Dimitrios Soudris · Axel Jantsch *Editors*

Scalable Multi-core Architectures

Design Methodologies and Tools

Springer

Scalable Multi-core Architectures

Dimitrios Soudris • Axel Jantsch
Editors

Scalable Multi-core Architectures

Design Methodologies and Tools

Springer

Editors
Dimitrios Soudris
Department of Electrical and
Computer Engineering
National Technical University
of Athens
Heroon Polytechneiou 9
157 80 Athens
Zographou Campus
Greece
dsoudris@microlab.ntua.gr

Axel Jantsch
Department of Electronic Systems
Royal Institute of Technology
Forum 105
164 60 Kista
Sweden
axel@kth.se

ISBN 978-1-4419-6777-0 e-ISBN 978-1-4419-6778-7
DOI 10.1007/978-1-4419-6778-7
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011938281

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

The objective of the European research programme in Information and Communication Technologies (ICT) is to improve the competitiveness of European industry and enable Europe to shape and master future developments in ICT. ICT is at the very core of the knowledge-based society. EU research funding has as target to strengthen Europe's scientific and technology base and to ensure European leadership in ICT, help drive and stimulate product, service and process innovation through ICT use and value creation in Europe, and ensure that ICT progress is rapidly transformed into benefits for Europe's citizens, businesses, industry and governments.

Over the last years, the European Commission has constantly increased the amount of funding going to research in computing architectures and tools with special emphasis on multicore computing. Typically, European research funding in a new area (like multi/many cores) starts with funding for a Network of Excellence. Networks of Excellence are an instrument to overcome the fragmentation of the European research landscape in a given area by bringing together around a common research agenda the leading universities and research centers in Europe; their purpose is to reach a durable restructuring/shaping and integration of efforts and institutions.

In the following years, a number of collaborative research projects may also be funded to address specific, more industrially oriented, research challenges in the same research area. It is important to note here that collaborative research projects are the major route of funding in the European research landscape in a way that is quite unique worldwide. In European collaborative research projects, international consortia consisting of universities, companies and research centers are working together to advance the state of the art in a given area. The typical duration of such a project is 3 years.

In 2004, the European Commission launched the HiPEAC Network of Excellence. In 2006, the European Commission launched the Future and Emerging Technologies initiative in Advanced Computing Architectures as well as a number of projects covering Embedded Computing. In 2008, a new set of projects were launched to address the challenges of the multi/many core transition – in embedded,

mobile and general-purpose computing – under the research headings "Computing Systems" and "Embedded Systems"; these projects were complemented by a second wave of projects that have started in 2010 under the same research headings together with a new Future and Emerging Technologies initiative on "Concurrent Tera-device Computing". This effort continues in 2011 with two Calls for Proposals: one under the heading "Computing Systems" with 45 million euros funding and the other under the heading "Exascale Computing" with 25 million euros of funding.

The MOSART collaborative research project was funded to perform research on scalable multicore architectures targeting embedded applications. Results from MOSART are presented in this book providing a valuable reference point to researchers and engineers.

It has been a long way, but we now have an important computing research community in Europe, both from industry and academia, engaging in collaborative research projects that bring together strong European teams in cutting-edge technologies. The book that you have in your hands is a clear demonstration of the breakthroughs that can be obtained through European collaboration.

Panagiotis Tsarchopoulos
Project Officer
European Commission

Preface

Designing an application-specific multi-core System-on-Chip is a tricky endeavor and requires to make dozens of system level decisions with profound impact on performance and cost at an early time in the design phase, when many details are still unknown and performance estimates are notoriously inaccurate. Technology development is progressing and is continuously increasing our raw computing capacity. At the same time application requirements and standards become more demanding. Together this leads to complex designs that require sophisticated and continuously developed architectures, tools, and methodologies.

In particular since the field during the last decade has embarked to massively increase the number of heterogeneous cores, the picture has become very complicated. Moreover, it is quickly changing with respect to demands on architectures and tools. Hence, the design technology seems to fall more and more behind manufacturing technology, which is also referred to as the *design productivity gap* and illustrated in Fig. 1a. In order to get to grips with this disturbing situation, researchers have tried to develop *scalable* architectures and tools that continue to be efficient and usable even as the number of cores dramatically grows. The promise of scalable architectures or tools is that we can use them in 2011 with 20 or 50 cores, and we can still use them in 2017 with 200 or 500 cores. Once a scalable architecture for an application domain or a scalable tool for a design problem has been developed, the design research stops running behind the semiconductor manufacturing advances and can truly focus on exploiting the given capabilities of technology, as illustrated in Fig. 1b.

The European FP7 MOSART project (http://www.mosart-project.org/) has set as its goal to develop scalable solutions to architectures, tools, and methodologies. Figure 2 gives an overview of the MOSART design flow and also indicates which parts are covered in this book. Even though not all the work of the project are fully documented here, several important results of our work on architecture and hardware (Part I), system level design and exploration (Part II), and applications (Part III) are reported.

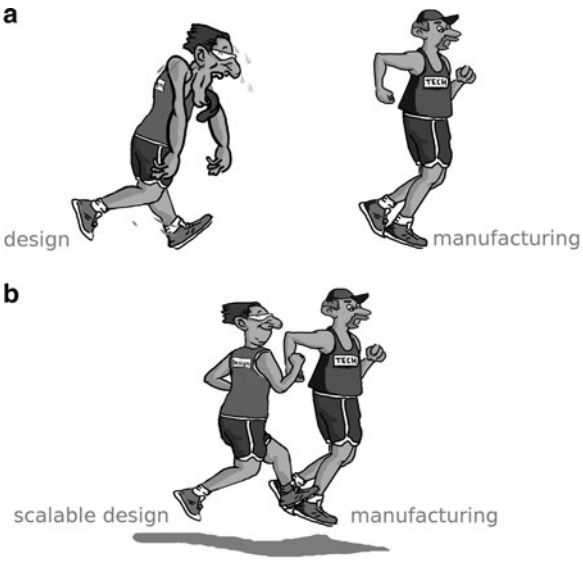

Fig. 1 Design technology catches up

The MOSART project was a joint effort from January 2008 till December 2010 by partners from industry (Thales Communication in Paris, Intracom in Athens, Arteris in Paris, and CoWare/Synopsys in Belgium and Germany), Research institutes (IMEC in Leuven and VTT in Oulu), and Universities (ICCS in Athens and KTH in Stockholm). Based on the combined competence, the team had the ambition to push new techniques in multi-core architecture platforms and design tools. Inspired from the applications provided by Thales and Intracom, as described in Chaps. 7 and 8, several innovations have been developed and demonstrated. A major part of these innovations are summarized in this book, and we sincerely hope it will be useful and stimulating for researchers and industrial practitioners in the area.

We would like to extend our gratitude to all members of the MOSART team for an exciting project collaboration, many inspiring discussions, and for the warm company at many meetings between mild Rhodes Island in the South and chilly Oulu in the North of Europe. We want to thank the authors for the contributions and the hard work on the chapters of this book making it into a concise and representative summary of 3 years of research and development. Furthermore, we would like to express our appreciation for the project reviewer's comments and feedback that were always insightful and to the point, and that greatly helped us to stay focused, to increase our efforts, and to keep our overall objectives in mind. They have certainly contributed to make the results of higher quality. We would also like to thank our project officers (Ms. Zulema Olivan-Tomas and Ms. Margot Bezz) for their professional and sensible handling of MOSART, and last but not least, we are very grateful to the team of Springer who has enthusiastically supported this book from the very beginning, very professionally transformed the material into a high quality publication, and kept patience and support when the delivery of the material were behind schedule.

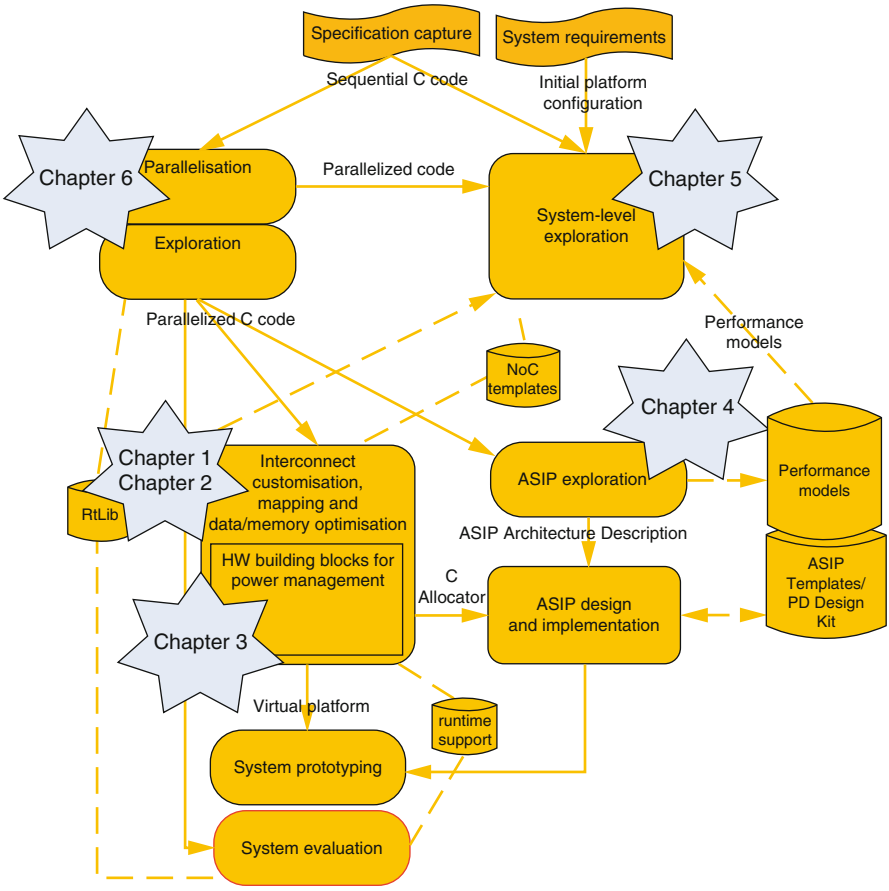

Fig. 2 Multi-core SoC design flow

Most importantly we hope that you, the reader, will enjoy reading this book and it triggers inspiration and many new ideas.

