once all tasks have been assigned, it is up to a *PE local scheduler* to manage the temporal ordering of its assigned tasks. However, scheduling heavily communicating tasks on separate processors in a parallel system does not automatically produce the expected speed-up. Indeed, a task scheduling mismatch combined with communication and synchronization overhead, could even result in a slower execution.



Time slot	PE	
	1	2
1	T_A	T_D
2	T_C	T_B
3	T_A	T_D
4	T_C	T_B

Figure 8.2: Naively scheduling multiple communicating tasks on an MPSoC system can mitigate the available parallel processing power.

Example 8.1: How parallelization does not lead to speed-up (Figure 8.2). Consider two applications, each containing two communicating tasks. The first application is composed of tasks T_A and T_B, the second application is contains tasks T_C and T_D. T_A and T_C are assigned to PE 1, while T_B and T_D are assigned to PE 2. Each PE is timeshared with T_A and T_D executing in the odd numbered time slices, while T_B and T_C run in the even numbered time slices. When closely synchronized, T_A, sends a lot of messages to T_B. However, T_B is unable to respond since its time slice has not yet arrived. Consequently, after the first time slot is over, a task switching occurs and T_B is able to receive and process the data received from T_A. Again, T_A is unable to either receive the results or to continue feeding data to T_B until it is scheduled again in the third time slot. This simple example illustrates how PE scheduling reduces the advantage of having parallel processing power.

As this issue also occurs in parallel & distributed systems, Ousterhout [165] introduced the *co-scheduling* or *gang-scheduling* concept to solve this kind of problem. Co-scheduling takes inter-task communication into account when scheduling tasks. This means that communicating tasks of the same application are scheduled in the same time-slice. In addition, the time-slices of the different processors need to be synchronized (for example by means of a broadcast message). This technique guarantees the desired parallelism. Figure 8.3 illustrates this solution.

The main problem with using co-scheduling is *fragmentation*. This means that, depending on how the application is parallelized, some PEs are left idle during a certain time slot. Another type of fragmentation is caused by load imbalance. The load difference and dependencies between tasks is a result of the chosen design-time parallelization and inter-task synchronization. A load imbalance potentially affects the whole parallel application, since each task requires a different amount of time to complete and the entire application is restricted by the slowest task.

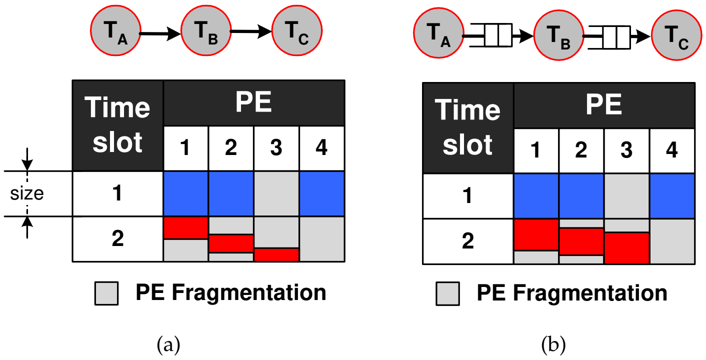


Figure 8.3: Co-scheduling. (a) Strict co-scheduling example. (b) Loose co-scheduling by insertion of communication buffers.

Example 8.2: Co-scheduling and resource fragmentation (Figure 8.3(b)). Consider task T_A being responsible for providing T_B with input data. If providing this input takes a long time, task T_B will be blocked and, hence, is not be able to make full use of its time-slice. So, in general, following a strict co-scheduling policy leads to sub-optimal system utilization (i.e. resource fragmentation). One possible solution is to loosen the synchronization between the tasks by inserting memory buffers between them. This will amortize the processing time variations and will enable a more pipelined processing approach which reduces PE fragmentation at the cost of extra memory usage. It will be up to the designer to provide this trade-off, while it will be the responsibility of the run-time manager to select the right application operating point and assign the tasks to the right resources.

In the context of parallel & distributed systems, several techniques have been proposed to relax the strict co-scheduling policy in order to reduce PE fragmentation.

Feitelson et al. [73] employ a processor control scheme called Distributed Hierarchical Control (DHC). This scheme enables synchronized task switching, effectively allowing co-scheduling. They describe a technique to quantify and reduce the waste of PE resources, caused by the fact that the gang size does not match the number of processors. The main idea of this technique is that small gangs can execute on the leftover processors from the larger gangs. Zhou et al. [243] depict a technique to reduce fragmentation caused by idle processing power. They use re-packing of tasks (i.e. shifting all tasks of a certain gang to another time slot), whenever a number of tasks in a certain time slot are terminated. In addition, they allow tasks to run in multiple time-slots. Frachtenberg et al. [74] introduce a methodology called *Flexible CoScheduling* to reduce fragmentation caused by load imbalance. This methodology detects load imbalance by monitoring the communication behavior of applications and defining metrics for their communication performance. Wiseman et al. [238] introduce a technique called pair gang scheduling. This technique tries to optimize system usage by combining an I/O bound gang with a CPU bound gang in the same time-slot on a single node. The reason for such matching is that these task gangs will not interfere with each other.

Recently, Kumar et al. [118] illustrate this scheduling issue for mapping a set of task graphs of streaming applications onto an MPSoC platform. The authors show, using some synthetic examples, that fully exploiting the computing performance of the system, while being able to handle dynamic varying load and achieving predictable performance is not trivial. Just adding the amount of cycles required for every task does not hold due to communication dependencies with tasks on other PEs and due to the decision making of the local scheduling mechanism. They conclude that the best resource utilization and predictability is achieved with a static schedule of tasks on a certain PE. However, a static schedule requires more design-time analysis. A round-robin task scheduler (with skipping) is better suited to handle task dynamism. These conclusions also have to be taken into account when performing resource assignment.

Overall, the above scheduling techniques have to be revisited in the (dynamic) MPSoC context. This not only means determining the effect of taking different PE properties into account with e.g. different context saving and restoring overheads, it also means considering the potential processor-versus-communication resource co-scheduling.

Impact on Resource Assignment: PE Affinity

In a parallel system, it might be more efficient to execute a task on one processor than on another. This is called processor affinity. One can distinguish three types of PE affinity that a certain task can have: speed affinity, resource affinity and cache affinity. As Squillante et al. [208] explain, the affinity of a certain task for a particular processor can have several causes. *Speed based affinity* occurs, for example, in a heterogeneous environment where a task can execute faster on a certain type of PE. This type of affinity is considered in Chapter 3. *Resource based affinity* mainly concerns the resources associated with the processors, rather than the processor itself. A task that requires a certain resource, can only execute on a processor that is associated with such a resource. This, for example, entails considering processor-local L1 memory or specific I/O related to a processor. The main subject of this section is the *cache-based affinity*. Most contemporary multiprocessors make use of a cache memory. A cache memory is a small on-chip memory that automatically retains recently used memory data or instructions. Having data available in L1 cache memory avoids accessing the much slower L2 or off-chip memory. The cache memory concept works because of the locality property. This property describes how most tasks tend to execute the same instruction a number of times (e.g. in a loop) on the same area of data (e.g. part of an image). During a task context switch, part of the cache memory contents used by the previous task is overwritten in order to store the contents coming from the main memory for the new (current) task. This overhead, called cache corruption, can cause performance degradation.

So in a multiprocessor environment, depending on the relevant tasks data still present in a processor's cache memory, it might be more efficient to execute a task on a certain PE instead of on another one of the same type. In contrast to the other types of processor affinity, cache-based affinity is time-dependent.

Squillante et al. [208] show that ignoring the cache affinity when assigning tasks in a multiprocessor environment can have a significant effect on the individual task performance (i.e. it takes longer for the task to execute), due to the fact that cache memory needs to be restored. In addition, ignoring cache affinity when scheduling a single task also degrades the performance of the whole system. This is caused by the fact that restoring the cache memory contents requires accessing main memory, which in turn can cause bus contention (i.e. other processors in the parallel system compete for the same bus resource). In addition, the cache-based affinity concept can create interesting assignment and scheduling dilemmas.

Example 8.3: Cache-based affinity scheduling dilemma.
Suppose processor A is running a certain task, when a higher priority task, previously running on processor B (currently unavailable) becomes runnable. In case the higher priority task is scheduled immediately on processor A (i.e. current task is pre-empted), performance degrades due to cache corruption. On the other hand, waiting until processor B becomes available means the system does not really exploit the parallelism offered by the parallel system. The Linux SMP scheduler solves this problem empirically by using a rough estimate of the time it takes to overwrite the cache content, called cacheflush_time. If this cacheflush_time is bigger than the average time a task is executed on a processor, no pre-emption is performed.

In conclusion, PE affinity will have an effect on the assignment of the task graph to the architecture graph. This affinity factor was currently not taken into account in the presented resource management algorithms. In addition, one should, for example, investigate the role of application design-time information in relation to the PE affinity concept. This means that, instead of relying on simple rules of thumb, the resource manager would rely on PE affinity information included in the design-time mapping information and run-time application state (provided by the RTLib) to make its assignment decisions.

Example 8.4: Scheduling with a memory hierarchy (Figure 8.4).
One could translate the cache affinity concept to an MPSoC assignment problem with a scratchpad memory hierarchy. In the considered assignment problem, there are only two PEs to accommodate three tasks, T_B should either be assigned to Tile 1 or Tile 2. However, the task graph details that there is intense communication between task T_B and T_C through a block of shared memory. In addition, T_B and T_C both read and write from the shared memory region. In contrast, the communication between T_A and T_B is handled by simple FIFO communication. In that context it would be an advantage (if possible from a PE load viewpoint) to assign both T_B and T_C to Tile 2. This way, the inter-task communication can be kept tile-local.

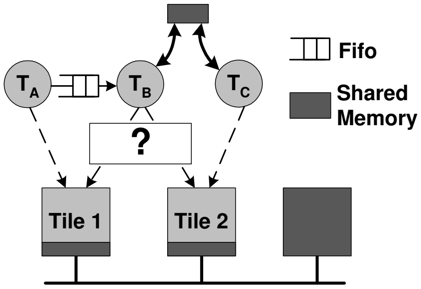


Figure 8.4: Memory affinity influence on task assignment.

Scheduling FPGA Fabric

The algorithms presented in Chapter 3 only considered placing a single task per reconfigurable hardware tile. However, one could consider assigning multiple hardware tasks (or soft IP cores) to a FPGA fabric tile in a time-multiplexing fashion. This means that configuration latency (i.e. task setup time) would become an issue. The main question is: how would the task assignment algorithm take this reconfigurable hardware configuration latency into account.

Consider the following situation. A hardware task needs to be assigned to a platform with two identical reconfigurable hardware tiles. Each tile already contains a hardware task. What FPGA tile should the assignment algorithm favor? A possible answer would be to select the tile which will ensure a minimal task context switch latency. This can be achieved by looking at configuration techniques that only consider the configuration difference between two tasks. The following authors have considered such techniques to reduce task configuration latency.

Charlwood et al. [212] present an idea denoted as *just-in-time reconfiguration*. This strategy will analyze the relationship (regarding routing resources) between the already present hardware configuration and the new configuration and will defer all unnecessary configuration. The overlay technique, described by Kennedy et al. [109] is based on the same idea. In order to speed up the task placement, they created software to analyze the differences between hardware tasks. This means that placing a new task only requires loading the differences between the new task and the already running task instead of the complete configuration. In addition, they describe a reconfigurable architecture that contains a small amount of memory and an associated controller at the top of each column of configuration frames. The memory contains the changes to go from one task to another. Similarly, Chen et al. [41] have simulated the impact of configuration reuse on scheduling reconfigurable hardware tasks. Their results also show that it is beneficial for the scheduling algorithm to take configuration reuse into account when placing hardware tasks.

So, in conclusion, a multiprocessor task assignment algorithm considering to assign multiple tasks to one single-context FPGA tile should take the redundancies between reconfigurable hardware tasks into account when deciding on the assignment of a particular hardware task. In a sense, this concept is closely related to the PE affinity concept.

8.2.3 Design-time Mapping and Run-time Management

In the embedded systems domain, there has always been a division of work and a collaboration between design-time tools and run-time managers. There is still quite some work to investigate and improve the collaboration between application design-tools and run-time management components.

In the short term, the presented techniques for task migration and operating point switching ideally require the existing design-tools to (semi) automatically place migration points and points for operating point switching in the code. This includes providing a (platform independent?) cost function or estimate to guide the run-time manager.

In the longer run, the division of work between design-time tools and run-time manager not only requires a view on the evolution of applications and platforms, but also on the economic context. Applications are becoming more complex and, just like complex MPSoC platforms, will be constructed using third-party components and services in order to prevent an *application* design productivity gap. Platforms are also evolving from *multi-core* to *many-core* (further discussed in Section 8.2.4). And finally, economics play will play an increasingly important role in the context of what can be calculated and decided at design-time and what decisions need to be postponed to run-time. In an environment where time-to-market is essential, where new applications pop-up in a fast pace and where applications and services can be downloaded from any source, one has to rely more on run-time management to *make things work*. This shift from *design-time to run-time*[2] requires researchers (or research teams) that can perform this cross-layer optimization: i.e. from design-tools down to platform services.

Also for fine-grain reconfigurable hardware manufacturers and their respective EDA tool designers, the role of design-tools and run-time managers will need to be revised. As todays' FPGA synthesis tools mainly focus on creating the platform and providing the run-time manager with the memory map of the *fixed platform*, future FPGA tools should come up with a set of platform templates or platform operating points (just like we have discussed application operating points in Chapter 7). In

[2] Either run-time management of platform services

this case, the run-time manager would have to select the best *platform operating point* given the set of active applications. The cost of selecting one point would be proportional to the reconfigurable granularity. Therefore such flexible platforms would probably contain a wide variety of predesigned configurable blocks glued together with fine-grain reconfigurable logic.

8.2.4 The *Many-Core* Run-Time Management Future

How will the future of multiple processing elements on a single chip evolve? For the coming ten years, the ITRS roadmap predicts a ten-fold increase in both the general purpose (main) processing elements and the application-specific processing elements (denoted *DPE*) present on SoC platforms. Indeed, by 2016 a many-core platform would contain about 12 general purpose porcessors and 100 application-specific processors. This *many-core era* will still be fueled by scaling silicon technology[3]. Recently, Intel has shown it is capable of creating such an 80-tile Tera-scale processor [226].

Figure 8.5 illustrates *my two cents* at predicting the (many-core) future. Contemporary embedded devices containing multiple (heterogeneous) processing elements in order to provide the required processing power within the power budget, while maintaining some degree of run-time programmability and flexibility. At the other end of the spectrum, we also experience the rise of (symmetric) multicore systems in general purpose computing. This phenomenon is caused by the fact that, in accordance with Pollack's Rule [182], single processor scaling no longer delivers the expected speed-up, while, at the same time, the power density (causing so-called *chip hot-spots*) is getting problematic.