Athens and Stockholm

Dimitrios Soudris
Axel Jantsch

Contents

Contributors

Sylvain Aguirre Thales Communications and Security, Paris, France, Sylvain.AGUIRRE@fr.thalesgroup.com

Iraklis Anagnostopoulos National Technical University of Athens, 9 Heroon Polytechneiou, Zographou Campus, Greece, iraklis@microlab.ntua.gr

Thomas J. Ashby IMEC Belgium, Kapeldreef 75, B-3001 Leuven, Belgium, ashby@imec.be

Alexandros Bartzas National Technical University of Athens, 9 Heroon Polytechneiou, Zographou Campus, Greece, alexis@microlab.ntua.gr

Bernard Candaele Thales Communications and Security, Paris, France, bernard.CANDAELE@fr.thalesgroup.com

Jean-Michel Chabloz ES Department, School of ICT, KTH, Isafjordsgatan 39, FORUM 120, 16440 Kista, Sweden, chabloz@kth.se

Xiaowen Chen Royal Institute of Technology, Stockholm, Sweden, xiaowenc@kth.se

Ahmed Hemani ES Department, School of ICT, KTH, Isafjordsgatan 39, FORUM 120, 16440 Kista, Sweden, hemani@kth.se

Frank Ieromnimon INTRACOM S.A.Telecom Solutions, Peania, Greece, fier@intracom.gr

Axel Jantsch Royal Institute of Technology, Stockholm, Sweden, axel@kth.se

Jari Kreku VTT Technical Research Centre of Finland, Kaitoväylä 1 FI-90570 Oulu, Finland, jari.kreku@vtt.fi

Dimitrios Kritharidis INTRACOM S.A.Telecom Solutions, Peania, Greece, dkri@intracom.gr

Zhonghai Lu Royal Institute of Technology, Stockholm, Sweden, zhonghai@kth.se

Abdul Naeem Royal Institute of Technology, Stockholm, Sweden, abduln@kth.se

Sandro Penolazzi Royal Institute of Technology, Stockholm, Sweden, sandrop@kth.se

Dimitrios Soudris National Technical University of Athens, 9 Heroon Polytechneiou, Zographou Campus, Greece, dsoudris@microlab.ntua.gr

Kari Tiensyrjä VTT Technical Research Centre of Finland, Kaitoväylä 1 FI-90570 Oulu, Finland, kari.tiensyrja@vtt.fi

Geert Vanmeerbeeck IMEC Belgium, Kapeldreef 75, B-3001 Leuven, Belgium, vanmeerb@imec.be

Bart Vanthournout Synopsys, Interleuvenlaan 15A B-3001 Leuven, Belgium, bartv@synopsys.com

Nikolaos S. Voros Technological Educational Institute of Mesolonghi, Department of Telecommunication Systems & Networks (consultant to Intracom Telecom Solutions S.A), Greece

Andreas Wieferink Synopsys, Team4 Building, Kaiserstrasse 100 D-52134 Herzogenrath, Germany, andreasw@synopsys.com

Sotirios Xydis National Technical University of Athens, 9 Heroon Polytechneiou, Zographou Campus, Greece, sxydis@microlab.ntua.gr

Yuang Zhang Royal Institute of Technology, Stockholm, Sweden, yazhang@kth.se

Part I
HW/SW/ Building Blocks: Architecture, Methods, and Techniques

Chapter 1
Memory Architecture and Management in an NoC Platform

Axel Jantsch, Xiaowen Chen, Abdul Naeem, Yuang Zhang, Sando Penolazzi, and Zhonghai Lu

Abstract The memory organization and the management of the memory space is a critical part of every NoC based platform design. We propose a Data Management Engine (DME), that is a block of programmable hardware and part of every processing element. It off-loads the processing element (CPU, DSP, etc.) by managing the memory space, memory access and the communication over the on-chip network. The DME's main functions are virtual address translation, private and shared memory management, cache coherence protocol, support for memory consistency models, synchronization and protection mechanisms for shared memory communication. The DME is fully programmable and configurable thus allowing for customized support for high level data management functions such as dynamic memory allocation and abstract data types. This chapter describes the main concepts, design and functionality of the DME and presents case studies illustrating its usage and performance.

Keywords Network on Chip • SoC Architecture • Memory Organization

1.1 On-Chip Memory Organization

On-chip Computation is moving away from a sequential to a parallel paradigm leading to dozens, hundreds, and soon even thousands of cores and computational units on a single die. These many core chips can be highly homogeneous or irregular and heterogeneous, depending on the application area and market segment. At the same time, the communication infrastructure is developing into a similarly parallel

A. Jantsch (✉) • X. Chen • A. Naeem • Y. Zhang • S. Penolazzi • Z. Lu
Royal Institute of Technology, Stockholm, Sweden
e-mail: axel@kth.se; xiaowenc@kth.se; abduln@kth.se; yazhang@kth.se; sandrop@kth.se; zhonghai@kth.se

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_1,

structure, which is often called a Network-on-Chip (NoC). Shared, serial buses are replaced by pipelined communication networks that allow hundreds or thousands of communications going on concurrently at any time.

Only the third member of the trio computation, communication, storage, is lacking behind in this rapid transformation of sequential into massively parallel activity. Among the profound obstacles against making memory access as parallel as computation and communication, are the usage of shared memory for communication and temporary storage, and the large divide between on-chip and off-chip memory technology.

In 1995 Wulf and McKee described in a remarkable, short article a trend that is since then known as the approach to the memory wall [32]. Based on the observation that processor speed grows by 80% every year but memory access time decreases only by 7% per year, they predict that, if no invention breaks this trend, program performance is solely limited by memory access within 5 to 12 years (in year 2000 and 2007, respectively). Before we discuss the situation today, it is worthwhile to recall Wulf and McKee's original argument. Consider the equation

$$t_{\text{avg}} = p \times t_c + (1 - p) \times t_m$$

where t_{avg} is the average access latency to access a data word, p is the probability of a cache hit, t_c is the access time to the cache, and t_m is the access time to main memory. If a program has 1 memory instruction per 4 other instructions, the memory wall is hit when $t_{\text{avg}} \geq 5$ cycles (assuming one instruction takes 1 cycle to execute). When this condition is met, the program execution time is determined by the access time to main memory and there is no benefit to improve the processor performance.

Since 1995 many architectural innovations have avoided a hard impact into the memory wall. Improved interfaces such as DDR, DDR2, and DDR3 have increased the bandwidth to DRAM memory; packaging technology has increased the pin count such that two or four DDR3 interfaces are feasible today; additional levels of caches have helped to avoid the penalty of going off-chip. The former two effectively keep t_m low while the latter increases p.

To illustrate current state of the art, consider a fictive example with a 10×10 core chip, each core executes three instructions per cycle (either due to pipelining or with multiple functional units). There is an L1 cache for each core and one L2 cache on chip. The chip has four 64bit DDR3 interfaces. For the sake of simplicity we assume the access time to L1 is 1 cycle, to L2 is 10 cycles and to external DRAM it is 20 cycles. Let the hit ratio p be 90% for both caches. With a slightly generalized version of above equation, taking into account a second level of caching, we obtain $t_{\text{avg}} = 2.0$ cycles. If each core would execute 1 instruction per cycle, we would be fine since we are below the limit $t_{\text{avg}} = 5$. But because each core can execute 3 instructions per cycle and every 5th instruction on average is a memory access, our limit is in fact $t_{\text{avg}} = 5/3 = 1.67$. Thus, our chip operates close to the memory wall and its performance is limited by the access latency to external memory. In fact it seems to be a fairly balanced system where neither the processors nor the memory access is over- or under-dimensioned.

However, this example assumes there is no contention in the network, at the DRAM ports, at the L2 cache, we have no delays due to cache coherence, memory consistency issues, or shared variable protection. If any of these issues appear, it will increase the t_{avg}, effectively making the memory access the bottleneck. Increasing the number of cores will significantly worsen the situation because it will increase the demand for on-chip data more than the off-chip interface will allow to pass through.

Wulf and McKee speculated that the most convenient resolution to the problem would be "the discovery of a cool, dense memory technology whose speed scales with that of processors". Such a technology may in fact come to our rescue. ZRAM [2] and the memristor [31] are contenders for very dense, very fast memory technology with nicely scalable access times. Integration of logic dies with memory dies in a 3D stack is already viable today and has very strong potential for the high performance, high volume consumer markets in the next few years. Placing DRAM dies on top of the logic dies puts the memory within a few tens of μm distance from the core that needs the data. Through Silicon Vias (TSVs), the interconnect between two vertically stacked dies, occupy an area of less than $100\,\mu m^2$, consume less than $25\,\mu W$, and incur a latency of less than a few hundred picoseconds. All this together gives us performance per power and cost improvements of 1000X to 10000X over off-chip connections [19, 30]. Hence, high density, low cost memory can be placed very close to the cores of the many-core chip, effectively bringing t_{avg} to 1 and eliminating one or two cache levels.

However, there is a catch, requiring architectural innovation in addition to manufacturing technology. The geometric distance and thus the access latency varies a lot depending on where in the system the core and the accessed memory are. A core can fetch a variable within one cycle if it is stored in the memory bank just above, but it may take 10–100 cycles or more to fetch it from across the chip. Already Wulf and McKee have suggested that it may be wise to abandon the assumption that access time is uniform to all parts of the address space, and in [19] it is shown, that the principle performance limit in a 3D stacked system and a projected 17 nm technology is a factor 34 higher for a distributed memory system compared to a centralized one. Following the same logic, Kim et al. [20] have proposed a *Non-uniform Cache Architecture (NUCA)* in 2002 and many others have elaborated this idea since then (e.g. [4, 10, 11, 18, 33]). When 3D integration became a realistic option, a number of groups used it to integrate logic and memory, first by reusing traditional memory components [23, 25]. In a next step, customized memory architecture were proposed to fully exploit the potential [22, 24].

Reviewing these arguments and the technology trends, we conclude that a distributed memory organization with massively parallel access to different parts of the memory will allow for scalability en par with many core architectures and general on-chip communication networks. It will avoid memory access to become a bottleneck even for systems with thousands of cores and NoCs with raw bandwidth in the range of petabit per seconds.

In order to make distributed memory an attractive option to system designers, we need to address all the caveats and problems that typically come with distributed

memory schemes. One first, critical question is, if the memory space is shared or not. Keeping memory only local and private is an elegant architecture solution but ignores rather than solves the issues of legacy code and programming convenience. A lot of legacy software assumes a shared memory model and most programmers find it easier to express themselves within a shared memory programming paradigm rather than a model that is strictly based on message passing. Consequently, we believe that the architecture should support both a private and a shared memory organization.

Distributed, shared memory (DSM) schemes need to address cache coherency, memory consistency, and synchronized and protected access to shared variables. In order to offer solutions for all these and potentially many other memory and data management related tasks, we propose a programmable controller, that is optimized for managing the memory and data. It is called *Data Management Engine (DME)* and it is described in the following sections.

The DME provides efficiency through specialization. It can be viewed as an attempt to increase the parallelism on chip and to increase efficiency by means of specialization. By increasing parallelism higher performance can be gained at the same frequency; by specialization, less energy is expended for the same task. Since memory access is common in all applications and determines to a large extent performance, cost, and power consumption, it is reasonable to expect that a specialized memory transaction handler can improve performance and efficiency at modest cost.

1.2 DME-Enhanced Multi-Core NoC Platform

The basic concepts of the DME have been first introduced at DATE 2010 [8]. It has since then been used for a range of different applications and in different ways, for instance for different synchronization techniques [5, 7, 9] and for run-time partitioning of private/shared memory [6]. In this section we review the basic architecture of the DME, its main functions and features. In the following sections we then touch upon some of the applications and usages of the DME.

Figure 1.1a shows an example of our DME-enhanced multi-core NoC platform. The system is composed of 16 Processor-Memory (PM) nodes interconnected via a packet-switched network. The network topology is a mesh. As shown in Fig. 1.1b, each PM node contains a processor core, a DME, a local memory, and other hardware modules connected to the local bus. In our experiments, we use Leon3 [1] as the processor core, but any other core, or IP, or local computing cluster could be used as well. The DME connects the local processor core, the local memory, and the network. It not only handles all memory transactions from a local processor to both local memory and remote memory on- or off-chip, but also serves remote memory requests from remote processors via the network.

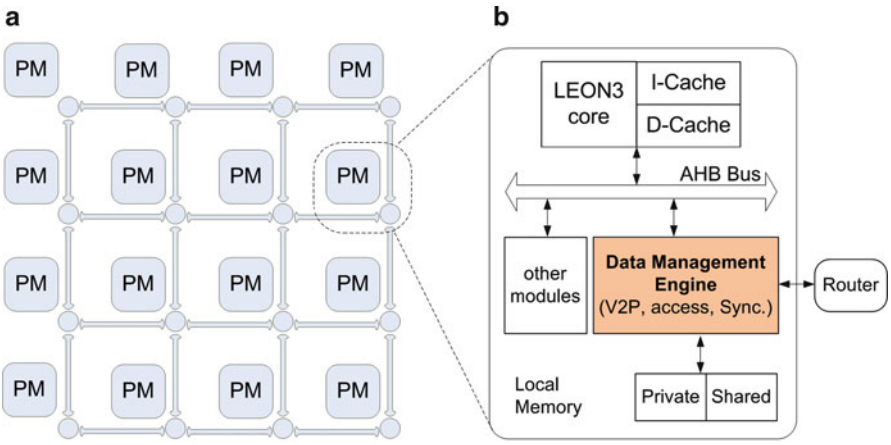

Fig. 1.1 In a multi-core network (**a**) each node contains its own DME (**b**) for handling memory transactions. The figure illustrates the experimental platform used

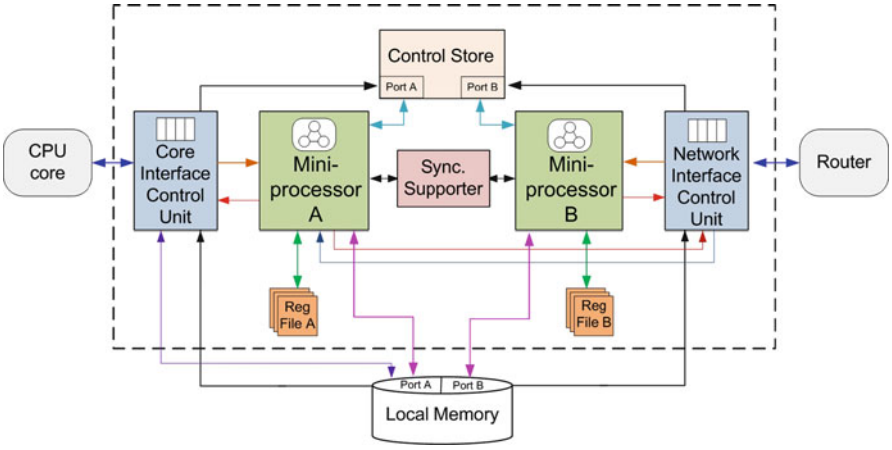

Fig. 1.2 Architecture of the Data Management Engine

The DME contains two programmable micro-controllers called *mini-processor A* and *mini-processor B* (Fig. 1.2). Their instruction sets are identical and include general purpose instructions such as load, store, addition, subtraction, condition testing, and branching, as well as a number of specialized instructions for support of synchronization, message generation, and address manipulations. In principle, all kinds of functions can be realized by the DME, since it is programmable, and the main local processor can use it as a local, specialized co-processor to which it can off-load memory management functions.

The two mini-processors fetch their instructions from a local control store, which they share, but operate on their own private set of registers. Mini-processor A is triggered by commands from the local processor that are received through the *Core Interface Control Unit (CICU)*. Mini-processor B is either invoked by a command from the local mini-processor A or by a request received from a remote node through the *Network Interface Control Unit (NICU)*. The *Sync Supporter* manages the access to locks by means of two specialized instructions, *load linked (ll)* and *store linked (sl)*. It allows for implementation of locks and barriers.

1.3 Data Management Engine (DME)

Although it is optimized for handling memory transactions and for managing the memory and address space, the DME is a programmable dual-core controller with a fairly general instruction set. First we describe the overall architecture of the DME and its components (Sect. 1.3.1), then we elaborate on the execution flow of the DME (Sect. 1.3.3) and the methodology for developing new micro-programs (Sect. 1.3.4).

1.3.1 Architectural Design

As shown in Fig. 1.2, the DME, which connects to the CPU core, the local memory, and the network, mainly contains six parts, namely, *Core Interface Control Unit (CICU)*, *Network Interface Control Unit (NICU)*, *control store*, *mini-processor A*, *mini-processor B*, and *Synchronization Supporter*. As their names suggest, the CICU provides a hardware interface to the local core, and the NICU a hardware interface to the network. The two mini-processors are the central processing engine. A microprogram is initially stored in the local memory, and is dynamically uploaded into the control store on demand during the program execution. The synchronization supporter coordinates the two mini-processors to avoid simultaneous accesses to the same memory address and it guarantees atomic read-and-modify operations (see Sect. 1.4.3). Both the local memory and the control store are dual ported: port A and B, which connect to the mini-processors A and B, respectively. The functions of each module are detailed as follows:

1.3.1.1 Core Interface Control Unit

The CICU connects with the core, the mini-processor A, the NICU, the control store and the local memory. Its main functions are: (i) it receives local requests in form of commands from the local core and triggers the operation of the mini-processor A accordingly; (ii) if the microcode is not already in the control store, it uploads it from the local memory to the control store through port A; (iii) it receives results from

the mini-processor A; (iv) it accesses the private memory directly using physical addressing if the memory access is private; (v) it sends results back to the local core.

1.3.1.2 Network Interface Control Unit

The NICU connects the network, the mini-processor B, the CICU, the control store, and the local memory. Its main functions are: (i) it receives remote requests in form of commands from the network and triggers the operation of the mini-processor B accordingly; (ii) if the microcode is not already in the control store, it uploads it from the local memory to the control store through port B; (iii) it sends remote requests from the mini-processor A or B to remote destination nodes by packaging the request in a message via the network; (iv) it receives the remote results as messages from remote destination nodes via the network, unpacks them, and forwards them to the mini-processor A or B.

Mini-Processor A

The mini-processor A connects with the CICU, the register file A, the synchronization supporter, the control store, and the local memory. Its operation is triggered by a command from the local core. It executes microcode from the control store through port A, uses register file A for temporary data storage, and accesses the local memory through port A.

Mini-Processor B

The mini-processor B connects with the NICU, the register file B, the synchronization supporter, the control store, and the local memory. Its operation is triggered by a command from remote cores via the network. It executes microcode from the control store through port B, uses register file B for temporary data storage and accesses the local memory through port B.

The two mini-processors feature a five-stage pipeline and four function units: *Load/Store Unit (LSU)*, *Adder Unit (AU)*, *Condition Unit* and *Message Passing Unit (MPU)*, to provide operations of memory access, addition, conditional branching, and message-passing. The microinstructions are designed to exploit the hardware architecture of the mini-processors. the microinstructions are organized *horizontally* [29].