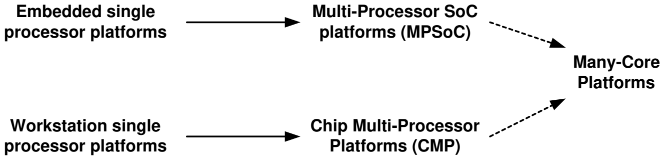

Embedded single processor platforms
Multi-Processor SoC platforms (MPSoC)
Many-Core Platforms
Workstation single processor platforms
Chip Multi-Processor Platforms (CMP)

Figure 8.5: *Vision of evolution to many-core systems.*

Today, these worlds are still apart. The embedded application designer (still) relies on *virtual platforms*, emulators, an MP-RTOS and a basic set of software tools (in essence EDA tool) to get the (heterogeneous) platform designed and programmed. The general-purpose application designer mainly relies on hardware and software abstraction layers, like e.g. caches and the operating system, to squeeze the most performance out of the (symmetric) multicore system. These designers are often supported by software development tools and compilers, like e.g. the Intel OpenMP enabled compiler, to split-up compute-intensive kernels in order to have them executed by a symmetric multiprocessor system more efficiently. In the many-core future, these two worlds are likely to join (or collide). I am not implying that we will end up with a single platform, mapping or run-time management solution. But it

[3] As the ITRS roadmap predicts that die size will remain constant at about 220mm^2

does mean we would end up with heterogeneous platforms with a (more) unified approach towards programming tools and run-time management technology.

The many-core era indeed shows a trend towards heterogeneity. We already see Intel [82], AMD [141] and ARM combine general purpose processing elements with graphics processing elements and accelerators on a single die. In addition, designers are experimenting with combining general purpose processing elements with fine-grain reconfigurable logic. On the other hand, we see today's embedded designers struggling with getting their multicore platforms programmed. Hence, the many-core era will also require a new set of programming tools that allow a designer to program such platform in an efficient and fast way. These programming tools and compilers might well be an evolution of the current CMP tools seasoned with today's embedded programming expertise.

With respect to run-time management, the many-core era not only provides new opportunities, it also introduces new research challenges. Obviously, there will be a need to support the heterogeneity and there will be a need to align with programming tools and compilers [193]. Designing a run-time manager will require revisiting (and maybe extending) the implementation space detailed in Chapter 2: what services will be provided by hardware and what services by software, what is the ideal amount of distribution and to what extend do we allow an application to manage its own resources?

Does this mean that the multicore run-time management concepts of this thesis will no longer be usable in the many-core era? Not necessarily. One could imagine a many-core being composed of a set of *multi-core clusters* each providing a specific service or executing a specific application. In this case, the concepts of this thesis can be applied to such a cluster. This component-based design would also (temporarily) relief the designer of having to handle a large amount of cores. From a run-time management viewpoint, it means that one is likely to end-up with a global run-time manager collaborating with multiple cluster-local run-time managers. In turn, this raises issues on how the different run-time managers would collaborate (i.e. more centralized or more distributed), especially when clusters need to collaborate in order to provide a more abstract platform or application service. Furthermore, it is likely that a platform contains clusters have been designed (hardware as well as software and cluster-local run-time manager) by a third party. This means that there will be a need to define interfaces that allows the global run-time manager to handle the third-party clusters and the services it provides. This also implies a shift towards more run-time management distribution.

8.2.5 Deep Deep Sub-Micron Run-Time Management Needs

In the sub-45nm technology nodes, process variability and reliability issues will start to play an important role [61]. Variability will cause a difference in behavior for two *identical* processing elements in a single SoC platform as well as two *identical* SoC platforms. Furthermore, these differences will vary over time. Reliability issues result in run-time failure of hardware components and their associated services. Essentially this means that the behavior of the predictable platform hardware services, on which the run-time manager relies, are themselves becoming unpredictable.

As these phenomena depend on run-time conditions, one requires a sort of run-time manager to handle these situations. IMEC's *Technology Aware Design* (TAD) research program focuses on introducing *Knobs and Monitors* inside various critical components in order to (1) detect when a component goes out of its designed operating conditions and (2) apply corrective measures if possible. This effectively means that costly worst case design is not necessary and that run-time phenomena such as temperature drift and aging effects can be managed. Figure 8.6 illustrates this setup. In this case, monitors include e.g. delay monitors, temperature sensors and signal level monitors, while the knobs[4] include e.g. setting the voltage or changing the speed/power ratio in line drivers. Just like for the NoC communication management (Chapter 5), there will (probably) be local and global monitoring and decision taking. In all this, one has to revisit the role of the different run-time management components and their design space (discussed in Chapter 2). Minor corrective services might be implemented as (reliable) platform components, while corrective services others will require action of the processor-local run-time library. There might be a need to also involve the resource manager, e.g. when a certain processing element or a NoC link fails. In this case, issues like task migration of communication rerouting can be performed. One thing seems certain, the role of the run-time manager will be increasingly important for handling platform variability and reliability issues.

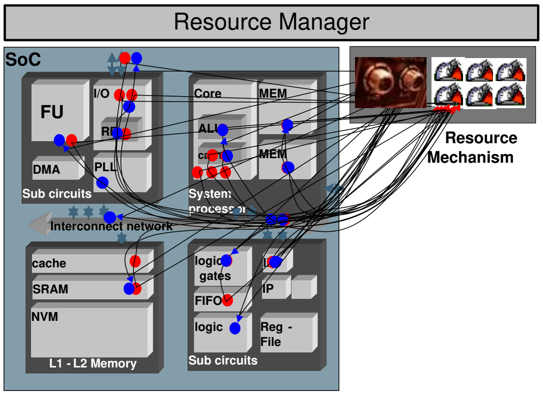


Figure 8.6: *SoC platform equipped with* Knobs and Monitors *[61]. Will the future role of the run-time manager include controlling the variability and reliability issues in such deep deep sub-micron platform?*

Intel also considers reliability issues and fault tolerance schemes for its future many-core platforms [18]. Also in their schemes, the run-time manager should be able to cope with tiles that are *out-of-spec*, or that are under-performing, and with rerouting communication in case of a faulty NoC communication link.

[4] A knob is a run-time controlled actuator.

8.3 Concluding Remarks

MPSoC platforms play an increasingly important role in satisfying the user hunger for extra application functionality and services in their embedded devices without an increase in cost. The run-time manager represents the *glue* between these applications and the MPSoC platform. It the plays an increasingly important role in enabling the flexibility, scalability, predictability and programmability of the MPSoC platforms. In addition, it is responsible for distributing the compute performance among the applications. Creating an MPSoC run-time manager that is both tuned to the needs of the application and its designer and adapted to the services provided by the hardware platform is not trivial. Indeed, most contemporary (industrial) MPSoC run-time managers are mainly focused on improving programmability by providing a run-time library. A more complete MPSoC run-time manager also considers application quality management and resource management. This thesis considered all three main run-time management components: the quality manager, the resource manager and the run-time library (RTLib). For each of these components, we have conceived several decision making algorithms and control mechanisms. In addition, we have created a proof-of-concept platform that not only integrates part of these algorithms and mechanism, but also illustrates their potential use case.

As long as MPSoC platforms and application continue to evolve, the run-time manager functionality and implementation will have to be adapted. This includes updating the relationship and the division of work between design-time tools and run-time management components. This also entails investigating the implementation and the functionality of the run-time managers for many-core platforms. Finally, as technology continues to scale, the platform services themselves will be subject to predictability issues. reliability the run-time manager could also take up the responsibility of monitoring and managing the platform hardware.

APPENDIX A

PSFF: Pareto Surfaces For Free

Run-time management research is often focused on defining new or optimizing existing resource management techniques. Validating these novel or improved algorithms requires a flexible and realistic application and/or platform testbench. Obviously, a run-time management researcher could develop and maintain a set of relevant, real-life testbench applications. However, maintaining and modifying such a testbench not only requires an enormous amount of effort, it also prohibits fellow researchers to reproduce the results as they do not possess this specific testbench.

In order to overcome the reproducibility problem, some researchers publish their synthetic input data on the web so others can reproduce and compare results. For example, the publicly available MMKP input data [1] allowed us to validate our MMKP algorithm and compare the results in a fair way to other algorithms (Section 7.2.2). However, publishing the synthetic input data does not address the testbench flexibility issue. This means that a testbench should be flexible enough to enable fast *what if* analysis. Meaning that one would like to change the boundary conditions of the input data to assess the limits of the run-time management algorithms. In order to overcome the flexibility problem, input data generators, like e.g. *Task Graphs For Free* (TGFF) [60] are developed. These input data generators not only allow other researcher to reproduce the input data based on a given set of generator input data, they also allow the researcher to quickly adapt the testbench during the research phase.

In order to have a the realistic, reproducible and flexible testbench with respect to adaptive multimedia applications where a single application quality level can have multiple implementations (like the example of Figure 7.1(b)), we have developed an input data generation tool denoted *Pareto Surfaces For Free* (PSFF).

The main goal of the PSFF tool is to generate multiple realistic application operating points given application kernel information and some overall designer knowledge about a specific application or the application domain. We used the QSDPCM video encoding application (Section 7.1.2) to substantiate the tool results.

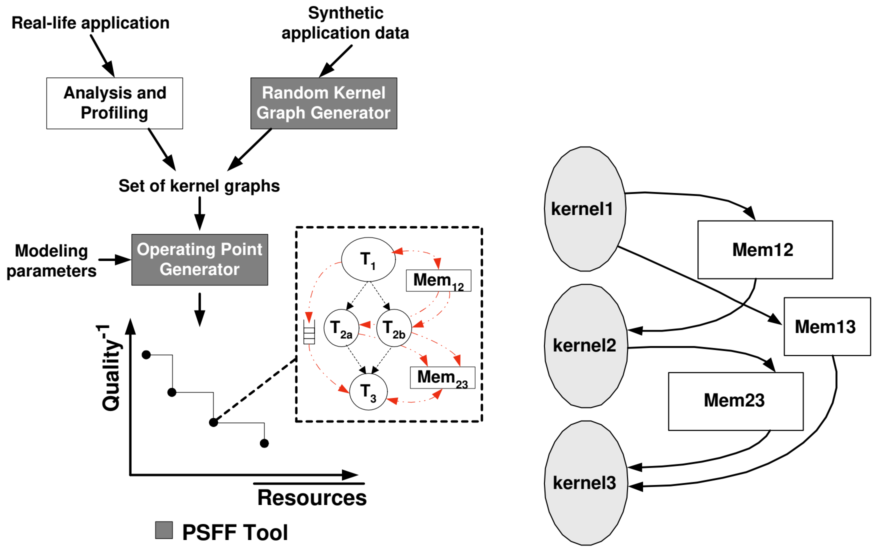
The rest of this appendix is organized as follows. First, we describe the operating point generation algorithm and motivate the tool knobs and options. This algorithm sits at the heart of the PSFF tool. The main input for the operating point selection algorithm is a kernel graph. This kernel graph is either synthetically generated by the PSFF tool or based on real-life application information. Then, we show how the tool estimations compare to measured application results. Finally, we provide more details on the PSFF-generated experimental setup of Chapter 7.

A.1 Operating Point Generation

The goal of the PSFF tool (Figure A.1) is to derive a set of realistic operating points for a real or synthetic adaptive application given a set of minimal input parameters. The tool considers multimedia applications that are composed of a set of inner loops, further denoted as *kernels*, glued together by the application control code, further denoted as the *skeleton*. These kernels consume and produce data according to their relative dependency. Brockmeyer et al. [32] discusses the concepts of application kernels and skeletons more in-depth. Information concerning these kernels forms the basis input for the tool and, for real-life applications, should be easily obtainable by profiling the sequential application on a target processor architecture.

This basic input information includes a *kernel graph* (Figure A.1(b)) and, for every kernel, its execution time (expressed in e.g. number of cycles, target processor load, etc.), its memory needs, i.e. the size of the required memory and, finally, the amount of accesses and the bandwidth to every kernel memory block. For a real life application the number of execution cycles (or processor load) and the number of data accesses can easily be obtained by a sequential profiling run. The amount of data accesses can be obtained either by using an e.g. (modified) instruction set simulator or by by using application analysis and profiling tools like e.g. ATOMIUM [39] or the *Software Instrumentation Tool* (SIT) [137]. To determine the required shared memory between the kernels, the amount of accesses and the associated read/write bandwidth, one requires an estimate of the data reuse factor. Again, for a real-life application, the data reuse distance can be estimated by profiling the application and the memory accesses and the associated bandwidth can be derived for a given memory hierarchy [6,99].

Furthermore, the PSFF tool user will have to provide a set of modeling parameters (either global or specified per kernel) based on specific application implementation knowledge (in case of a real-life application) or based on application domain knowledge. These parameters include:



(a) (b)

Figure A.1: PSFF tool flow. The PSFF tool (a) requires a kernel graph (b) as input. This kernel graph can be based on synthetic application data provided by the designer or can be obtained by analyzing and/or profiling a real-life application.

- **Skeleton overhead *(sko)*.** This represents the overhead of the skeleton code and can be estimated by subtracting the total amount of cycles (or time) spent in kernel processing from the total sequential application execution time.

- **Functional-split overhead *(fso)*.** This represents the overhead introduced by the synchronization of the different kernels when executing in an asynchronous fashion. This overhead depends on the synchronization granularity and, in case of a real-life application, obviously requires some application knowledge to estimate.

- **Data-split overhead *(dso)*.** This represents the overhead introduced when duplicating a kernel in order to have multiple identical kernels. This overhead represents 3 components: the potentially uneven workload distribution, execution dependencies and the fact that processing one part of the data might require accessing a part belonging to another kernel. The data-split overhead can be constant or can grow with the size of the data-split. Providing an estimate for the data split overhead for a real-life application requires some amount of application knowledge.

- **Trade-offs.** A description of the trade-offs that have to be explored. The most common trade-offs are the processing-time versus bandwidth trade-off and the memory size versus bandwidth trade-off. Intuitively, it is easy to understand that limiting the bandwidth of an edge to a shared memory will increase the processing wait time, while having more local memory will decrease the amount of accesses to a higher layer memory and, potentially, also reduces the required bandwidth.

- **Constraints.** This includes the maximum number of tasks and whether only consecutive kernels can belong to the same task (i.e. in order) or not.

Consider, for every quality $q \in Q$, a kernel graph $KG_q(k, m, e)$ containing a set of application kernels k_i, memory blocks m_k and edges e_{ik}. Every kernel has an execution time k_i^{exec}, while every memory block has a size m_k^{size}. Every edge has a certain read and write bandwidth, denoted e_{ik}^{bw-rd} and e_{ik}^{bw-wr} respectively, and a certain amount of read and write accesses, denoted e_{ik}^{acc-rd} and e_{ik}^{acc-wr} respectively. Further, the tool receives a skeleton overhead parameter sko, a functional split overhead parameter fso, a data-split overhead dso and a maximum amount of tasks N.

Algorithm 16: Algorithm to generate a adaptive application Pareto surface

```
Input: $KG_q(k, m, e)$, $Application\ Parameters$, $SurfaceAxis$
Output: Application Pareto Surface
PSFF($KG_{quality_i}(K, E, M)$, $Application\ Parameters$, $SurfaceAxis$)
(1) foreach $q \in Q$
(2)   $NewTaskGraphs = PerformFunctionSplit(KG_q(K, E, M), N, fso)$
(3)   $AddToTaskGraphList(NewTaskGraphs, TaskGraphList)$
(4)   foreach $TaskGraph \in TaskGraphList$ with amount of tasks $< N$
(5)     $NewTaskGraphs = PerformDataSplit(TaskGraph)$
(6)     $AddToTaskGraphList(NewTaskGraphs, TaskGraphList)$
(7)   foreach $TaskGraph \in TaskGraphList$
(8)     $NewTaskGraphs = PerformTradeOffs(TaskGraph)$
(9)     $AddToTaskGraphList(NewTaskGraphs, TaskGraphList)$
(10) $OperatingPointSurface = CreateOperatingPointSurface(SurfaceAxis)$
(11) $PruneDominatedPoints(OperatingPointSurface)$
```

The rest of this section provides more details about the most important parts of Algorithm 16. First, Section A.1.1 details the $PerformFunctionSplit()$ function used on line 2. Section A.1.2 describes the $PerformDataSplit()$ function used on line 5. Finally, Section A.1.3 details the $PerformTradeOffs()$ function used on line 8

A.1.1 Functional Split

This part of the algorithm combines the application kernels k_i, memory blocks m_k and edges e_{ik} into a task graph TG containing one or more functional tasks T_n and their associated memory blocks M_j and task edges E_l. Figure A.2 illustrates the result of this exercise for the kernel task graph shown in Figure A.1(b).