1.3.1.3 Synchronization Supporter

The synchronization supporter, which connects with the mini-processor A and B, is a hardware module to support atomic read-and-modify operations. This is necessary to support locks when two synchronization requests try to access the same lock at the same time.

Table 1.1 Synthesis results with 90 nm SMIC technology

	Optimized for area	Optimized for speed
Frequency	448 MHz (2.23 ns)	500 MHz (2 ns)
Area (logic)	46 k NAND gates	57 k NAND gates
Area (control store)	237k NAND gates	

Table 1.2 DME power consumption at 400 MHz and implemented with 90 nm TSMC

	Power consumption [mW]
Mini A	6.9
Mini B	7.0
NICU	2.3
CICU	5.2
Synchronizer	0.2
DME total	21.6

1.3.1.4 Control Store

The control store, which connects with the CICU, the NICU, the mini-processor A and B, and the local memory, is a local storage for microcode. It acts as an instruction cache and dynamically uploads microcode from the local memory. It feeds microcode to the mini-processor A through port A, and the mini-processor B through port B. This uploading and feeding are controlled by the CICU for commands from the local core and the NICU for commands from remote cores via the network.

In summary, the DME features: (i) dual interfaces and dual processors, (ii) cooperation of the interface units and the mini-processors, (iii) dual-port shared control store and local memory, (iv) hardware support for mutex synchronization, and (v) dynamic uploading of microcode into the control store.

1.3.2 DME Implementation

The DME design has been synthesized by Synopsys Design Compiler in SMIC 90 nm technology and the control store is generated by Artisan Memory Compiler. The control store is 2048 × 128b dual port SRAM. Table 1.1 shows the results of the synthesis in terms of maximum frequency and gate count. For power analysis the DME has been synthesizes with the Cadence RTL compiler with TSMC 90 nm technology, and the gate netlist has been simulated for all possible transactions and instructions. Table 1.2 gives the power consumption of the different DME components. The power consumption is given for an idle DME at 400 MHz clock frequency. Depending on the operation of the DME, the power consumption has been observed to be up to 15–20% higher compared to the idle state.

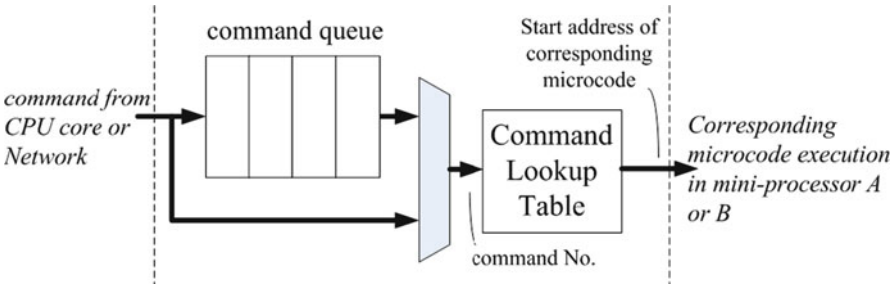

Fig. 1.3 Command-triggered microcode execution

1.3.3 Command-Triggered Microcode Execution

For the DME, the execution of the mini-processors is triggered by requests (in form of commands) from the local and remote cores. This is called *command-triggered microcode execution*.

As shown in Fig. 1.2, the two interface units are coupled with their corresponding mini-processors to support the *command-triggered microcode execution*. The two interface units are pure hardware modules responsible for receiving commands from the local CPU core and remote cores via the on-chip network, respectively, and then triggering the execution of the two mini-processors. The two mini-processors are microprogrammable. Figure 1.3 illustrates command-triggered microcode execution. As shown in the figure, there is a *command queue* as well as a *Command Lookup Table (CLT)* in each interface unit. The queues buffer commands from the CPU core or remote cores via the on-chip network. If both the command queue is empty and the mini-processor is idle, the command bypasses the command queue to reach the CLT directly.

The Command Lookup Table (CLT) reflects the correspondence of a command and a microcode. The CLT is indexed by the command to output the start address of the command's corresponding microcode. The start address is forwarded to the mini-processor, so the mini-processor is able to know where the current microcode execution starts. Figure 1.4a shows an example CLT. The "*Symbol*" is mnemonic. The command "*Number*" has a one-to-one correspondence with the "*Start_Addr*" of the related microcode. Figure 1.4a lists several commands we have implemented, and Fig. 1.4b shows snapshots of three pieces of microcode. As we can see, "LOAD_HWORD" command with its command No. 4 is responsible for loading a half word from the local memory. The start address of its related microcode is 24, so in the CLT we have an item recording the relationship between "LOAD_HWORD" command and its microcode.

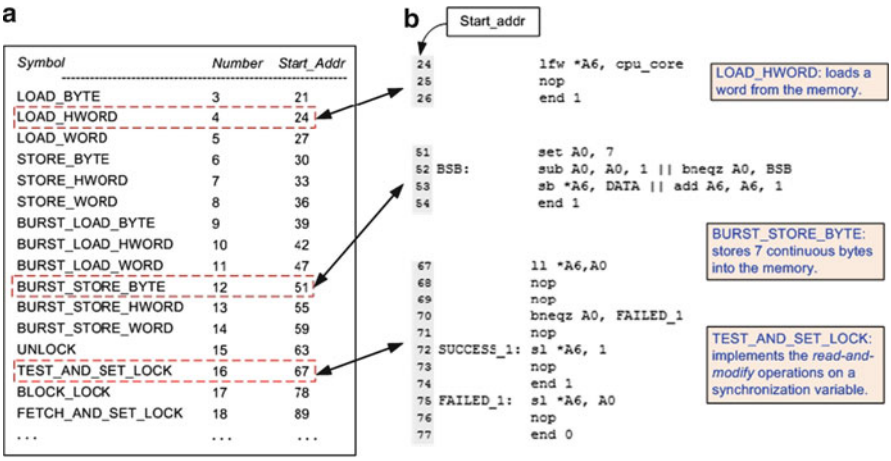

Fig. 1.4 (**a**) Command Lookup Table (CLT) and (**b**) microcode segments

As illustrated in Fig. 1.5, the DME works as follows (the microprogram is initially stored in the local memory):

① The CICU/NICU receives a command from the local or a remote core.
② A command will trigger the uploading of its microcode from the local memory to the control store. The control store has limited storage. If there is no space available when uploading the microcode to the control store, a replacement policy will be activated to replace a microcode with the currently activated one.
③ Then the mini-processor A or B will generate addresses to load the microinstruction from the control store to the datapath of the mini-processor.
④ The mini-processor A or B executes the microinstructions of the microcode.

This procedure is iterated during the entire execution period of the system.

1.3.4 Microprogramming Development Flow

Developers can develop new micro-programs for the DME in order to customize existing functions for a specific application and/or architecture, or to develop whole new functions. The current, rather rudimentary, tool support consists of an assembler and a configuration tool. For future development a C compiler and debugging support is planned. Figure 1.6 depicts the entire flow of microprogramming the DME. The flow consists of six steps:

① The user specifies the function to be implemented in the DME.
② A new generic command is defined according to the function specification.

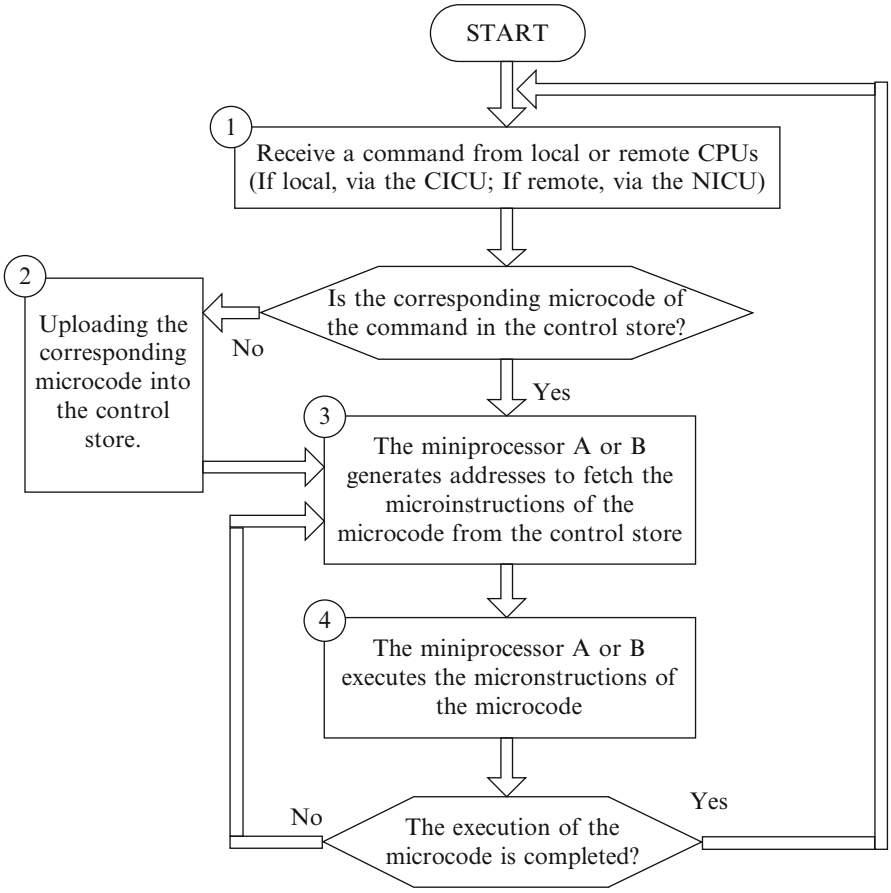

Fig. 1.5 The execution flow of the DME

③ A newly defined command is added into the CLT or an existing command is updated or removed by the "*CLT command*".
④ The user writes microcode in the DME assemble language for the specified function, then translates it into executable binary code by the DME assembler.
⑤ The binary microcode is uploaded from the local memory into the control store. If the programmer omits this step, the code is uploaded automatically at run time when needed.
⑥ The DME library has to be updated in order for the newly defined command to be supported by the compiler and used by application programs.

Users can follow this flow iteratively to add, update and delete commands of different functions. For instance, as shown in Fig. 1.6, our function specification is implementing a function to store a half word into the local memory. In step ②,

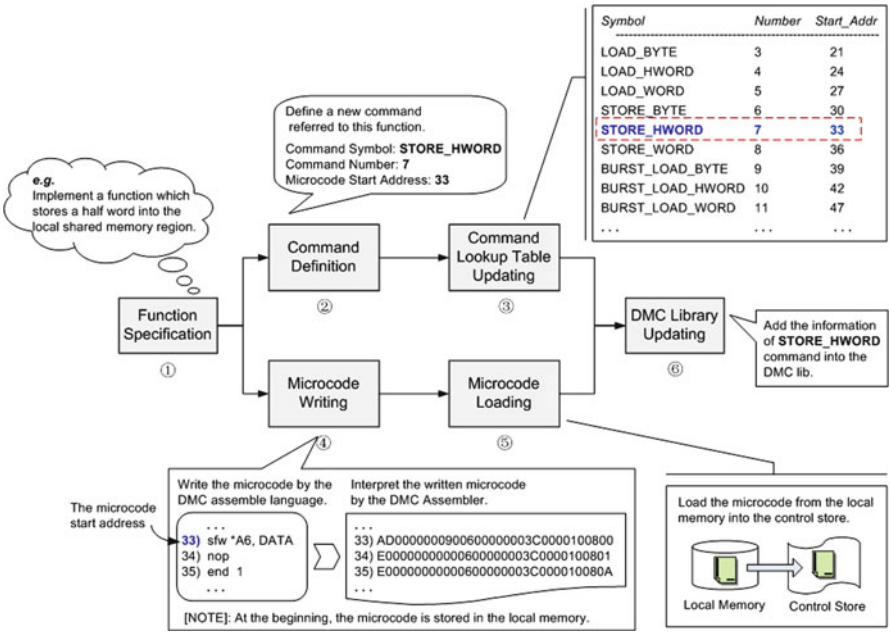

Fig. 1.6 Microprogramming development flow

we define a new generic command with Command Symbol **STORE_HWORD**, Command No. **7** and the start address **55**. The start address is obtained in step ④ after the corresponding microcode is written and interpreted. In step ③, the newly defined command is added into the Command Lookup Table (see the red dashed box) using "*CLT command*". In step ⑤, the binary code of the newly written microcode is uploaded from the local memory to the control store using "*MDL command*". Finally, write a user-defined function to add the information of **STORE_HWORD** into the DME Library. Afterwards, the command **STORE_HWORD** can be used successfully.

1.4 DME Applications

Due to its general purpose nature, a wide range of memory and data management functions can be realized on the DME in many different ways. So far we have experimented with memory partitioning, virtual address space, synchronization, cache coherency, local synchronization, memory consistency, and dynamic memory allocation. We describe some of these functions in the following sections to illustrate the usage and efficiency of the DME. The design and implementation of a complete dynamic memory allocator on the DME is described in Chap. 2.

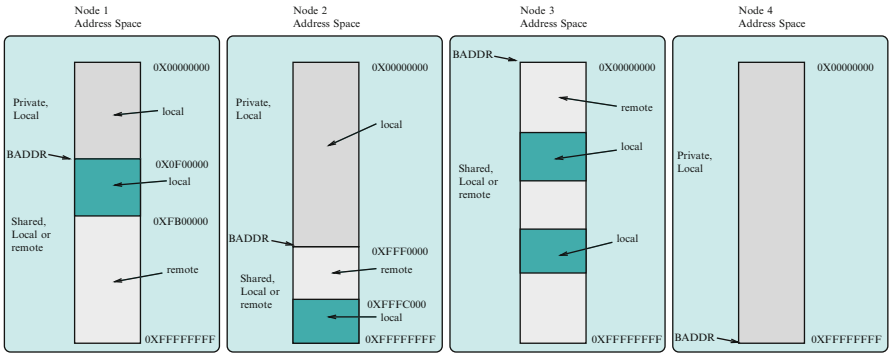

Fig. 1.7 Different nodes may partition their address space differently

1.4.1 Private and Shared Memory Partitions

A main source of overhead is the size of the shared memory space. For virtual-to-physical address translation, the size of translation tables are a linear function of the number of pages in the shared address space. Likewise, in a directory based cache coherence protocol, the size of the directory is in some schemes directly proportional to the size of the total shared memory. Moreover, access to shared memory carries a significant latency penalty, compared to private memory access, due to address translations, table look-ups, coherence, and consistency protocols. Consequently, it is desirable to keep the shared memory space as small as possible.

The DME allows the application programmer to specify the size of private and shared memory partitions in a very flexible way. These partitions can be different in different nodes and can be dynamically modified to track the varying needs of an application at run-time.

The DME maintains a register called BADDR (Base Address), that is a pointer dividing the private from the shared address space. If a memory transaction uses an address less or equal to this number, it addresses the private space; otherwise it is a shared address. This pointer can be set differently in different nodes, as illustrated in Fig. 1.7, and it can be modified dynamically. Local, private memory accesses are handled by mini-processor A only and are very fast. Essentially, they are simply forwarded by the DME to the local memory port within one cycle. Access to shared and remote memory requires more bookkeeping and incur a few cycles delay inside the DME. Requests to a remote memory location are handed over to mini-processor B, which packs the memory transaction into a packet and sends it over to the remote DME through the network. At the far side mini-processor B receives the request, accesses its own local memory, and sends the response back over the network. Locally, mini-processor B receives the response and hands it over to the main processor.

Table 1.3 Supported combinations of memory access features

local	/ private / physical	Supported
local	/ private / virtual	–
local	/ shared / physical	–
local	/ shared / virtual	Supported
remote	/ private / physical	–
remote	/ private / virtual	–
remote	/ shared / physical	–
remote	/ shared / virtual	Supported

If we have a physical address space only, a memory translation process based on equally sized pages yields the details of the remote memory address. A table in local memory is used to translate the remote address into a node id, a page address in the remote memory, and a page offset.

The dynamic change of the BADDR pointer gives the platform user very high flexibility. Moreover, it allows for the usage of tricks to maximize performance. For instance, consider a producer node that generates an array of data, which another consumer node uses for further processing. The array can be allocated in the shared memory part of the consumer's local memory. The producer writes to this array, allowing the write access latency over the network to be hidden since the producer does not have to wait until the writes complete. Once the full array is generated, the consumer node can change BADDR such that the array now falls into local, private memory. Access to this part would now be very fast because the bookkeeping overhead of shared memory access can be avoided.

Tricks like these can maximize performance, but they can potentially lead to inconsistent system states and hard-to-find errors. Therefore, we consider to implement an automatic table update mechanism triggered by a change to BADDR.

1.4.2 Virtual Address Space

The current version of the DME also supports a virtual address space giving flexibility to application programmers and compilers because the addresses used in a program can be kept separate of other programs and independent of where in the system the program is executed. On the other hand, the management of a virtual address space and the translation from virtual to physical addresses incur quite some overhead in terms of latency, translation tables, and power consumption.

Therefore we allow currently only a few of the possible combinations, listed in Table 1.3. Private memory can only be local and with a physical address. In contrast, shared memory requires a virtual address. This policy limits the number of possible cases and allows for a high performance, low overhead implementation. Access to local private memory requires no virtual address translation and is therefore processed in one cycle, essentially passing through the DME from the CPU to local memory. Shared memory access requires a virtual-to-physical address translation and can be local or remote. The current DME implementation performs a virtual-to-physical address translation in 11 cycles.

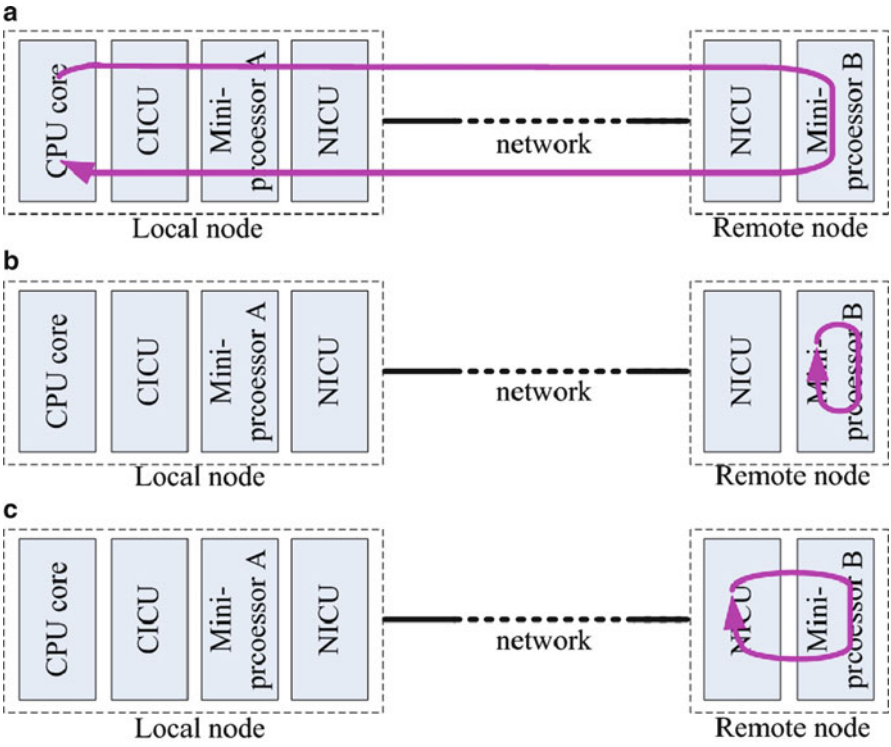

Fig. 1.8 (**a**) *test-and-set()*; (**b**) the conventional blocking *test-and-set()*; (**c**) our proposed *test-and-set-blocking()*

1.4.3 Synchronization

The Synchronization Supporter in the DME provides the hardware support for memory synchronization. It works with microoperations **ll** and **sc** to guarantee atomic read-and-modify operations on mutex locks. Based on them, higher level synchronization mechanisms can be built.