At the extremes, Figure A.2(a) illustrates the case where all kernels are combined in a single task, while Figure A.2(e) illustrates the situation where there is only one kernel per task. In between, Figure A.2(b), Figure A.2(c) and Figure A.2(d) illustrate the combinations for a functional split with two tasks. The functional combination illustrated by Figure A.2(d) combines two non-consecutive kernels, k_1 and k_3, into a single task. This type of task graph can be avoided if the designer specifies to only

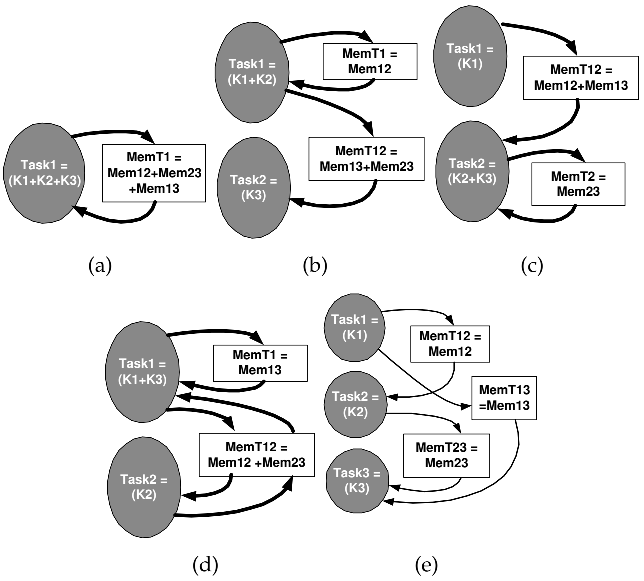


Figure A.2: *Creating a functional split application based on the kernel graph of Figure A.1(b). Of course, this includes a single task (a), a set of functional splits containing two tasks (b,c,d) and a functional split containing three tasks (e).*

combine consecutive kernels. As these figures illustrate, there is not more than one memory block for communication between two tasks.

calculation of the task execution time. In case of a single task (Figure A.2(a)), the functional split overhead factor fso is dropped. The size of a memory block is given by Equation A.2. Every task edge E_k is associated with a type (*read* or *write* from the perspective of the task), with an amount of memory accesses (Equation A.3) and with a communication bandwidth (Equation A.4). Indeed, the bandwidth of a task edge should be the maximum of the bandwidth required for the kernels this task contains if all contained kernels are to meet their execution time. The graph execution time is equal to the largest task execution time.

$$T_n^{exec} = \sum_i k_i^{exec} \times (1 + sko + fso) \tag{A.1}$$

$$M_j^{size} = \sum_k m_k^{size} \tag{A.2}$$

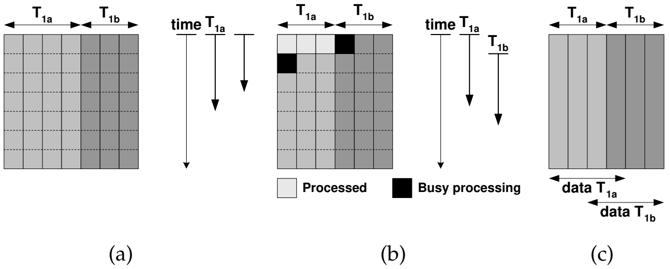
$$E_l^{acc-rd/wr} = \sum_{\forall k_i, k_j} e_{ij}^{acc-rd/wr} \tag{A.3}$$

$$E_l^{bw-rd/wr} = MAX_{\forall k_i, k_j} e_{ij}^{bw-rd/wr} \tag{A.4}$$

A.1.2 Data Split

The second part of the algorithm performs a data split on the task graphs generated in the previous step. Performing a data split involves duplicating the task graph, but splitting one or more tasks into two or more data-parallel tasks. Such data-split tasks are identical in functionality, but they will operate on a subset of the data that needs to be processed. Although one could exhaustively make a combination of all possible data-splits on all tasks of a specific task graph, the currently implemented data-split policy is focused on splitting the task with the largest execution time in order to get a task graph which is more balanced with respect to task execution time.

Theoretically, this would imply that the execution time and the amount of data accesses of the original task is evenly distributed over the data-split tasks, while the required bandwidth for every data-split task remains the same as for the original task (i.e. half the amount of accesses in half the amount of time results in the same bandwidth requirement). In practice, however, there are three potential issues. First, it might be that the data that needs to be processed cannot be evenly distributed among the data-split tasks (Figure A.3(a)). In turn, this means that the execution time and the amount of data accesses will not be distributed evenly. Secondly, there might be execution dependencies between two data-split tasks causing some amount of serialization (Figure A.3(b)). This means that the overall execution time required for this task will be larger than the execution time of the original task divided by the amount of data-split tasks. Finally, there might be an overlap in the required data between two data-split tasks (Figure A.3(c)).



(a) (b) (c)

Figure A.3: Potential causes of uneven task data-split. (a) An uneven data workload distribution. (b) Data processing dependencies. (c) Additional input needs for processing assigned data.

The properties of the n data split tasks $(T_{i1}, T_{i2}, ..., T_{in})$ originating from task T_i and the edges $(E_{i1}, E_{i2}, ..., E_{in})$ originating from edge E_i are calculated as follows. In these formulas, α describes whether the data-split overhead is constant or growing with the size of the split. This means α is either equal to n if the data-split overhead is not proportional to the amount of data-split tasks or equal to 1 in case of proportional data-split overhead. Equation A.5 details the task execution time, while Equation A.6 and Equation A.7 detail the edge accesses and bandwidth.

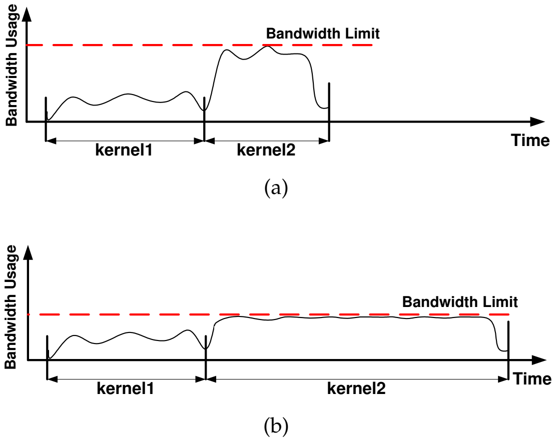
$$T_{ij}^{exec} = \frac{T_i^{exec}}{n} \times (1 + \frac{n \times dso}{\alpha}) \tag{A.5}$$

$$E_{ij}^{acc-rd/wr} = \frac{E_i^{acc-rd/wr}}{n} \times (1 + \frac{n \times dso}{\alpha}) \tag{A.6}$$

$$E_{ij}^{bw-rd/wr} = E_i^{bw-rd/wr} \times (1 + \frac{n \times dso}{\alpha}) \tag{A.7}$$

A.1.3 Applying Trade-offs

Limiting the bandwidth of a task edge may increase the task execution time as Figure A.4 illustrates. The task execution time is equal to the sum of its kernel execution times, while the bandwidth of its communication edge is equal to the maximum of the included kernel bandwidth figures (Figure A.4(a)). Limiting the edge bandwidth leaves kernel 1 unaffected but results in a prolonged kernel 2 execution time and, as a consequence, a prolonged task execution time (Figure A.4(b)).



(a)

(b)

Figure A.4: Bandwidth reduction results in a prolonged task execution time. (a) A task contains two kernels. The task bandwidth is set to the maximum of the required kernel bandwidth. (b) Reducing the maximum bandwidth does not affect kernel 1, but does prolong the execution of kernel 2. In turn this results in a prolonged task execution time.

For the selected task graph, the algorithm limits every edge to a specified value and assesses the impact on the execution time of the task kernels. In case of Figure A.4, the policy is to proportionally increase the kernel execution time. The task execution time is recalculated based on the updated kernel execution times. Another trade-off, currently not supported by the PSFF tool, is the memory-bandwidth trade-off. The idea is to trade either L1 memory size for L1-L2 communication bandwidth or L2 memory size for L2-L3 communication bandwidth. Indeed, by using more L1 memory, one can reduce the amount of accesses (also denoted as L1 *misses*) and, as a (potential) consequence, the required bandwidth. This trade-off requires a locality metric for every kernel to determine its number of *access misses* as a function of the L1 memory size.

A.2 Generating Random Surfaces

By including a random kernel graph generator (Figure A.1(a)), the PSFF tool also allows a researcher to quickly create a large set of synthetic operating point surfaces. This random kernel graph generator is detailed by Algorithm 17. It essentially consists of two parts. In the first part, a random kernel graph structure is generated based on a set of designer-provided kernel graph parameters. In the second part, this kernel graph structure is annotated with *resource usage* values according to one or more quality requirements.

Creating the kernel graph structure (line 1-line 8) requires a minimum and maximum value for the number of kernels, the number of memory blocks and the number of edges per kernel. A kernel graph structure is created with N_k kernels, N_m memory blocks and N_e^k edges for a given kernel k. Every edge is also linked to a memory and is either READ or WRITE (from the kernel perspective).

Once the kernel graph structure has been created, resource usage figures, corresponding to the required quality, have to be added (line 8-line 15). This means that for every quality q, one has to determine, for every kernel k, its execution time k^{exec}, for every memory m its size m^{size} and for every edge e_{ij} its bandwidth $e_{ij}^{bw-rd/wr}$ between kernel k_i and memory m_j. The amount of accesses of every edge is derived from it bandwidth and the kernel execution time of its associated kernel. The resource usage figures are based an average (μ) and a spread (δ) resource usage value for every quality level q provided by the designer.

This way, the $GenerateKernelGraph(\cdots)$ algorithm provides, for every quality $q \in Q$, a kernel graph $KG_q(K, E, M)$ that can be used as input for the operating point surface generation algorithm (Algorithm 16).

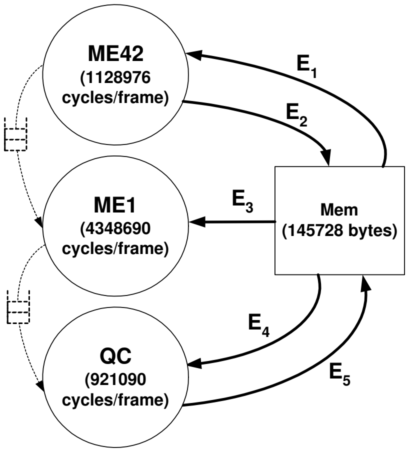
Algorithm 17: *Algorithm to generate a random kernel graph set*

```
Input: $\mu_k, \delta_k, \mu_m, \delta_m, \mu_e, \delta_e$
Output: $KG_q(K, E, M)$
GenerateKernelGraph($\mu_k, \delta_k, \mu_m, \delta_m, \mu_e, \delta_e$)
(1)  $N_k = random(min_k, max_k)$
(2)  $N_m = random(min_m, max_m)$
(3)  foreach $kernel\ k$
(4)    $N_e^k = random(min_e, max_e)$
(5)    foreach $edge\ e_k$
(6)      $e_{k,mem} = random(N_m)$
(7)      $e_{k,type} = random(READ, WRITE)$
(8)  foreach $q \in Q$
(9)    foreach $m \in M$
(10)     $m^{size} = random(\mu_m^{size}, \delta_m^{size})$
(11)   foreach $k \in K$
(12)     $k^{exec} = random(\mu_k^{exec}, \delta_k^{exec})$
(13)   foreach $e_{ij} \in E\ between\ k_i\ and\ m_j$
(14)     $e_{ij}^{bw-rd/wr} = random(\mu_e^{bw}, \delta_e^{bw})$
(15)     $e_{ij}^{acc-rd/wr} = e_{ij}^{bw-rd/wr} \times k_i^{exec}$
```

A.3 QSDPCM Operating Points

In order to assess the tool on its realism, we reproduce the outcome of the QSDPCM mapping experiments of Section 7.1.2 by using the PSFF tool. Figure A.5 details the

QSDPCM QCIF input kernel graph and its properties. These properties have been derived from the sequential QSDPCM analysis performed by Brockmeyer et al. [32]. The kernel graph consists of three kernels that mainly communicate through a single shared memory block. A few scalars (i.e. the motion vectors) are communicated directly between the kernels using FIFO communication (dotted lines). Although the PSFF tool is capable of handling this communication (just another four kernel edges with two memory blocks), this data is currently not considered in this exercise as it was not considered in the QSDPCM analysis.



Edge#	Type	#acc	BW (MB/s)
1	READ	50688	26
2	WRITE	6336	3
3	READ	68112	19
4	READ	68112	35
5	WRITE	25344	13

Figure A.5: *QSDPCM kernel graph for QCIF resolution.*

Table A.1 details the used overhead parameters, also derived from the QSDPCM analysis by Brockmeyer et al. [32]. The skeleton overhead (sko) was calculated by subtracting the sum of the kernel execution times from the total sequential execution time. The functional split overhead (fso) represents an estimate based on the QSDPCM execution time analysis experiments (also briefly detailed in Figure 7.6). The data-split overhead (dso) has to consider two issues. First, there is an overlap in the required data between two data parallel tasks, illustrated by Figure A.3(c). For QCIF this has been evaluated to be 22% [32]. Secondly, as there are 11 columns, the data-split cannot be performed equally. This concept is illustrated by Figure A.3(a). Indeed, a split into two tasks will produce a task that processes 6 columns and a task that processes 5 columns (i.e. a difference of about 20%). When going to more data-parallel tasks, this relative overhead may increase[1]. Hence, the α parameter is set to 1.

By using this kernel graph and the parameters as input for the PSFF tool and by selecting those points that correspond to the mapping experiments of Section 7.1.2, one can compare the measured operating point set versus the PSFF generated set. This comparison is illustrated by Figure A.6. Figure A.6(a) and Figure A.6(b) compare the results when the task edge bandwidth is limited to respectively 100MB/s and 20MB/s.

[1] Here we neglect the fact that, depending on the kernel, processing image boundaries can require more or less time than processing an inner column.