We have implemented two synchronization primitives: *test-and-set()*, *test-and-set-blocking()*. *test-and-set()* is non-blocking and the programmer may use it to implement a spin lock. However it incurs additional network traffic as the program re-spins the lock, as illustrated in Fig. 1.8a. *test-and-set-blocking()* Fig. 1.8c is an improvement over the conventional spin-lock based blocking *test-and-set()* [17] (Fig. 1.8b). With the conventional blocking *test-and-set()*, the mini-processor B in the remote node would continue to spin the lock until success while executing the microcode. This does not incur additional network traffic, but other requests from other nodes will be blocked as the mini-processor B continuously polls the lock. Our proposed *test-and-set-blocking()* utilizes the cooperation of the mini-processor

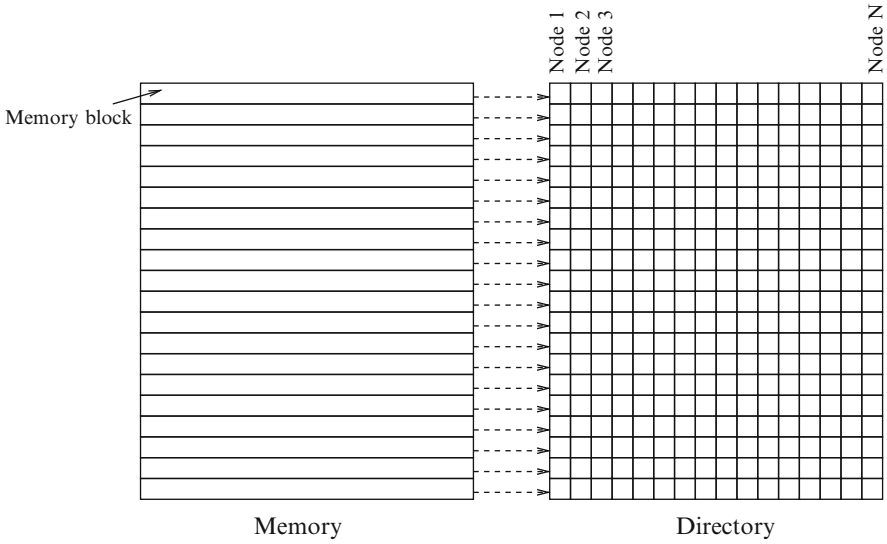

Fig. 1.9 The directory contains an entry for each block of shared memory

B and the NICU. If an acquire of the lock fails, the related command will be placed to the tail of the command queue to wait for the next execution. This avoids incurring additional network traffic and will not block other commands.

1.4.4 Cache Coherency

One of the more challenging problems in a distributed shared memory system is to realize a scalable, high performance, low overhead cache coherence scheme. Snooping based protocols do not scale well with interconnects where broadcasting transactions to all members in the system is prohibitively expensive. Directory based protocols scale relatively well in terms of performance but care has to be taken to avoid large memory overhead due to the directory (see [12, 16] for general discussions of the topic).

As motivated above, we expect the application programmer to keep the shared memory space small. Therefore, we have, in a first stage, implemented a classic and simple directory based cache coherence protocol. The memory overhead due to the directory is modest if the size of the shared memory space is kept within reasonable bounds.

The directory maintains status information about each block of shared memory (Fig. 1.9). A block in main memory corresponds to a line in the cache. When a node has a shared memory partition, a fraction of the memory has to be sacrificed for the directory. For each block of memory there is one entry in the directory, which contains one bit for each node in the network. This bit is 1, when the corresponding

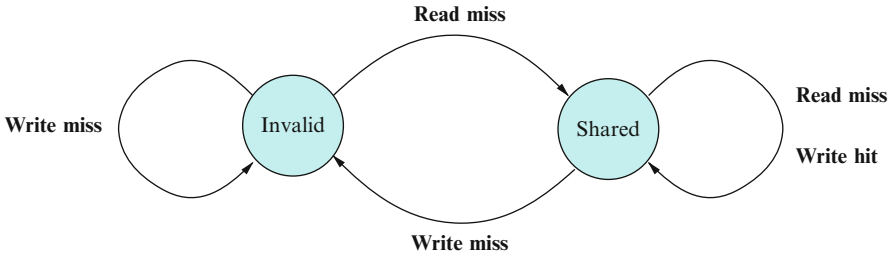

Fig. 1.10 State transition diagram of the SI cache coherence protocol

node holds a copy of the block in its local cache (the node is a sharer); 0 otherwise. Thus the size of the directory is $N \times M$ bit, where N is the number of nodes in the network, and M is the number of blocks in the shared memory space.

We use a simple, write-through, no-allocate policy with a Shared-Invalidate (SI) protocol [13]. On a write, data is always written to main memory which is thus kept up to date. On a write-miss, the date is not written to the local cache (no-allocate). With this policy no node is ever the exclusive owner of a memory block and the corresponding cache coherence protocol can be kept simple (Fig. 1.10).

This scheme has several advantages: (1) it is easy to implement; (2) the main memory has always the most recent value; (3) a read miss results never in a write to memory. On the downside we have: (1) write is slow, because it always goes to main memory; (2) every write results in a memory transaction even if the same node writes many times to the same location in sequence; (3) more memory transactions appear on the network which has to be able to cope with this load.

We implement several routines on the DME to operate the protocol. These routines realize the directory control and the memory access. The routines generate messages according to the protocol and change the directory states.

Here, three kinds of nodes are involved. The *Home Node* is the node which hosts the target memory and the related directory. The *Local Node* is the node which issues the read/write request. Finally, the *Remote Node* is the node which also maintains the data copy in its cache. When there is a read/write miss in the processor's cache, a read/write request is sent to its DME. If the address is in local, shared space, the request is processed by its own DME. If it is a remote, shared address, the Local Node sends the request to the Home Node. Once the request arrives at the Home Node, the read/write routine for cache coherence is triggered.

For read request, the Home Node always keeps the up-to-date data. The data is returned to the Local Node directly. Figure 1.11a shows the read procedure. In a write request (Fig. 1.11b), there may be one or more Remote Nodes that also have the data in their local caches. So the Home Node needs to send invalidation requests to all these Remote Nodes. And once all the invalidation acknowledgements have returned, the Home Node grants the Local Node the right to update the data.

Although this scheme is relatively simple and is not effective in all cases, it can offer significant performance improvements and good scalability properties

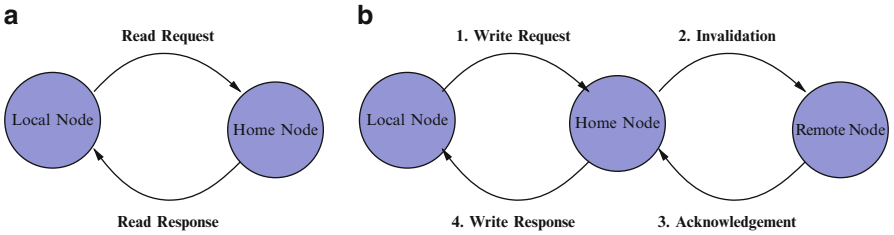

Fig. 1.11 The read procedure (**a**) and write procedure (**b**) in the cache coherence protocol

for applications with a modest amount of shared memory and fairly localized communication and memory access patterns. As future work we plan to realize a hierarchical cache coherence scheme along the lines described in [34].

1.4.5 Memory Consistency

The memory consistency model is a contract between the programmer and shared memory parallel system. The programmer follows the rules that are guaranteed by the parallel system to get the predictable results of the shared memory operations. The shared memory access latency can be reduced by the hardware and software optimizations in the system architecture. These performance optimizations can reorder the shared memory operations and the system may give unexpected results. For the expected results, these reordering should be controlled carefully. Different memory consistency models enforce different ordering constraints on the shared memory operations [3, 12]. Here we focus on some of the memory consistency models that are realized in the DME based multi-core systems.

1.4.5.1 Sequential Consistency

The sequential consistency (often called Strong Ordering) defined by Lamport [21] has to maintain the program order among operations of each individual processor and sequential order among multiple processors in the system. It is a strict model and does not allow reordering between shared memory operations in the multi-core systems. A memory operation (read or write) cannot be reordered or overlapped with the following memory operation to the different locations in the shared memory. The shared memory operations are completed according to the program order as shown in Fig. 1.12a. The sequential consistency enforces the global orders on shared memory operations as given in Fig. 1.12b.

The sequential memory consistency model in the DME based multi-core NoC platform is realized by stalling the processor on the issuance of a shared memory operation till the completion of the preceding operations [27]. Then, the processor

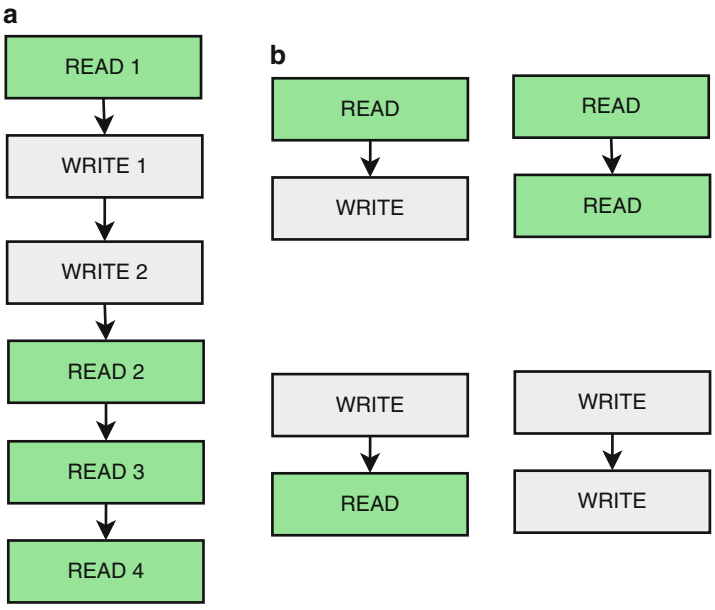

Fig. 1.12 (**a**) Strong ordering; (**b**) Global orders to enforce

issues the next operation. The completion of preceding operation is indicated by the return data or acknowledgment. The program order is maintained due to the strict order between the shared memory operations and sequential order is maintained by read-modify-write memory operation in the system.

The sequential memory consistency model does not allow the performance optimizations [3] in the hardware (write buffer, cache, interconnection network) and in the software (compiler reordering, register allocation) due to the strict order enforcement on the shared memory operations. As a result relaxed memory consistency models emerged that permit such optimizations. The relaxed consistency models relax the program order requirement to allow the possible reordering in the shared memory operations by the system optimizations. The shared memory operations may not complete according to the program order. The overall program correctness is ensured by enforcing ordering constraints on a subset of memory operations. We consider the two relaxed consistency models (weak and release consistency) which relax the strict program order requirement and allow reordering in the shared memory operations.

1.4.5.2 Weak Consistency

The weak consistency model (also called weak ordering) was proposed by Dubois et al. [14] which classifies shared memory operations as *synchronization* and *data*

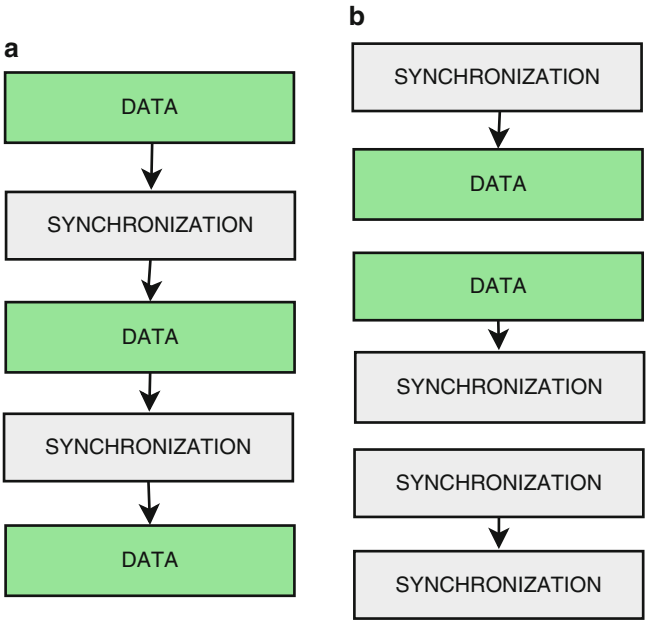

Fig. 1.13 (**a**) Weak Ordering; (**b**) Global orders to enforce

operations. Synchronization operations are related to the special synchronization variables (locks, semaphores) in the shared address space. The lock must be gained exclusively in the multi-processor shared memory systems. Data operations are the load-store operations related to the ordinary shared variables. According to the weak consistency model all previously issued outstanding data operations must be completed before the issuance of synchronization operation. Similarly, previously issued outstanding synchronization operations must also be completed before the issuance of any data operation as depicted in Fig. 1.13a. The weak consistency model ensures the final consistent result of the program execution in the multi-processor systems. The weak consistency model enforces some global orders on the shared memory operations as shown in Fig. 1.13b. The enforcement of these global orders on the shared memory operations ensures the program correctness in the weak consistency model with the permitted relaxation in data operations.

The transaction counter (TC) based approach [3, 27, 28] is adapted for the realization of the weak memory consistency model in the DME based multi-core systems. The counter is implemented in the DME hardware in each node to keep track of the outstanding data operations issued between the two synchronization points. It is incremented and decremented by the issuance and completion of data operations correspondingly. It is not affected by the synchronization operations.

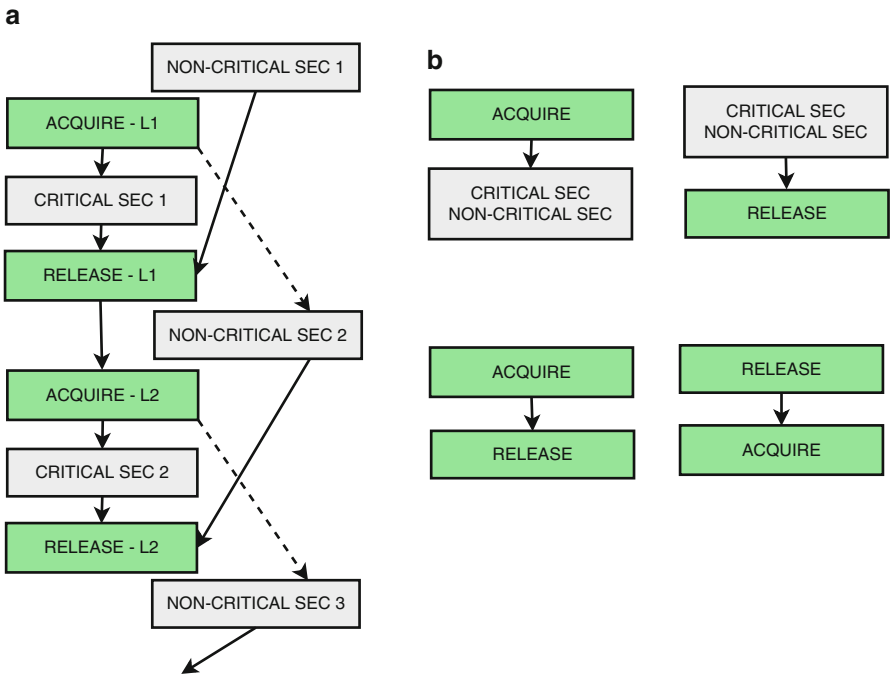

Fig. 1.14 (**a**) Release consistency model; (**b**) Global orders to enforce

1.4.5.3 Release Consistency

The release consistency model proposed by Gharachorloo et al. [15] is a refinement of the weak consistency and distinguishes synchronization operations as *acquire* and *release* operations. It removes the following two unnecessary ordering constraints and allows further relaxation in the program order as compared to the weak consistency model:

- Non-critical section data to synchronization
- Synchronization to non-critical section data

According to the release consistency model, an acquire operation must be performed before the issuance of any data operation in the critical section (CS) and in the non-critical section (NCS) after it. All the data operations in the critical and non-critical sections prior to the release operation must be completed before the issuance of the release operation as shown in Fig. 1.14a. The release consistency model enforces the global orders on the shared memory operations as given in Fig. 1.14b.

The release memory consistency model can be realized by using two transaction counters in the hardware of DME in each node of the platform [26]. Two counters in each node correspond to two types of data operations. The transaction counter 1 (TC1) keeps track of outstanding data operations issued in the non-critical section of

code by the processor. The transaction counter 2 (TC2) keeps track of outstanding data operations issued within the critical section. Each counter is incremented and decremented by the issuance and completion of the relevant data operations correspondingly. Both these counters are not affected by the acquire and release synchronization operations. "TC1 = 0" indicates the completion of all previously issued outstanding data operations in the non-critical section of code. "TC2 = 0" indicates the completion of all the previously issued outstanding data operations in the critical section of code. The lock acquire operations do not check the counters at the issuance time, while the release operations are not issued by the processor until both these counters become zero.

1.5 Experiments

We have studied the DME behavior in a number of different experiments. In the following we describe some of these experiments to show, that the usage of the DME is feasible (Sect. 1.5.1), how the DME can cleverly be used to maximize performance (Sect. 1.5.2), and the performance under synchronization constraints (Sect. 1.5.3).

1.5.1 Viability Analysis

We map three applications, matrix multiplication, 2D radix-2 DIT FFT, and Wavefront Computation, manually over the LEON3 processors, based on our proposed hardware/software co-design flow. The matrix multiplication calculates the product of two matrices, $A[64,1]$ and $B[1,64]$, resulting in a $C[64,64]$ matrix and does not involve synchronization. We consider both integer and floating point matrix multiplication.

The data of the 2D radix-2 DIT FFT are equally partitioned into n rows stored on n nodes respectively. The 2D FFT application performs 1D FFT of all rows firstly and then does 1D FFT of all columns. There is a synchronization point between the FFT-on-rows and the following FFT-on-columns.

Wavefront Computations are common in scientific applications. In Wavefront Computation, the computation of each matrix element depends on its neighbors to the left, above, and above-left. If the solution is computed in parallel, the computation at any instant forms a wavefront propagating through the processor array.

Figure 1.15 shows the performance speedup of the three applications. We observe, that the multi-core NoC achieves fairly good speedup. When the system size increases, the speedup increases almost linearly. The speedup Ω_{m} is defined as $\Omega_{\mathrm{m}} = \mathrm{T}_{1\mathrm{core}}/\mathrm{T}_{\mathrm{mcore}}$, where $\mathrm{T}_{1\mathrm{core}}$ is the single core execution time as the baseline, $\mathrm{T}_{\mathrm{mcore}}$ the execution time of m cores.