Table A.1: *QSDPCM PSFF overhead parameters.*

PSFF Parameter	Value
sko	2%
fso	20%
dso	22%
α	1

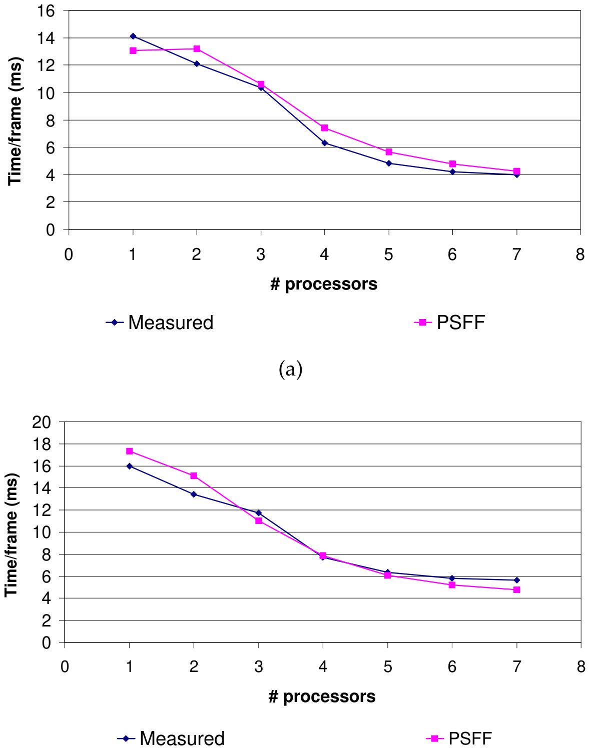
Overall, the generated results are quite close to the measured values. The maximum difference between the generated values and the measured values is 17%, while the average difference in about 9%. This is satisfactory from a perspective of performing research on selecting operating points.

In case of 100MB/s, some results require a closer inspection. It is odd that the measured value is higher than the generated value as the generated value is merely the sum of the kernels. A brief look at the source code used for this measurement revealed that the sequential code was obtained by concatenating three tasks, each containing a single kernel, into one large task. This means that, compared to the sequential code, there is additional overhead that the PSFF tool cannot take into account. With respect to the implementation using two tasks, the synergies between ME1 and QC were maximally exploited in order to reduce the load imbalance. Indeed, without this additional effort, the 2-processor implementation would be slower than its single processor counterpart due to the additional functional split overhead. As PSFF does not model this optimization, the difference between measured and PSFF results is significantly larger.

When comparing both graphs, one notices a difference between both graphs. In the 20MB/s graph, the PSFF estimate is relatively lower with respect to the measured values than for the 100MB/s graph. This is, most probably, caused by the fact that a bandwidth limitation has a high impact on the execution of pre-amble and post-amble of a task. During the pre-amble and post-amble, a task requires, for a short amount of time, more bandwidth to retrieve initial data of store final results. This occurs both for cache-based implementations and for scratchpad-based implementations. This phenomena is discussed by Marescaux [132]. In order for PSFF to take this into account, the model should be extended to also consider bandwidth needs and kernel execution time during preamble and postamble.

A.4 Random Operating Point Surfaces

The PSFF tool has been used to generate a set of random operating point surfaces to study the collaboration between the quality manager and the resource manager. These sets have been used in the experiments of Section 7.3 and Section 7.4. We created 100 random sets that each contain between 10 and 40 Pareto operating points (22 on average) distributed over a low, a medium and a high quality level. The input parameters for the PSFF random kernel graph generator tool are given by Ta-



(a)

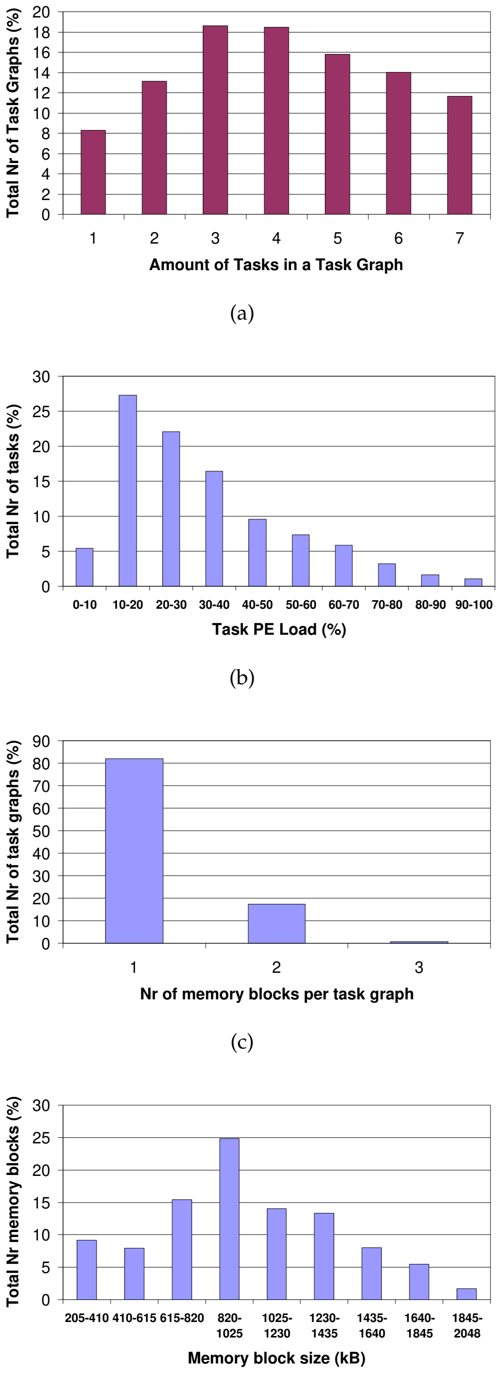
(b)

Figure A.6: *Comparison of measured and PSFF estimated QSDPCM operating points for different parallelizations. These graphs assume one task per processor. First (a) in case of an edge bandwidth limit of 100MB/s, then (b) in case of a 20MB/s limit.*

ble A.2, while the parameters for the PSFF operating point generator are detailed in Table A.3[2].

The characteristics of the generated operating points and their associated task graphs are detailed in Figure A.7. First, it shows that a task graph contains up to 7 tasks (Figure A.7(a)). and up to 3 memory blocks (Figure A.7(c)). The fact that most of the task graphs have only a single memory block is due to clustering of kernels into tasks (and hence kernel memory blocks into a single task graph memory block). The distribution of task load and memory block size of all operating points is given by respectively Figure A.7(b) and Figure A.7(d).

[2] These parameters have to be considered in the context of the resources available on the evaluation platform detailed in Figure 7.14.



(a)

(b)

(c)

(d)

Figure A.7: *PSFF random generator results for all generated operating points and their associated task graph. (a) Distribution of the number of tasks. (b) Distribution of task load. (c) Distribution of the number of memory blocks. (d) Distribution of memory block size.*

Table A.2: *PSFF Overhead Parameters.*

PSFF Parameter	**Value**
sko	2%
fso	20%
dso	22%
α	1

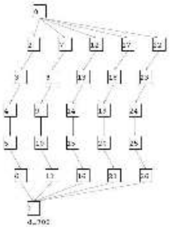
Table A.3: *PSFF Random Generator Parameters. This includes the kernel graph parameters and, for every quality, the kernel load parameters.*

Kernel Graph Parameter	**min**	**max**
# kernels	2	5
# kernel memory blocks	1	4
# edges per kernel	1	4

Load Parameters			
Quality	Resource	μ	δ
Low	PE load	10	5
	Memory size	400	100
	Edge BW	20	10
Medium	PE load	20	10
	Memory size	800	200
	Edge BW	40	20
High	PE load	40	20
	Memory size	1200	300
	Edge BW	80	40

A.5 Conclusions

The PSFF tool is primarily aimed at allowing a researcher to create, in a fast way, a realistic operating point sets with specific properties. This can be achieved starting from the profiled properties of a real application or from synthetic data. Although the model is, on some points, quite coarse for predicting the mapping result for realistic applications, it provides a high enough accuracy for performing run-time management research. Furthermore, we provide more detail on the experimental setup for assessing the collaboration between the quality manager and the resource manager.

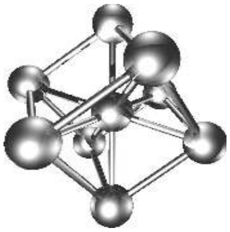


APPENDIX B

TGFF Input Values

This appendix describes the TGFF option file for generating task graphs and associated details used in the resource assignment experiments detailed in Section 3.6. In addition, this option file also enables TGFF to generate the softcore library used for the experiments detailed in Section 3.9.

```
#
# TGFF option file for resource management experiments
#

#########################################################
# average time per task (including communication)
task_trans_time 3
period_mul 1, 2

# number of task graphs
tg_cnt 1000
# minimum number of tasks within graph: <average> <multiplier>
task_cnt 3 1.2
# sets the maximum number of edges per task: <in> <out>
task_degree 3 3
# sets the number of possible task types
task_type_cnt 128
# sets the number of possible communication types
trans_type_cnt 128

####################################
# COMMUNICATION LOAD SPECIFICATION
#
# => total timeslots available on a link = 300
# => average used:
# - low = 25% --> 75
# - medium = 50% --> 150
# - high = 75% --> 225
# jitter = 0.5 by default
# rounding = 1
#
table_cnt 1
table_label COMM_LOAD
table_attrib
# type attribute <name> <average> <multiplier> <jitter> <rounding>
type_attrib slot_low 75 40 1.0 1.0, slot_med 150 75 1.0 1.0, slot_high 225 75 1.0 1.0
# write it to file ('trans_write' applies to transmission events)
trans_write

###################################################
# WHICH HARD PROCESSOR TYPES ARE AVAILABLE ON THE PLATFORM ??
#
# ISP => ARM (type 0)
table_cnt 1
table_label PE_INFO
table_attrib pe_type 0 0 0 0, soft_size 0 0 0 0
type_attrib low 25 15 1.0 1.0, med 50 15 1.0 1.0, high 75 15 1.0 1.0, size 0 0 0 0
pe_write

# ISP => DSP (type 1)
table_cnt 1
table_label PE_INFO
table_attrib pe_type 1 0 0 0, soft_size 0 0 0 0
type_attrib low 20 10 1.0 1.0, med 40 10 1.0 1.0, high 60 10 1.0 1.0, size 0 0 0 0
pe_write

# FPGA (type 2)
table_cnt 1
table_label PE_INFO
table_attrib pe_type 2 0 0 0, soft_size 0 0 0 0
type_attrib low 18 10 1.0 1.0, med 35 10 1.0 1.0, high 50 10 1.0 1.0, size 50 30 0.5 1.0
pe_write

# ACCEL (type 3)
table_cnt 1
table_label PE_INFO
table_attrib pe_type 3 0 0 0, soft_size 0 0 0 0
type_attrib low 15 10 1.0 1.0, med 30 10 1.0 1.0, high 45 10 1.0 1.0, size 0 0 0 0
pe_write

####################################################
#
# SOFTCORES (type 10)
# - softcores have a certain size (so one cannot put them on any tile)
# - the performance of a softcore is lower than that of a hardcore
#
# LARGE SOFTCORES
table_cnt 4
table_label PE_INFO
table_attrib pe_type 10 0 0 0, soft_size 70 30 0.5 1.0
type_attrib low 35 10 1.0 1.0, med 80 25 1.0 1.0, high 120 25 1.0 1.0, size 0 0 0 0
pe_write

# SMALL SOFTCORES
table_cnt 4
table_label PE_INFO
table_attrib pe_type 10 0 0 0, soft_size 40 20 0.5 1.0
type_attrib low 35 10 1.0 1.0, med 80 25 1.0 1.0, high 120 25 1.0 1.0, size 0 0 0 0
pe_write

####################################################
# write tasks graphs and figures
tg_write
eps_write
```



APPENDIX C

IMEC MPSoC Application Design and Mapping

By establishing a close relation between run-time management and application design and mapping, one can come to better run-time decision-making. Indeed, if the run-time manager can rely on design-time application analysis information, it can improve the overall user experience and make better use of the available platform resources.

Although it would be possible for a designer to perform all application exploration and analysis manually, having design-time application exploration, design and mapping tools greatly increase the speed and accuracy of this process. Hence, IMEC is spending effort to create such proof-of-concept tools. Just like the run-time manager, these tools fit into the global MPSoC application mapping picture.

This appendix provides a brief overview of the IMEC MPSoC design-time tools. In addition, it illustrates the relation between the design tools and the run-time manager. So the rest of this chapter is organized as follows. Section C.1 briefly details the application mapping flow with its associated tools and the role of the run-time manager. Section C.2 details the application design exploration investigated by the mapping tools. Finally, Section C.2.4 provides more details on the relation between the tools and the run-time manager.

C.1 Application Mapping Flow and Tools

In order to relief the embedded designer as much as possible, the IMEC MPSoC tools focus on assisting the designer in mapping sequential C code in an efficient, predictable and fast way onto a multiprocessor platform. In order to achieve this, these mapping tools should:

- perform parallelization, add synchronization and inter-task communication, and manage the memory hierarchy;
- explicitly control the (data) communication to avoid the hardware cache coherency problem/scaling bottleneck and avoid resource contention. This involves active management of inter-task synchronization and communication, and handling the memory hierarchy communication;
- make the code-transition to the parallel programming model and make sure the transition is functionally correct and optimized for a given set of platform parameters;
- be capable of providing feedback to the designer on a sequential level.

Creating such mapping tools obviously comes at a cost in both software and hardware. The software cost boils down to the fact that it is not possible to use plain ANSI C. Some language constructs are not well analyzable (e.g. pointer aliasing), while others make it hard for the mapping tool to efficiently perform, for example, a parallelization (e.g. global variables). Hence, sequential *Clean C* needs to be derived from sequential ANSI C in order to obtain good mapping results! The hardware cost boils down to the MPSoC platform needing to provide services that allow a mapping tool to reason about parallelization performance, inter-task communication, resource assignment, resource scheduling and application component performance. These services include are, for example, a bounded latency on a memory-to-memory data movement transaction. This, in turn, requires a platform service that allocates a certain amount of communication bandwidth. Another example is a service that enables a software-controlled memory hierarchy.

Figure C.1 shows the general MPSoC flow for mapping sequential ANSI C code onto a multiprocessor platform. This flow starts by cleaning the code, i.e. deriving Clean C code, by means of the MPSoC interactive cleaning tools in order to remove undesired language constructs. Consequently, the designer uses the MPSoC mapping tools to map the code onto the multiprocessor platform. These mapping tools require as input: Clean C, high-level platform properties and designer parallelization hints. Note that this sequential C code can contain calls to accelerator functionality.

The tools map the application component onto a parallel programming API provided by the RTLib library. This library is part of the run-time manager and is implemented on top of the platform hardware services (and hence should be provided to the embedded SW designer together with the MPSoC platform). The designer can, if desired, program the platform directly by using the RTLib API. Finally, the compiler is responsible for transforming the parallel source code into platform executables. The run-time manager is responsible for flexibly assigning platform resources to one or more application components. In essence, the embedded software

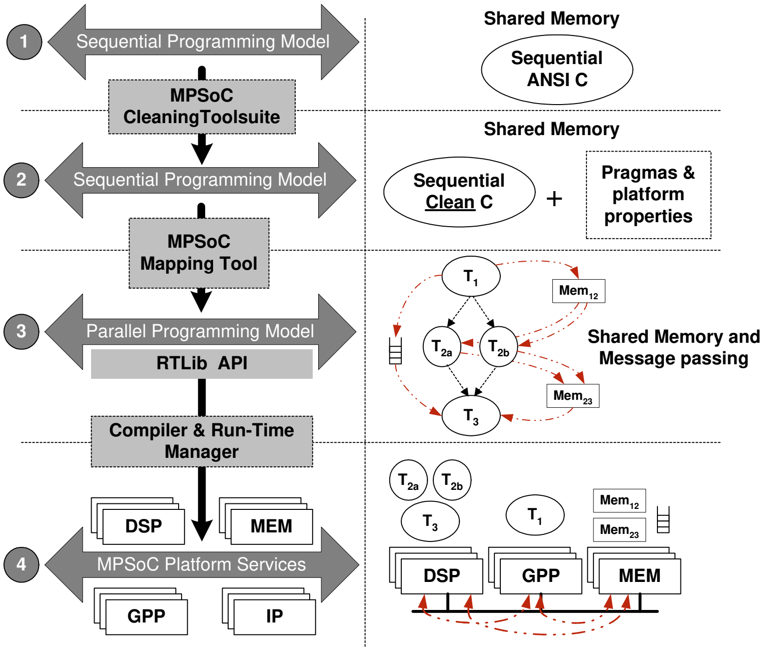


Figure C.1: *General MPSoC flow for mapping sequential ANSI C code onto a MPSoC platform.*

designer should only be concerned with step 1 and step 2 (Figure C.1), the rest, i.e. the platform services and RTLib, should be provided by the platform designer and are *under the hood* with respect to the embedded software designer.

The MPSoC mapping tools are responsible for parallelizing the sequential application component (given some designer pragmas). This includes automatically adding inter-task communication and synchronization. Furthermore, these tools explicitly manage and exploit the scratchpad memory hierarchy in a component-specific way.

C.2 Application Design Exploration

This section focuses on the application design and mapping exploration. This includes exploring the trade-off between memory size and bandwidth usage (Section C.2.1) and the parallelization exploration (Section C.2.2). IMEC is also developing a set of application mapping tools to help the designer perform this exploration (Section C.2.3).

C.2.1 Memory Size versus Bandwidth Exploration

The MPSoC mapping tools are able to explicitly manage data communication by inserting data transfer statements into the code. These statements, denoted as *block-transfers*, should be implemented by the RTLib and rely on a platform DMA service to perform these data transfers. By relying on a guaranteed bandwidth service, the tools enable the designer to perform a trade-off between memory size and communication bandwidth. Indeed, by changing the prefetch initiation time of a certain memory block, one can change the required bandwidth for transferring the data. Unfortunately, prefetching earlier means that a certain block of memory will have a longer life-time and, as a consequence, the overall required memory will rise.

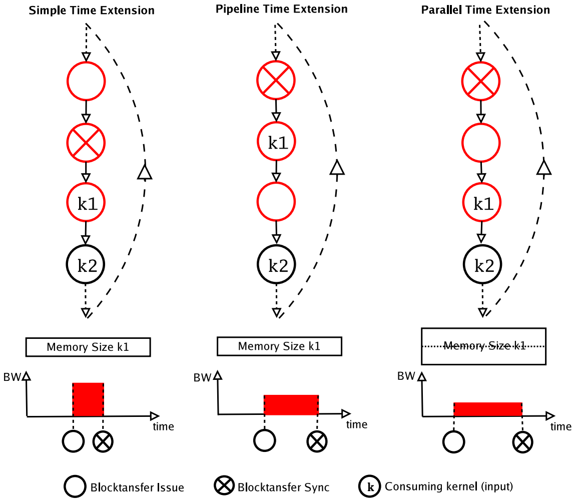


__Figure C.2:__ Memory-Bandwidth trade-off that needs to be considered when inserting explicit data management statements.

With respect to explicit data communication, the MPSoC mapping tool adds two types of RTLib-supported statements into the code. A *blocktransfer issue* for starting a DMA data transfer and a *blocktransfer sync* for waiting on the finalization of the blocktransfer. There are three ways the MPSoC mapping tool can insert these statements into the source code (Figure C.2).

The first way, denoted as *simple time extensions*, is composed of a blocktransfer issue immediately followed by a blocktransfer sync and the data consuming kernel (k1). This blocktransfer requires a significant amount of bandwidth to minimize the blocktransfer waiting time, since the communication does not happen in parallel with the computation and, hence, the computation waits until the data is transferred.

In the second way, denoted as *pipeline time extensions*, the blocktransfer happens in parallel with the computation of other kernels (k2). Meaning that there is a blocktransfer sync just before the consuming kernel. After all data has been consumed, a new blocktransfer is started. The blocktransfer can happen in parallel with all the other computation in the loop (not shown in the figure). Hence, the amount of required bandwidth with respect to the first case can be lower.

In the third way, denoted as *parallel time extensions*, the data for the next loop iteration is already prefetched while the previously fetched data is still unprocessed. This can be seen by the fact that there is a blocktransfer sync to ensure that the data for the current loop iteration (i.e. for the consuming kernel) is ready. But even before the data is consumed, a new blocktransfer is already started. Only then, the ready data is consumed. This solution obviously requires more memory since new data is being fetched, while the ready data is still being processed. Due to the fact that the data can take more time to arrive, the required bandwidth is lower.

Brockmeyer et al. provide more detail on the concept of explicit data transfers [31] and performs these trade-offs (and more) with respect to the QSDPCM video encoding application [32].

C.2.2 Parallelization Exploration

The general idea of parallelization with the IMEC MPSoC mapping tools is that the designer identifies parts of the sequential code that are heavily executed and should be executed by multiple threads in parallel to improve the performance of the application component. These pieces of code that will be parallelized are denoted as *parsections*.

For each parsection, the designer specifies the number of threads that execute it. The designer can divide the work over threads in terms of functionality (i.e. functional parallelism), in terms of loop iterations (i.e. data parallelism), or a flexible combination of both depending on what is the most appropriate for a given parsection.

These parallelization directives are to be written in a file provided to the mapping tool. The main reason for using directives in a separate file instead of pragmas inserted in the input code, is that it simplifies exploration (and retainment) of multiple parallelization specifications for the same sequential code.

Given the input code and the parallelization directives, the tool will generate a parallel version of the code and insert FIFO and synchronization statements where needed. These statements correspond to the target RTLib API.

Hence, the designer does not need to worry about dependencies between threads, etc. This is all taken care of automatically by the tool. However, as designers like to be in control, the MPSoC mapping tool provides the *optional* mechanism of shared variables. By specifying a variable as being shared, the designer *explicitly* tells the tool that it does not have to add synchronization and communication mechanisms for that variable, because he will take care of that himself. This can be done in several ways. One way is to add dummy dependencies that will be converted into scalar FIFOs. Another way is to specify a *loop sync*, that synchronizes certain iterations of a loop with other iterations of loops in other threads. Sometimes, there is even no need to add additional synchronization mechanisms, if it is already guaranteed by other existing synchronization mechanisms (e.g. FIFOs).

C.2.3 IMEC MPSoC Design Tools

IMEC is developing a set of application exploration and mapping tools. These tool not only help the designer in managing scratchpad memory or in parallelizing sequential code, they also allow the designer to quickly explore and evaluate different mapping options with respect to the available resources. Figure C.3 shows the functionality of these tools.

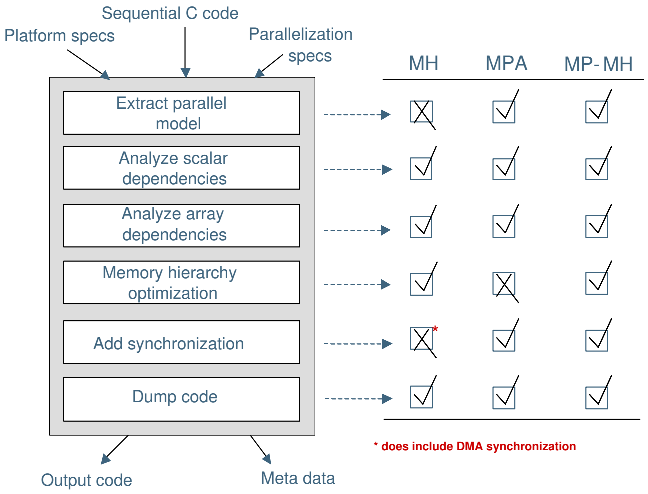


Figure C.3: IMEC MPSoC design-time mapping and exploration tools.

Given the platform specification information, the *Memory Hierarchy* (MH) tool is responsible for inserting *blocktransfer issue* and *blocktransfer sync* statements into the sequential C input code. By analyzing the scalar and array dependencies the tool is able to determine what data copies need to be made and how they should be scheduled in order to have the data prefetching finalized before the consuming kernel is started. The output of the tool is sequential code with scratchpad memory management statements and its associated design-time mapping information. The *Multiprocessor Parallelization Assist* (MPA) tool is responsible for parallelizing the sequential code according to the parallelization specification provided by the designer. This tool first extracts a parallel model of the application. Consequently, it analyzes scalar and array dependencies, adds synchronization statements and, finally, dumps the parallel code with its associated analysis information. The final goal is to develop a single tool, denoted *Multiprocessor Parallelization and Memory Hierarchy* (MPMH) tool that combines both the MH and the MPA tool functionality.

C.2.4 Design-Time Tools and Run-Time Management

There are two main relations between relation between the design-tools of Section C.2.3 and the run-time manager. First, the RTLib defines the API functions that are inserted by the design-tools in the output code. This includes APIs for task management (e.g. spawn(), join()), inter-task communication and synchronization (e.g. fifo_put(), fifo_get(), loopsync()), and memory hierarchy management (e.g. blocktransfer_issue() and blocktransfer_sync()). Secondly, by changing the platform specification file and/or the parallelization specification file, the designer can provide the run-time manager with a set of application operating points. This allows the run-time manager to choose an operating point, i.e. a certain parallelization and memory size versus bandwidth trade-offs, given the run-time application user requirements and the available platforms resources. More information regarding the collaboration between design-time tools and the run-time manager can be found in Chapter 2. More information on the usage of application operating points by the run-time manager can be found in Chapter 7.

C.3 Concluding Remarks

This appendix details the IMEC MPSoC mapping flow. It briefly introduces the IMEC MPSoC mapping tools that perform memory hierarchy management and parallelization for a sequential input application. In addition, it shows how the run-time manager and the design-time tools are linked through the run-time library API and the application mapping and analysis information.

Bibliography

[1] Multi-choice multi-dimension knapsack problems. http://www.laria.u-picardie.fr/hifi/OR-benchmark/MMKP/MMKP.html.

[2] Simit-ARM. [online] http://simit-arm.sourceforge.net.

[3] International Technology Roadmap for Semiconductors (ITRS), 1999 edition. http://public.itrs.net/, 1999.

[4] International Technology Roadmap for Semiconductors (ITRS), 2005 edition: Design Chapter. http://public.itrs.net/, 2005.

[5] International Technology Roadmap for Semiconductors (ITRS), 2005 edition: System Drivers Chapter. http://public.itrs.net/, 2005.

[6] Tom Vander Aa, Murali Jayapala, Francisco Barat, Henk Corporaal, Francky Catthoor, and Geert Deconinck. A high level memory energy estimator based on reuse distance. In: *Proc. of 3nd Workshop on Optimizations for DSP and Embedded Systems (ODES2005, together with CGO2005)*, San Jose, CA, USA, March 20-23 2005.

[7] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. Zeferino. Spin: A scalable, packet switched, on-chip micro-network. In: *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 70–73. IEEE Computer Society, Washington, DC, USA, 2003.

[8] A.S. Grimshaw A.J. Ferrari, S.J. Chapin. *Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification*. Technical report, University of Virginia, 1997.

[9] M. Mostofa Akbar, Eric G. Manning, Gholamali C. Shoja, and Shahadat Khan. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In: *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pp. 659–668. Springer-Verlag, London, UK, 2001.

[10] Md. Mostofa Akbar, M. Sohel Rahman, M. Kaykobad, E. G. Manning, and G. C. Shoja. Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Comput. Oper. Res.*, 33 (5): pp. 1259–1273, 2006.

[11] M. Allman, V. Paxson, and W. Stevens. RFC2581 - TCP Congestion Control, 1999.

[12] AndreiRadulescu and Kees Goossens. Communication services for networks on silicon. In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2002.

[13] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44 (2): pp. 427–440, 2005.

[14] R. D. Armstrong, D. S. Kung, P. Sinha, and A. A. Zoltners. A computational study of a multiple-choice knapsack algorithm. *ACM Transactions on Math. Softw.*, 9 (2): pp. 184–198, 1983.

[15] Alain Artieri. Nomadik: an MPSoC Solution for Advanced Multimedia. In: *Proceedings of the 5th International Forum on Application-Specific Multi-Processor SoC (MPSoC)*, Relais de Margaux, France, July 2005.

[16] G. Attardi, I. Filotti, and J. Marks. Techniques for Dynamic Software Migration. In: Information Industries Commission of the European Communities, Directorate-General Telecommunications and Innovation (Eds.), *Proceedings of the 5th Annual Esprit Conference (Esprit88)*, pp. 475–491. North-Holland, 1988.

[17] P. Avasare, V. Nollet, J-Y. Mignolet, D. Verkest, and H. Corporaal. Centralized end-to-end flow control in a best-effort Network-on-Chip. In: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pp. 17–20. ACM Press, New York, NY, USA, 2005.

[18] Mani Azimi, Naveen Cherukuri, D. Jayasimha, Akhilesh Kumar, Partha Kundu, Seungjoon Park, Ioannis Schoinas, and Aniruddha Vaidya. Integration Challenges and Trade-offs for Tera-scale Architectures. *Intel Technology Journal*, 11 (3): pp. 173–184, August 2007.

[19] Nilanjan Banerjee, Praveen Vellanki, and Karam S. Chatha. A power and performance model for network-on-chip architectures. In: *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pp. 1250–1256. IEEE Computer Society, Washington, DC, USA, 2004.

[20] D. Bansal and H. Balakrishna. Binomial Congestion Control Algorithms. In: *INFOCOM*, 2001.

[21] T.A. Bartic, J-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Highly scalable network on chip for reconfigurable systems. In: *Proceedings of the International System-on-Chip Conference (SoC)*, pp. 79–82, November 2003.

[22] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, 35 (1): pp. 70–78, 2002.

[23] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pp. 15–20, March 2006.

[24] Matt Bishop, Mark Valence, and Leonard F. Wisniewski. *Process Migration for Heterogeneous Distributed Systems*. Technical Report TR95-264, Dartmouth College, 1995.

[25] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of Network-on-Chip. *ACM Computing Surveys (CSUR)*, 38 (1): pp. 1–51, 2006.

[26] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23 (5): pp. 35–43, 1990.

[27] Shekhar Borkar. Thousand Core Chips - A Technology Perspective. In: *Proceedings of the 44th International Conference on Design automation (DAC)*, pp. 746–749, San Diego, CA, USA, June 2007.

[28] B. Bougard, S. Pollin, Gr. Lenoir, L. Van der Perre, Fr. Catthoor, and W. Dehaene. Transport level performance-energy trade-off in wireless networks and consequences on the system-level architecture and design paradigm. In: *Proceedings of the IEEE Symposium on Signal Processing Systems (SIPS)*, pp. 77–82, October 2004.

[29] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[30] R.J. Bril, C. Hentschel, E.F.M. Steffens, M. Gabrani, G. van Loo, and J.H.A Gelissen. Multimedia QoS in consumer terminals. In: *IEEE Workshop on Signal Processing Systems*, pp. 332–343, Antwerp, Belgium, September 2001.

[31] Erik Brockmeyer, Miguel Miranda, Henk Corporaal, and Francky Catthoor. Layer Assignment Techniques for Low Energy in Multi-layered Memory Organisations. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pp. 1070–1075, 2003.

[32] Erik Brockmeyer and Bart Vanhoof. *Updated Design Flow Proposal for 3MF Platform Definition and Mapping*. Technical Report M4.3MF.3MF5, IMEC, July 2005.

[33] Hajo Broersma, Daniël Paulusma, Gerard J. M. Smit, Frank Vlaardingerbroek, and Gerhard J. Woeginger. The Computational Complexity of the Minimum Weight Processor Assignment Problem. In: *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG2004)*, pp. 189–200, June 2004.

[34] Barry P. Brownstein. Pareto Optimality, External Benefits, and Public Goods: A Subjectivist Approach. *Journal of Libertarian Studies*, 4 (1): pp. 93–106, 1980.

[35] Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A Dynamic Reconfiguration Run-Time System. In: Kenneth L. Pocek and Jeffrey Arnold (Eds.), *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 66–75. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[36] R. Butenuth. The COSY-Kernel as an Example for Efficient Kernel Call Mechanisms on Transputers. In: *Proceedings of the 6th Transputer/occam International Conference*, 1994.

[37] R. Butenuth, W. Burke, C. De Rose, S. Gilles, and R. Weber. Experiences in building Cosy - an Operating System for Highly Parallel Computers. In: *Proceedings of the Conference Parallel Computing: Fundamentals, Applications and New Directions (ParCo)*, pp. 469–476, 1997.

[38] John Carbone. A SMP RTOS for the ARM MPCore Multiprocessor. *Design Strategies and Methodologies*, 4 (3): pp. 64–67, 2005.

[39] Francky Catthoor, Eddy de Greef, and Sven Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[40] Pramod Chandraiah and Rainer Domer. *Specifcation and Design of a MP3 Audio Decoder*. Technical Report CECS-05-04, Center for Embedded Computer Systems University of California, Irvine, May 2005.

[41] G. Chen, M. Kandemir, and U. Sezer. Configuration-Sensitive Process Scheduling for FPGA-Based Computing Platforms. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, p. 10486. IEEE Computer Society, Washington, DC, USA, 2004.

[42] Călin Ciordaş, Twan Basten, Andrei Rădulescu, Kees Goossens, and Jef van Meerbergen. An event-based monitoring service for networks on chip. *ACM Transactions on Design Automation of Electronic Systems*, 10 (4): pp. 702–723, October 2005. HLDVT'04 Special Issue on Validation of Large Systems.

[43] Theo A.C.M. Claasen. High speed: Not the only way to exploit the intrinsic computational power of silicon. In: *Proceedings of the International Conference on Solid-State Circuits (ISSCC)*, pp. 22–25, February 1999.

[44] Johan Cockx, Kristof Denolf, Bart Vanhoof, and Richard Stahl. SPRINT: a tool to generate concurrent transaction-level models from sequential code. *EURASIP Journal on Applied Signal Processing*, 2007 (1): pp. 213–213, 2007.

[45] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. pp. 46–93, 1997.

[46] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Survey*, 34 (2): pp. 171–210, 2002.

[47] AMDREL Consortium. *Survey of existing fine-grain reconfigurable hardware platforms*. Technical Report IST-2001-34379-WP4-D9-R, 2002.

[48] C. Couvreur, E. Brockmeyer, V. Nollet, Th. Marescaux, Fr. Catthoor, and H. Corporaal. Design-time application exploration for MP-SoC customized run-time management. In: *Proceedings of the International Symposium on System-on-Chip*, pp. 66–73, Tampere, Finland, November 2005.

[49] C. Couvreur, V. Nollet, F. Catthoor, and H. Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In: *Proceedings of the International Symposium on System-on-Chip (SOC)*, pp. 195–198, November 2006.

[50] C. Couvreur, V. Nollet, F. Catthoor, and H. Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. *ACM Transactions in Embedded Computing Systems*, (awaiting publication), 2008.

[51] C. Couvreur, V. Nollet, T. Marescaux, E. Brockmeyer, F. Catthoor, and H. Corporaal. Pareto-Based Application Specification for MPSoC Customized Run-Time Management. In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 78–84, July 2006.

[52] C. Couvreur, V. Nollet, T. Marescaux, E. Brockmeyer, F. Catthoor, and H. Corporaal. Design-time application mapping and platform exploration for MP-SoC customized run-time management. *IEE Computers & Digital Techniques*, 1 (2): pp. 120–128, March 2007.

[53] Chantal Couvreur and Vincent Nollet. *Pareto Point Selection Heuristics: Survey Report*. M4 Deliverable M4.MPSOC.DDT28, IMEC V.Z.W., Kapeldreef 75, 3001 Heverlee, Belgium, February 2006.

[54] Peter Cumming. *The TI OMAP Platform Approach to SoC*, Chapter 5, pp. 97–118. Kluwer Academic Publishers, 2003.

[55] Derek Curd. *Power Consumption in 65 nm FPGAs*. White Paper WP246-v1.2, Xilinx, February 2007.

[56] L. Dagum and R. Menon. OpenMP: an industry standard api for shared-memory programming. *Computational Science and Engineering, IEEE [see also Computing in Science & Engineering]*, 5 (1): pp. 46–55, Jan.-March 1998.

[57] William J. Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In: *Proceedings of the Design Automation Conference*, pp. 684–689, Las Vegas, NV, June 2001.

[58] F. Dammeyer and S. Voss. Dynamic tabu list management using the reverse elimination method. *Annals of Operations Research*, 41 (1-4): pp. 31–46, 1993.

[59] Kristof Denolf, Peter Vos, Jan Bormans, and Ivo Bolsens. Cost-Efficient C-Level Design of an MPEG-4 Video Decoder. In: *PATMOS*, pp. 233–242, 2000.

[60] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In: *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pp. 97–101. IEEE Computer Society, Washington, DC, USA, 1998.

[61] Bart Dierckx. Scaling below 90nm - designing with unreliable components, May 2007.

[62] O. Diessel and G. Wigley. *Opportunities for Operating Systems Research in Reconfigurable Computing*. Technical Report ACRC-99-018, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, August 1999.

[63] F. Douglis. Experience with Process Migration in Sprite. In: *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 59–72. USENIX Association, Berkeley, CA, 1989.

[64] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum. A Comparison of Two Distributed Systems: Amoeba and Sprite. *Computing Systems*, 4 (4): pp. 353–384, 1991.

[65] A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *IEEE Computing*, 40 (1): pp. 1–8, 1988.

[66] J. Duato, S. Yalamanchili, and L. Ni. *Interconnect Networks. An Engineering Approach.* IEEE Comp. Soc. Press, 1998.

[67] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In: *Proceedings of the 1988 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pp. 63–72. ACM Press, New York, NY, USA, 1988.

[68] Hendrik Eeckhaut, Mark Christiaens, Harald Devos, Philippe Faes, and Dirk Stroobandt. RESUME wavelet-based scalable video decoder. In: *DATE 2007 University Booth*, p. S42, Nice, France, 4 2007.

[69] Hendrik Eeckhaut, Mark Christiaens, Dirk Stroobandt, and Vincent Nollet. Optimizing the critical loop in the H.264/AVC CABAC decoder. In: *Proceedings of International Conference on Field Programmable Technology (FPT)*, pp. 113–118. IEEE, Bangkok, December 2006.

[70] Hendrik Eeckhaut, Harald Devos, Peter Lambert, David De Schrijver, Wim Van Lancker, Vincent Nollet, Prabhat Avasare, Tom Clerckx, Mark Christiaens, Dirk Stroobandt, Rik Van de Walle, and Peter Schelkens. Scalable, Wavelet-Based Video: From Server to Hardware-Accelerated Client. *IEEE Transactions on Multimedia*, 9 (7): pp. 1508–1519, November 2007.

[71] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pp. 251–266, Copper Mountain Resort, Colorado, December 1995.

[72] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The operating system kernel as a secure programmable machine. *Operating Systems Review*, 29 (1): pp. 78–82, January 1995.

[73] D. G. Feitelson and L. Rudolph. Wasted resources in gang scheduling. In: *Proceedings of the 5th Jerusalem Conference on Information Technology*, pp. 127–136, 1990.

[74] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. In: *Proceeding of the 17th Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2003.

[75] David Friedman. Should medicine be a commodity: An economist's perspective. *Rights to health care.*, pp. 259–305, 1991.

[76] João Garcia, Paulo Ferreira, and Paulo Guedes. Parallel operating systems. In: Jacek Bazewicz, Denis Trystram, and Denis Plateau (Eds.), *Handbook on Parallel and Distributed Processing*. Springer Verlag, 2000.

[77] L. Gauthier, Yoo Sungjoo, and A.A. Jerraya. Automatic generation and targeting of application-specific operating systems and embedded systems software. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 20, pp. 1293 – 1301, November 2001.

[78] L. Gauthier, S. Yoo, and A. Jerraya. Automatic generation and targeting of application specific operating systems and embedded systems software. In: *Proceedings of the conference on Design Automation and Test in Europe (DATE)*, pp. 679–685, 2001.

[79] Marc Geilen, Twan Basten, Bart Theelen, and Ralph Otten. *An Algebra of Pareto Points*. Technical Report ESR-2005-2, Eindhoven University of Technology, January 2005.

[80] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers Inc., The, 1991.

[81] Manuel G. Gericota, Gustavo R. Alves, Miguel L. Silva, and Jose M. Ferreira. Run-time management of logic resources on reconfigurable systems. In: *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, p. 10974. IEEE Computer Society, Washington, DC, USA, 2003.

[82] Antone Gonsalves. Intel to compete in high-end graphics market. *EETimes*, September 2007.

[83] John Goodacre and Andrew N. Sloss. Parallelism and the ARM Instruction Set Architecture. *Computer*, 38 (7): pp. 42–50, 2005.

[84] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22 (5): pp. 21–31, Sept-Oct 2005.

[85] V. Graefe and R. Bischoff. Past, present and future of intelligent robots. In: *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pp. 801–810, July 2003.

[86] S. Guccione, D. Levi, and P. Sundararajan. JBits: A Java based Interface for Reconfigurable Computing. In: *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD).*, 1999.

[87] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In: *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pp. 250–256. ACM Press, New York, NY, USA, 2000.

[88] Sinem Guven and Steven Feiner. Authoring 3D Hypermedia for Wearable Augmented and Virtual Reality. In: *ISWC '03: Proceedings of the 7th IEEE International Symposium on Wearable Computers*, p. 118. IEEE Computer Society, Washington, DC, USA, 2003.

[89] V. Chaudhary Hai Jiang. Compile/run-time support for thread migration. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 58–66, 2002.

[90] V. Chaudhary Hai Jiang. On improving thread migration: Safety and performance. In: *Proceedings of the 9th International Conference on High Performance Computing*, pp. 474–484, 2002.

[91] Andreas Hansson, Kees Goossens, and Andrei Radulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In: *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 75–80. ACM Press, New York, NY, USA, 2005.

[92] Scott Hauck. The future of reconfigurable systems. In: *Proceedings of the 5th Canadian Conference on Field Programmable Devices*, June 1998.

[93] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera Reconfigurable Functional Unit. In: *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pp. 87–96, 1997.

[94] Raimo Haukilahti. Energy Characterization of a RTOS Hardware Accelerator for SoCs. In: *Proceeding of the Swedish System-on-Chip Conference*, 2002.

[95] Seongmoo Heo, Kenneth Barr, and Krste Asanovic. Reducing power density through activity migration. In: *Proceedings of the 2003 international symposium on Low power electronics and design*, pp. 217–222. ACM Press, 2003.

[96] Jingcao Hu and Radu Marculescu. Energy-Aware Mapping for Tile-based NoC Architectures Under Performance Constraints. In: *Proceedings of the Asia & South Pacific Design Automation Conference (ASP-DAC)*, January 2003.

[97] Xilinx Inc. PicoBlaze 8-bit Microcontroller for Virtex Devices, XAPP213 (v1.2), April 2002.

[98] Spencer Isaacson and Doran Wilde. The Task-Resource Matrix: Control for a Distributed Reconfigurable Multi-Processor Hardware RTOS. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 130–136, Las Vegas, Nevada, USA, June 21-24 2004.

[99] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Data reuse analysis technique for software-controlled memory hierarchies. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, pp. 49–52. IEEE Computer Society, Washington, DC, USA, 2004.

[100] R. Marculescu J. Hu. Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures Under Real-Time Constraints. In: *Proceedings of DATE04 Conference*, pp. 234–239, 2004.

[101] Vernon Rego Janche Sang, Geoffrey W. Peters. Thread migration on heterogeneous systems via compile-time transformation. In: *Proceedings of the International Conference on Parallel and Distributed Systems*, pp. 634–639, 1994.

[102] Ahmed Jerraya, Hannu Tenhunen, and Wayne Wolf. Multiprocessors Systems-on-Chips. *IEEE Computer*, 38 (7): pp. 36–40, July 2005.

[103] Ahmed Amine Jerraya and Wayne Wolf. *MultiProcessor Systems-on-Chips*, Chapter The What, Why, and How of MPSoCs, pp. 1–20. Morgan Kaufmann, 2005.

[104] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An automated exploration framework for FPGA-based soft multiprocessor systems. In: *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pp. 273–278. ACM Press, New York, NY, USA, 2005.

[105] Klara Nahrstedt Jingwen Jin. *Classification and Comparison of QoS Specification Languages for Distributed Multimedia Applications*. Technical Report Technical Report UIUCDCS-R-2002-2302/UILU-ENG-2002-1745, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2002.

[106] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pp. 52–65, Saint-Malô, France, October 1997.

[107] Heiko Kalte and Mario Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In: Tero Rissa, Steven J. E. Wilton, and Philip Heng Wai Leong (Eds.), *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL)*, pp. 223–228. IEEE, August 2005.

[108] Eric Keller, Gordon J. Brebner, and Philip James-Roxby. Software decelerators. In: *13th International Conference on Field Programmable Logic and Applications*, pp. 385–395, 2003.

[109] Irwin Kennedy. Exploiting Redundancy to Speedup Reconfiguration of an FPGA. In: *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 262–271, 2003.

[110] Irwin Kennedy. Fast Reconfiguration Through Difference Compression. In: *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2003.

[111] Md. Shahadatullah Khan. *Quality adaptation in a multisession multimedia system: model, algorithms, and architecture*. PhD thesis, University of Victoria, Department of Electrical Engineering, 1998. Adviser-Kin F. Li and Adviser-Eric G. Manning.

[112] S. Khan, K.F. Li, and E. Manning. The utility model for adaptive multimedia systems. In: *Proceedings of the International Workshop on Multimedia Modeling*, pp. 111–126, 1997.

[113] Shahadat Khan, Kin F. Li, Eric G. Manning, and Md. Mostofa Akbar. Solving the knapsack problem for adaptive multimedia systems. *Studia Informatica Universalis*, 2 (1): pp. 157–178, 2002.

[114] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In: *SAC '94: Proceedings of the 1994 ACM symposium on Applied computing*, pp. 188–193. ACM Press, New York, NY, USA, 1994.

[115] Tommy Klevin. Get RealFast RTOS with Xilinx FPGAs. *Xcell Journal*, 45, February 2003.

[116] P. Koleser. A branch and bound algorithm for knapsack problem. *Management Science*, 13 (9): pp. 723–735, May 1967.

[117] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney. A configurable hardware scheduler for real-time systems. In: *Proceeding of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 95–101, 2003.

[118] Akash Kumar, Bart Mesman, Henk Corporaal, Jef van Meerbergen, and Ha Yajun. Global Analysis of Resource Arbitration for MPSoC. In: *Proceedings of the 9th Euromicro Conference in Digital System Design (DSD)*, pp. 71–78. IEEE Computer Society Press, August-September 2006.

[119] Shashi Kumar. On packet switched networks for on-chip communication. pp. 85–106, 2003.

[120] Michail G. Lagoudakis. *The 0-1 Knapsack Problem: An Introductory Survey*. Technical report, Center for Advanced Computer Studies, Fall 1996.

[121] Chen Lee, John Lehoczky, Dan Siewiorek, Ragunathan Rajkumar, and Jeff Hansen. A Scalable Solution to the Multi-Resource QoS Problem. In: *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, p. 315. IEEE Computer Society, Washington, DC, USA, December 1999.

[122] L. Levinson, R. Manner, M. Sessler, and H. Simmler. Preemptive Multitasking on FPGAs. In: *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 301. IEEE Computer Society, Washington, DC, USA, 2000.

[123] B. Lewis, I. Bolsens, R. Lauwereins, C. Wheddon, B. Gupta, and Y. Tanurhan. Reconfigurable SoC - What Will it Look Like? In: *DATE '02: Proceedings of the conference on Design, Automation and Test in Europe*, p. 660. IEEE Computer Society, Washington, DC, USA, 2002.

[124] lija Hadzic, Sanjay Udani, and Jonathan M. Smith. FPGA Viruses. In: *Proceeding of the 9th International Workshop Field-Programmable Logic and Applications (FPL)*, pp. 291–300, 1999.

[125] D. Lim and M. Peattie. *Xilinx Application Note XAPP290: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, 2002.

[126] Qingming Ma and P. Steenkiste. On path selection for traffic with bandwidth guarantees. In: *Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, p. 191. IEEE Computer Society, Washington, DC, USA, October 1997.

[127] Zhe Ma, Danielle Scarpazza, and Francky Catthoor. Run-time task overlapping on multiprocessor platforms. In: *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pp. 47–52, October 2007.

[128] Ted Maeurer. Cell Architecture and Broadband Engine Processor. In: *Proceedings of the 5th International Forum on Application-Specific Multi-Processor SoC (MPSoC)*, Relais de Margaux, France, July 2005.

[129] S. Mamagkakis, D. Atienza, C. Poucet, F. Catthoor, and D. Soudris. Energy-Efficient Dynamic Memory Allocators at the Middleware Level of Embedded Systems. In: *Proceedings of the Sixth ACM & IEEE International Conference on Embedded Software (EMSOFT 2006)*, pp. 215 – 222, Seoul, Korea, 2006.

[130] Hugo De Man. On Nanoscale Integration and Gigascale Complexity in the Post .Com World. In: *DATE '02: Proceedings of the conference on Design, Automation and Test in Europe*, p. 12. IEEE Computer Society, Washington, DC, USA, 2002.

[131] T. Marescaux, B. Brické, P. Debacker, V. Nollet, and H. Corporaal. Dynamic Time-Slot Allocation for QoS Enabled Networks-on-Chip. In: *Proceedings of the 3rd workshop on embedded systems for real-time multimedia (ESTIMedia)*, pp. 47–52, September 2005.

[132] Theodore Marescaux. *Mapping and Management of Communication Services on MP-SoC Platforms*. PhD thesis, Technische Universiteit Eindhoven (TU/e), September 2007.

[133] Theodore Marescaux. *MPSoC Design Flow Optimized QSDPCM Application*. Technical Report MPSOC Freescale Deliverable 4, IMEC, January 2007.

[134] Théodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs. In: *FPL '02: Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pp. 795–805. Springer-Verlag, September 2002.

[135] Théodore Marescaux, Jean-Yves Mignolet, Andrei Bartic, W. Moffat, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In: *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 595–605, 2003.

[136] Théodore Marescaux, Vincent Nollet, Jean-Yves Mignolet, Andrei Bartic, W. Moffat, Prabhat Avasare, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable SoCs. *Integration*, 38 (1): pp. 107–130, 2004.

[137] Marco Mattavelli and Massimo Ravasi. High Level Extraction of SoC Architectural Information from Generic C Algorithmic Descriptions. In: *IWSOC '05: Proceedings of the Fifth International Workshop on System-on-Chip for Real-Time Applications*, pp. 304–307. IEEE Computer Society, Washington, DC, USA, 2005.

[138] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In: *USENIX Annual Technical Conference*, pp. 279–294, 1996.

[139] Pedro Mejia-Alvarez, Eugene Levner, and Daniel Mosse. Power-optimized scheduling server for real-time tasks. In: *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 239–250. IEEE Computer Society, Washington, DC, USA, 2002.

[140] S. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. SPS: A Strategically Programmable System. In: *Proceedings of the Reconfigurable Architecture Workshop (RAW)*, 2001.

[141] Rick Merritt. Merged AMD-ATI plans combined CPU/GPU chips in '08. *EETimes*, October 2006.

[142] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag, 2000.

[143] J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. *Enabling Run-time Task Relocation on Reconfigurable Systems*, Chapter 6, pp. 69–80. Springer, 2005.

[144] Jean-Yves Mignolet, Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In: *Proceedings of the International Conference on Design, Automation and Test Europe (DATE)*, pp. 986–992, March 2003.

[145] Mikael Millberg, Erland Nilsson, Rikard Thid, Shashi Kumar, and Axel Jantsch. The Nostrum backbone - a communication protocol stack for networks on chip. In: *Proceedings of the VLSI Design Conference*, Mumbai, India, January 2004.

[146] Steve Muir and Jonathan Smith. AsyMOS - An Asymmetric Multiprocessor Operating System. In: *Open Architectures and Network Programming*, pp. 25–34, April 1998.

[147] Andre Nacul, Francesco Regazzoni, and Marcello Lajolo. Hardware Scheduling Support in SMP Architectures. In: *Proceedings of the Design Automation and Test in Europe conference (DATE)*, Nice , France, April 2007.

[148] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware implementation of a real-time operating system. In: *Proceeding of the 12th TRON Project International Symposium*, p. 34, 1995.

[149] David Needle. Intel's arizona plans are fabulous. *EETimes*, 2005.

[150] Nam Pham Ngoc, Gauthier Lafruit, Jean-Yves Mignolet, Geert Deconinck, and Rudy Lauwereins. QoS aware HW/SW partitioning on run-time reconfigurable multimedia platform. In: *Proceedings of ERSA*, pp. 84–92, 2004.

[151] P.N. Ngoc, G. Lafruit, J-Y. Mignolet, S. Vernalde, G. Deconick, and R. Lauwereins. A framework for mapping scalable networked applications on run-time reconfigurable platforms. In: *Proceedings of the 2003 International Conference on Multimedia and Expo (ICME 03)*, volume 1, pp. 469–472, July 2003.

[152] Erland Nilsson, Mikael Millberg, Johnny Oberg, and Axel Jantsch. Load distribution with the proximity congestion awareness in a network on chip. In: *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, p. 11126. IEEE Computer Society, Washington, DC, USA, 2003.

[153] Erland Nilsson and Johnny Öberg. *PANACEA - a case study on the PANACEA NoC - a Nostrum network on chip prototype.* Technical Report Technical Report TRITA-ICT/ECS R 06:01, School of Information and Communication Technology, Royal Institute of Technology (KTH), Sweden, 2006.

[154] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In: *Sixteen ACM Symposium on Operating Systems Principles*, pp. 276–287, Saint Malo, France, 1997.

[155] V. Nollet, P. Avasare, J-Y. Mignolet, and D. Verkest. Low Cost Task Migration Initiation in a Heterogeneous MP-SoC. In: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 252–253. IEEE Computer Society, Washington, DC, USA, 2005.

[156] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS)*, p. 174.1. IEEE Computer Society, 2003.

[157] V. Nollet, T. Marescaux, P. Avasare, and J-Y. Mignolet. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 234–239. IEEE Computer Society, Washington, DC, USA, 2005.

[158] V. Nollet, J-Y. Mignolet, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems. In: *Proceeding of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 81–87, June 2003.

[159] Vincent Nollet. *Pareto Point Switching Report*. M4.DDT Deliverable M4.DDT8, IMEC V.Z.W., Kapeldreef 75, 3001 Leuven, 2005.

[160] Vincent Nollet, Prabhat Avasare, Hendrik Eeckhaut, Diederik Verkest, and Henk Corporaal. Run-Time Management of a MPSoC Containing FPGA Fabric Tiles. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 16 (1): pp. 24–33, January 2008.

[161] Vincent Nollet, Prabhat Avasare, Diederik Verkest, and Henk Corporaal. Exploiting Hierarchical Configuration to Improve Run-Time MPSoC Task Assignment. In: *Proceeding of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 49–55, June 2006.

[162] Vincent Nollet, Théodore Marescaux, Diederik Verkest, Jean-Yves Mignolet, and Serge Vernalde. Operating-system controlled network-on-chip. In: *Proceedings of the 41st International Conference on Design automation (DAC)*, pp. 256–259. ACM Press, June 2004.

[163] Vincent Nollet, Diederik Verkest, and Henk Corporaal. A Quick Safari Through the MPSoC Run-Time Management Jungle. In: *Proceedings of the 5th IEEE workshop on embedded systems for real-time multimedia (ESTIMedia)*, pp. 41–46, Salzburg, Austria, October 2007.

[164] U.Y. Ogras and R. Marculescu. Prediction-based flow control for network-on-chip traffic. In: *Design Automation Conference, 2006 43rd ACM/IEEE*, pp. 839–844, 24-28 July 2006.

[165] J.K. Ousterhout. Scheduling techniques for concurrent systems. In: *Proceedings of Third International Conference on Distributed Computing Systems*, May 1982.

[166] O. Ozturk, M. Kandemir, S. W. Son, and M. Karakoy. Selective code/data migration for reducing communication energy in embedded MPSoC architectures. In: *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 386–391. ACM Press, New York, NY, USA, 2006.

[167] Karen Parnell. Could microprocessor obsolescence be history? *XCell Journal*, (45), Spring 2003.

[168] Rafael Parra-Hernandez and Nikitas J. Dimopoulos. A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35 (5): pp. 708–717, September 2005.

[169] R. Pasko, S. Vernalde, and P. Schaumont. Techniques to evolve a c++ based system design language. In: *Proceedings of the International Design, Automation and Test in Europe Conference (DATE)*, pp. 302–309, 4-8 March 2002.

[170] Milan Pastrnak and Peter H. N. de With. Multidimensional Model of Estimated Resource Usage for Multimedia NoC QoS. In: *27th Symposium on Information Theory in the Benelux*, pp. 109–116, Nooordwijk, The Netherlands, June 2006.

[171] Milan Pastrnak, Peter H. N. de With, and Jef van Meerbergen. Realization of QoS Management Using Negotiation Algorithms for Multiprocessor NoC. In: *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1912–1915, Kos, Greece, May 2006.

[172] Milan Pastrnak, Peter Poplavko, Peter H.N. de With, and Jef van Meerbergen. Hierarchical QoS concept for multiprocessor system-on-chip. In: *Proccedings of the Workshop On Resource Management for Media Processing in Networked Embedded Systems*, pp. 139–142, Eindhoven, The Netherlands, 2005.

[173] Intel Corp. Patrick P. Gelsinger CTO, Senior Vice President. Gigascale integration for teraops performance – challenges, opportunities, and new frontiers (dac2004 keynote), June 2004.

[174] Bruce Jacob Paul Kohout, Brinda Ganesh. Hardware support for real-time operating systems. In: *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 45–51. ACM Press, 2003.

[175] Pierre Paulin. SoC Platforms of the Future: Challenges and Solutions, July 2005.

[176] Pierre G. Paulin, Chuck Pilkington, Michel Langevin, Essaid Bensoudane, and Gabriela Nicolescu. Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In: *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 48–53. ACM Press, New York, NY, USA, 2004.

[177] Linda D. Paulson. TV Comes to the Mobile Phone. *IEEE Computer Society*, pp. 13–16, April 2006.

[178] Li-Shiuan Peh. *Flow control and micro-architectural mechanisms for extending performance of interconnection networks*. PhD thesis, Stanford University, 2001.

[179] Norman C. Hutchinson Peter Smith. *Heterogeneous Process Migration: The Tui System*. Technical report, University of British Columbia, 1996.

[180] Michele Pittau, Andrea Alimonda, and Salvatore Carta. Impact of Task Migration on Streaming Multimedia for Embedded Multiprocessors: A Quantitative Evaluation. In: *Proceedings of the 5th IEEE workshop on embedded systems for real-time multimedia (ESTIMedia)*, pp. 59–64, Salzburg, Austria, October 2007.

[181] Francesco Poletti, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose Mendias. An integrated hardware/software approach for run-time scratchpad-management. In: *Proceedings of the 41st International Conference on Design Automation (DAC)*, pp. 238–243, June 2004.

[182] Fred J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. In: *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, p. 2. IEEE Computer Society, Washington, DC, USA, 1999.

[183] Swaroop Sridhar Prashanth P. Bungale. An Approach to Heterogeneous Process State Capture/Recovery to Achieve Minimum Performance Overhead During Normal Execution. In: *Proceeding of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[184] Calton Pu and Jonathan Walpole. A case for adaptive os kernels. In: *Proceedings of the ACM Object Oriented Programming Systems, Languages and Applications*, 1994.

[185] J. Regehr, M. Jones, and J. Stankovic. *Operating System Support for Multimedia: The Programming Model Matters*. Technical Report MSR-TR-2000-89, Microsoft Research, September 2000.

[186] Javier Resano, Daniel Mozos, and Francky Catthoor. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In: *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pp. 106–111. IEEE Computer Society, Washington, DC, USA, 2005.

[187] Javier Resano, Daniel Mozos, Diederik Verkest, and Francky Catthoor. Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware. In: *Proceedings of the 41st annual conference on Design automation (DAC)*, pp. 119–124. ACM Press, New York, NY, USA, 2004.

[188] Chris Rowen. Using configurable processors for high-efficiency multiple-processor systems. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, p. 7, Las Vegas, Nevada, USA, June 2006.

[189] Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24 (1): pp. 4–17, January 2005.

[190] S. Russ, J. Robinson, M. Gleeson, and J. Figueroa. *Dynamic Communication Mechanism Switching in Hector*. Technical report, Mississippi State University, 1997.

[191] M. J. Rutten, E.-J. Pol, J. van Eijndhoven, K. Walters, and G. Essink. Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture. In: S. Sudharsanan, V. M. J. Bove, and S. Panchanathan (Eds.), *Proceedings of the SPIE (Embedded Processors for Multimedia and Communications II).*, volume 5683, pp. 53–63, March 2005.

[192] Martijn J. Rutten, Jos T. J. van Eijndhoven, and Evert-Jan D. Pol. Design of multi-tasking coprocessor control for Eclipse. In: *Proceedings of the tenth international symposium on Hardware/software codesign*, pp. 139–144. ACM Press, New York, NY, USA, 2002.

[193] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Tatiana Shpeisman, Mohan Rajagopalan, Anwar Ghuloum, Eric Sprangle, Anwar Rohillah, and Doug Carmean. Runtime environment for tera-scale platforms. *Intel Technology Journal*, 11 (3): pp. 207–215, August 2007.

[194] Erno Salminen, Ari Kulmala, and Timo D. Hamalainen. On network-on-chip comparison. In: *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pp. 503–510. IEEE Computer Society, Washington, DC, USA, 2007.

[195] Tobias Samuelsson, Mikael Åkerholm, Peter Nygren, Johan Stärner, and Lennart Lindh. A comparison of multiprocessor real-time operating systems implemented in hardware and software. In: *International Workshop on Advanced Real-Time Operating System Services (ARTOSS)*, Porto, Portugal, July 2003.

[196] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens. Hardware reuse at the behavioral level. In: *Design Automation Conference, 1999. Proceedings. 36th*, pp. 784–789, 21-25 June 1999.

[197] P. Schaumont and I. Verbauwhede. ThumbPod puts security under your Thumb. *Xilinx Xcell Journal*, 2003.

[198] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. Quick Safari Through the Reconfiguration Jungle. In: *Proceedings of the 38th Design Automation Conference (DAC)*, pp. 172–177, 2001.

[199] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens. A programming environment for the design of complex high speed asics. In: *Design Automation Conference, 1998. Proceedings*, pp. 315–320, 15-19 Jun 1998.

[200] Kelly A. Shaw. *Resource Management in Single-Chip Multiprocessors*. PhD thesis, Stanford University, March 2005.

[201] Mohammed El Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In: *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, March 2002.

[202] Peter Silverman. New lithography technologies. *Future Fab International*, 19, June 2005.

[203] H. Simmler, L. Levinson, and Reinhard Männer. Multitasking on FPGA Coprocessors. In: *10th International Workshop on Field-Programmable Logic and Applications (FPL)*, pp. 121–130. Springer-Verlag, 2000.

[204] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database and Multiprocessor Operating Systems*. McGraw Hill, 1994.

[205] Stergios Skaperdas and Constantinos Syropoulos. Guns, butter, and openness: On the relationship between security and trade. In: *American Economic Review*, 2001.

[206] Lodewijk T. Smit, Gerard J. M. Smit, Johann L. Hurink, Hajo Broersma, Daniel Paulusma, and Pascal T. Wolkotte. Run-time assignment of tasks to multiple heterogeneous processors. In: *Proceedings of the 4rd PROGRESS workshop on embedded systems*, pp. 185–192, October 2004.

[207] Lodewijk T. Smit, Gerard J. M. Smit, Johann L. Hurink, Hajo Broersma, Daniel Paulusma, and Pascal T. Wolkotte. Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture. In: *Proceedings of the International Conference on Field-Programmable Technology*, pp. 421–424, December 2004.

[208] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4 (2): pp. 131–143, February 1993.

[209] Chris Steketee, Weiping Zhu, and Philip Moseley. Implementation of Process Migration in Amoeba. In: *Proceedings of the 14th International Conference on Distributed Computing Systems*, pp. 194–201, 1994.

[210] Georg Stellner. Consistent Checkpoints of PVM Applications. In: *Proceedings of the First European PVM User Group Meeting*, 1994.

[211] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In: *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pp. 526–531, Honolulu, Hawaii, 1996.

[212] Steven F. Quigley Stephen M. Charlwood. The impact of routing architecture on reconfiguration overheads. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 102–108, 2003.

[213] P. Stravers and J. Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In: *International Symposium on VLSI Technology, Systems, and Applications*, pp. 184–187, 2001.

[214] P. Strobach. QSDPCM - A New Technique in Scene Adaptive Coding. In: *Proceeding of the 4th European Signal Processing Conference*, pp. 1141–1144, 1988.

[215] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. pp. 777–782, June 2007.

[216] Andrew S. Tanenbaum. A Comparison of Three Microkernels. *The Journal of Supercomputing*, 9 (1–2): pp. 7–22, 1995.

[217] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[218] Andrew S Tanenbaum. *Computer Networks*. Prentice Hall PTR, August 2002.

[219] B. D. Theelen and A. C. Verschueren. Architecture design of a scalable single-chip multiprocessor. In: *DSD '02: Proceedings of the Euromicro Symposium on Digital Systems Design*, p. 132. IEEE Computer Society, Washington, DC, USA, 2002.

[220] Bruce Thomas, Ben Close, John Donoghue, John Squires, Phillip De Bondi, and Wayne Piekarski. First person indoor/outdoor augmented reality application: Arquake. *Personal Ubiquitous Computing*, 6 (1): pp. 75–86, 2002.

[221] J.G. Kuhl T.L. Casavant. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14 (11): pp. 1578–1588, 1988.

[222] Nick Tredennick. The rise of reconfigurable systems. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 3–10, June 2003.

[223] J.W. van den Brand, C. Ciordas, K. Goossens, and T. Basten. Congestion-controlled best-effort communication for networks-on-chip. In: *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, pp. 1–6, 16-20 April 2007.

[224] L. Van der Perre, B. Bougard, J. Craninckx, W. Dehaene, L. Hollevoet, M. Jayapala, P. Marchal, M. Miranda, P. Raghavan, T. Schuster, P. Wambacq, F. Catthoor, and P. Vanbekbergen. Architectures and circuits for software-defined radios: Scaling and scalability for low cost and low energy. In: *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, pp. 568–569, San Francisco, CA, USA, February 2007.

[225] W. Van Raemdonck, G. Lafruit, E.F.M. Steffens, C.M. Otero Perez, and R.J. Bril. Scalable 3D graphics processing in consumer terminals. In: *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on*, volume 1, pp. 369–372, 26-29 Aug. 2002.

[226] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In: *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, pp. 98–99, San Francisco, CA, USA, February 2007.

[227] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens. Hardware/software partitioning of embedded system in OCAPI-xl. In: *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pp. 30–35. ACM Press, New York, NY, USA, 2001.

[228] M. Verma, L. Wehmeyer, and P Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS).*, pp. 104–109, September 2004.

[229] Herbert. Walder and Marco Platzner. Non-preemptive multitasking on FPGA: Task placement and footprint transform. In: *Proceeding of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 24–30, June 2002.

[230] Herbert Walder and Marco Platzner. Online Scheduling for Block-Partitioned Reconfigurable Devices. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pp. 290–295. IEEE Computer Society, March 2003.

[231] Herbert Walder, Christoph Steiger, and Marco Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In: *Proceedings of the Reconfigurable Architectures Workshop (RAW)*, 2003.

[232] John Walko. Mobile phone users demand decent batteries. *EETimes*, September 2005.

[233] Grant Wigley and David Kearney. Research issues in operating systems for reconfigurable systems. In: *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architectures (ERSA)*, pp. 10–16, June 2002.

[234] Clark Williams. *Linux Scheduler Latency*. Technical report, Red Hat Inc., 2002.

[235] D. Wingard. MicroNetwork-based integration for SOCs. In: *Proceedings of the International Design Automation Conference*, pp. 673–677, 2001.

[236] Pascal T. Wolkotte, Gerard J.M. Smit, Nikolay Kavaldjev, Jens E. Becker, and Jurgen Becker. Energy model of networks-on-chip and a bus. In: *Proceedings of the International Symposium on System-on-Chip*, pp. 82–85, November??? 2005.

[237] Clemens C. Wust, Reinder J. Bril, Christian Hentschel, Liesbeth Steffens, and Wim F.J. Verhaegh. QoS Control Challenges for Multimedia Consumer Terminals. In: *Proceedings of the International Workshop on Probabilistic Analysis Techniques for Real Time and Embedded Systems (PARTES)*, September 2004.

[238] Dror G. Feitelson Yair Wiseman. Paired Gang Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14 (5): pp. 581–592, May 2003.

[239] Peng Yang and Francky Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. In: *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 120–125. ACM Press, New York, NY, USA, 2003.

[240] Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins. Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs. *IEEE Des. Test*, 18 (5): pp. 46–58, 2001.

[241] Michael A. Yukish. *Algorithms to identify Pareto points in multi-dimensional data sets*. PhD thesis, 2004. Adviser-Tim Simpson.

[242] Raimo Haukilahti Zhonghai Lu. *NOC Application programming interfaces: high level communication primitives and operating system services for power management*, pp. 239–260. Kluwer Academic Publishers, 2003.

[243] B. B. Zhou, P. Mackerras, C. W. Johnson, D. Walsh, and R. P. Brent. An efficient resource allocation scheme for gang scheduling. In: *Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing*, pp. 187–194, 1999.

[244] Daniel F. Zucker, Ruby B. Lee, and Michael J. Flynn. An automated method for software controlled cache prefetching. In: *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, p. 106. IEEE Computer Society, Washington, DC, USA, 1998.

List of Publications

Book Chapter

Jean-Yves Mignolet, Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde and Rudy Lauwereins. Enabling Run-time Task Relocation on Reconfigurable Systems (Chapter 6). *New Algorithms, Architectures and Applications for Reconfigurable Computing*, Patrick Lysaght, Wolfgang Rosenstiel (Eds.), 2005, ISBN: 1-4020-3127-0

Journal Papers

1. Vincent Nollet, Prabhat Avasare, Hendrik Eeckhaut, Diederik Verkest and Henk Corporaal. Run-Time Management of a MPSoC containing FPGA Tiles. In *IEEE Transactions on Very Large Scale Integration Systems*, 16(1): 24–33, January 2008.

2. Hendrik Eeckhaut, Harald Devos, Peter Lambert, David De Schrijver, Wim Van Lancker, Vincent Nollet, Prabhat Avasare, Tom Clerckx, Mark Christiaens, Dirk Stroobandt, Rik Van de Walle and Peter Schelkens. Scalable, Wavelet-Based Video: From Server to Hardware-Accelerated Client. In *IEEE Transactions on Multimedia*, 9(7): 1508–1519, November 2007.

3. Chantal Couvreur, Vincent Nollet, Francky Catthoor, and Henk Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In *ACM Transactions on Embedded Computing Systems*, (awaiting publication).

4. Chantal Couvreur, Vincent Nollet, Theodore Marescaux, Erik Brockmeyer, Francky Catthoor and Henk Corporaal. Design-Time Application Mapping and Platform Exploration for MP-SoC Customized Run-Time Management. In *IEE Proceedings Computers & Digital Techniques*, 1(2):120–472, March 2007.

5. Iole Moccagatta, Vincent Nollet and Jean-Yves Mignolet. An architecture for Agile Rich-Media Platforms. Electronic Design, Strategy, News (EDN), Vol 50, September 2006.

6. Theodore Bartic, Jean-Yves Mignolet, Vincent Nollet, Theodore Marescaux, Diederik Verkest, Serge Vernalde and Rudy Lauwereins. Topology Adaptive Network-on-Chip Design and Implementation. In *IEE Computers and Digital Techniques*, 152(4): 467–472, July 2005.

7. Theodore Marescaux, Vincent Nollet, Jean-Yves Mignolet, Andrei Bartic, Will Moffat, Prabhat Avasare, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable SoCs. In *Integration, the VLSI Journal*, 38(1):107–130, 2004.

Conference Papers

1. Vincent Nollet, Diederik Verkest and Henk Corporaal. A Quick Safari Through the MPSOC Run-Time Management Jungle. In *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 41–46. ACM, Salzburg, Austria, October 2007.

2. Hendrik Eeckhaut, Mark Christiaens, Dirk Stroobandt and Vincent Nollet. Optimizing the critical loop in the H.264/AVC CABAC decoder. In *Proceedings of International Conference on Field Programmable Technology (FPT)*, pages 113–118, Bangkok, Thailand, December 2006.

3. Chantal Couvreur, Vincent Nollet, Francky Catthoor and Henk Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In *Proceedings of the International Symposium on System-on-Chip (SOC)*, pages 195–198, Tampere, Finland, November 2006.

4. Chantal Couvreur, Vincent Nollet, Theodore Marescaux, Erik Brockmeyer, Francky Catthoor and Henk Corporaal. Pareto-Based Application Specification for MPSoC Customized Run-Time Management. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 78–84, Samos, Greece, July 2006.

5. Vincent Nollet, Prabhat Avasare, Diederik Verkest and Henk Corporaal. Exploiting Hierarchical Configuration to Improve Run-Time MPSoC Task Assignment. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 49–55, Las Vegas, Nevada, USA, June 2006.

6. Chantal Ykman-Couvreur, Erik Brockmeyer, Vincent Nollet, Theodore Marescaux, Francky Catthoor and Henk Corporaal, Design-time Application Exploration for MP-SoC Customized Run-Time Management. In *Proceedings of the International Symposium on System-on-Chip (SOC)*. pages 66–69, Tampere, Finland, November 2005.

7. Theodore Marescaux, Benjamin Bricke, Peter Debacker, Vincent Nollet, and Henk Corporaal. Dynamic Time-Slot Allocation for QoS Enabled Networks on Chip. In *Proc. IEEE 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 47–52, New York, USA, September 2005.

8. Theodore Marescaux, Anders Rångevall, Vincent Nollet, Andrei Bartic, and Henk Corporaal. Distributed congestion control for packet switched networks on chip. In *Parallel Computing Conference (ParCo 2005), Proceedings*, Malagà, Spain, September 2005.

9. Prabhat Avasare, Vincent Nollet, Jean-Yves Mignolet, Diederik Verkest and Henk Corporaal. Centralized End-to-End Flow Control in a Best-Effort Network-on-Chip. In *Proceedings of the EMSOFT Conference*, pages 17–20, Jersey City, New Jersey, September, 2005.

10. Vincent Nollet, Theodore Marescaux, Prabhat Avasare, and Jean-Yves Mignolet. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 234–239, Washington, DC, USA, March 2005. IEEE Computer Society.

11. Vincent Nollet, Prabhat Avasare, Jean-Yves Mignolet and Diederik Verkest. Low Cost Task Migration Initiation in a Heterogeneous MP-SoC. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 234–239, Munich, Germany, March 2005.

12. Guillermo Talavera, Vincent Nollet, Jean-Yves Mignolet, Diederik Verkest, Serge Vernalde, Rudy Lauwereins and Jordi Carrabina. Hardware-software debugging techniques for reconfigurable systems-on-chip. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*, vol. 3, pages 1402–1407, Hammamet, Tunisia, December 2004.

13. Vincent Nollet, Theodore Marescaux, Diederik Verkest, Jean-Yves Mignolet, and Serge Vernalde. Operating System Controlled Network-on-Chip. In *Proceedings of the Design Automation Conference (DAC04)*, pages 256–259, San Diego, June 2004.

14. Theodore Bartic, Jean-Yves Mignolet, Vincent Nollet, Theodore Marescaux, Diederik Verkest, Serge Vernalde and Rudy Lauwereins. Highly Scalable Network on Chip for Reconfigurable Systems, In *Proceedings of the International Systems on Chip Conference (SOC)*, pages 79–82, Tampere, Finland, November 2003.

15. Vincent Nollet, Jean-Yves Mignolet, Theodore Bartic, Diederik Verkest, Serge Vernalde and Rudy Lauwereins. Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 81–87, Las Vegas, Nevada, USA, June 2003.

16. Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde and Rudy Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *Proceedings of the Reconfigurable Architectures Workshop (RAW)*, page 17, Nice, France, April 2003.

17. Jean-Yves Mignolet, Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde and Rudy Lauwereins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pages 986–991, Munich, Germany, March 2003.

Curriculum Vitae

Vincent Nollet was born in Veurne, Belgium on March 26, 1976. He obtained his MSc in Electrical Engineering in 1999 from the Vrije Universiteit Brussel (VUB). After graduation, he worked as an embedded software developer at Acunia until October 2001, after which he joined, as a researcher, the Nomadic Embedded Systems division, at the Interuniversity Micro Electronics Center (IMEC), Leuven, Belgium. He consequently received a Postgraduate Degree in Management from the Vrije Universiteit Brussel (VUB), Belgium, and a Master of Business Administration (MBA) Degree from the University of Hasselt (UHasselt), Belgium, in respectively 2002 and 2005. From begin 2005 till June 2007, he lead the IMEC MPSoC research activity in its quest for developing novel multiprocessor application mapping tools. Since July 2007, he heads a new research activity focusing on run-time management technology for MPSoC platforms. Meanwhile, from 2004 till 2008, he was also working towards the PhD degree with the department of Electrical Engineering at the Technische Universiteit Eindhoven (TU/e), The Netherlands.

Index