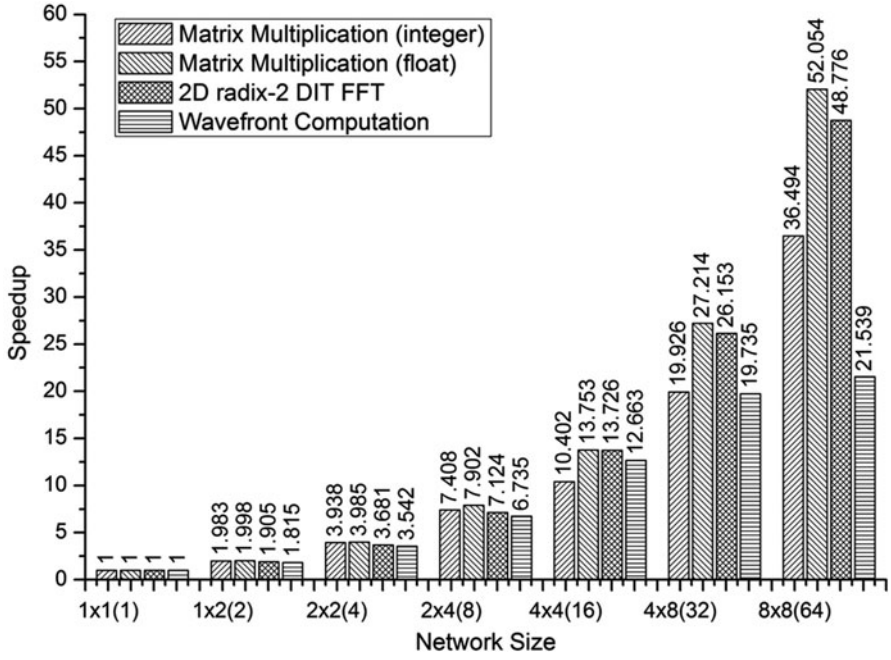

Fig. 1.15 Speedup of matrix multiplication, 2D radix-2 DIT FFT and wavefront computation

For instance, the speedup for the matrix multiplication on the 8×8 system is 52.054, which is very close to the ideal speedup of 64. However, as the system size increases, the speedup acceleration slows down. This is due to the growing communication latency which increases linearly with the system size, limiting the performance.

Note, that the speedup for the floating point matrix multiplication is higher than that for the integer matrix multiplication, because, when increasing the computation time, the portion of communication delay becomes less significant, thus achieving higher speedup.

Wavefront Computation is synchronization-intensive. Its speedup increases more slowly than Matrix Multiplication and 2D DIT FFT, because synchronization overhead and communication delay become dominating as the network size is scaled up.

Obviously, the speedup figures strongly depend on the application characteristics and on the task partitioning and mapping. We could achieve very good speedup results, because the four applications are inherently easy to parallelize and have been mapped well. This is not true for many other applications. But the experiments demonstrate, that, given applications that can be parallelized and mapped well onto a multi-core platform, the DME, handling the memory access and communication, is not limiting the speedup and does not constitute a bottleneck for system performance.

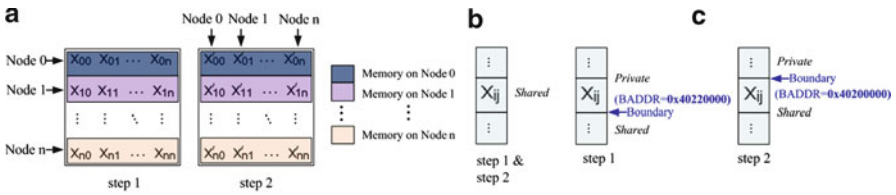

Fig. 1.16 (**a**) Memory allocation for 2D DIT FFT, (**b**) conventional DSM organization, and (**c**) Hybrid DSM organization with run-time partitioning

1.5.2 Performance Optimization

In the following experiment we show, how system performance can be improved by exploiting specific features and the flexibility of the DME.

We implement a 2D radix-2 DIT FFT. As shown in Fig. 1.16a, the FFT data are equally partitioned into n rows, which are stored on the n PM nodes, respectively. According to the 2D FFT algorithm, the application first performs FFT on rows (step 1). After all nodes finish the row FFT (synchronization point), the FFT on columns are started (step 2).

We experiment with two DSM (Distributed Shared Memory) organizations. One is the conventional DSM organization, as shown in Fig. 1.16b, for which all FFT data are shared. The other is the hybrid DSM organization, as illustrated in Fig. 1.16c. The data used for row FFT calculations at step 1 are located locally in each PM node. After step 1, they are updated and their new values are to be used for column FFT calculations at step 2. We can dynamically re-configure the boundary address (BADDR in Fig. 1.16) at run time, such that, the data are *private* at step 1 but become *shared* at step 2.

Figure 1.17 shows the speedup of the FFT application with the conventional DSM organization, and performance enhancement of the hybrid DSM organization with run-time partitioning. As we can see, when the system size increases from 1 to 64, the speedup with conventional DSM organization goes up from 1 to 48.776, and the speedup with hybrid DSM organization improves from 1.525 to 55.070.

For the different system sizes the improvement of the hybrid DSM organization is between 11.44% and 34.42%.

We can summarize the experiment as follows:

- Using run-time partitioning in hybrid DSM organization, a fast *physical addressing scheme* is performed in step 1 of the 2D DIT FFT and the entire virtual address translation overhead is avoided.
- As the system size is scaled up, larger communication delay leads to the decrease of performance improvement.
- The single PM node case has a higher improvement because all data accesses are local and shared for the conventional DSM organization and private for the hybrid

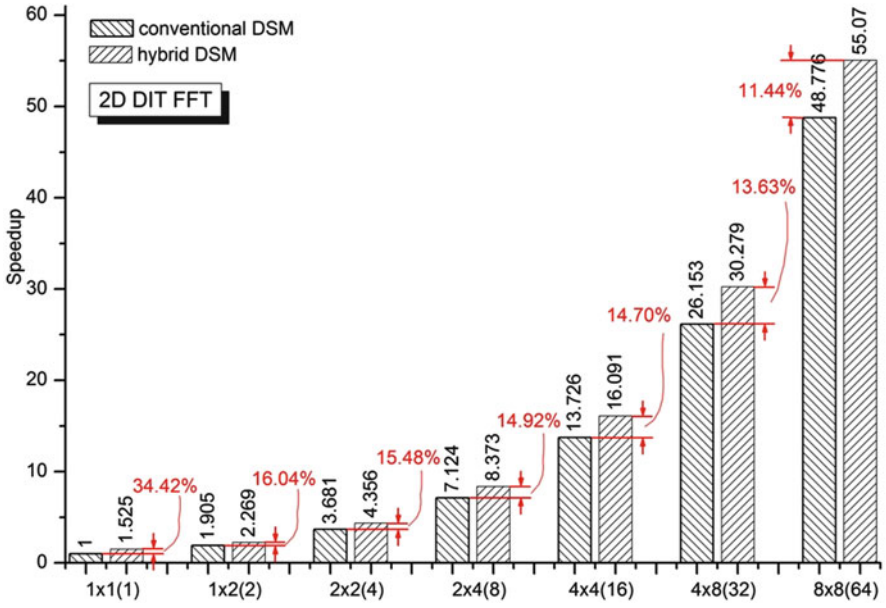

Fig. 1.17 Speedup and performance improvement of 2D DIT FFT

DSM organization, and there is no synchronization overhead. Thus, the single PM case also shows the bookkeeping and translation overhead of managing a shared memory space.

1.5.3 Experiments for Memory Consistency

We analyze the performance of the three consistency models that are realized in the multi-core system. The effects of network size on average and maximum code latencies are investigated. As traffic pattern we use a set of synthetic workloads running on each node in the platform. The sequence of transactions is the same for each node and is shown in Fig. 1.18. Two nodes constitute hotspots: one for the critical section (CS-node) and the other for the lock (SYNC-node). Each node sends synchronization requests to the SYNC-node. The shared memory locations in the CS-node are protected by the acquire and release operations to the lock in the SYNC-node.

The average and maximum code execution time, which we call *code latency*, for different network sizes is shown in Fig. 1.19. The code latency increases for all the three consistency models as the network grows from a single core to 64 cores. The average code latency for the release consistency model in the 8×8 network is about 236.59 times of the single core, whereas for the weak and sequential

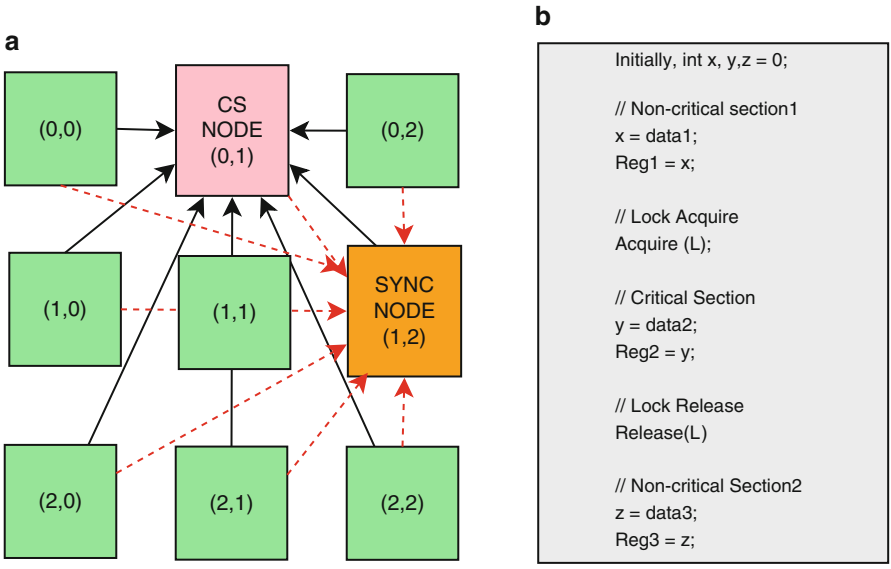

Fig. 1.18 Each node generates the same sequence of transactions

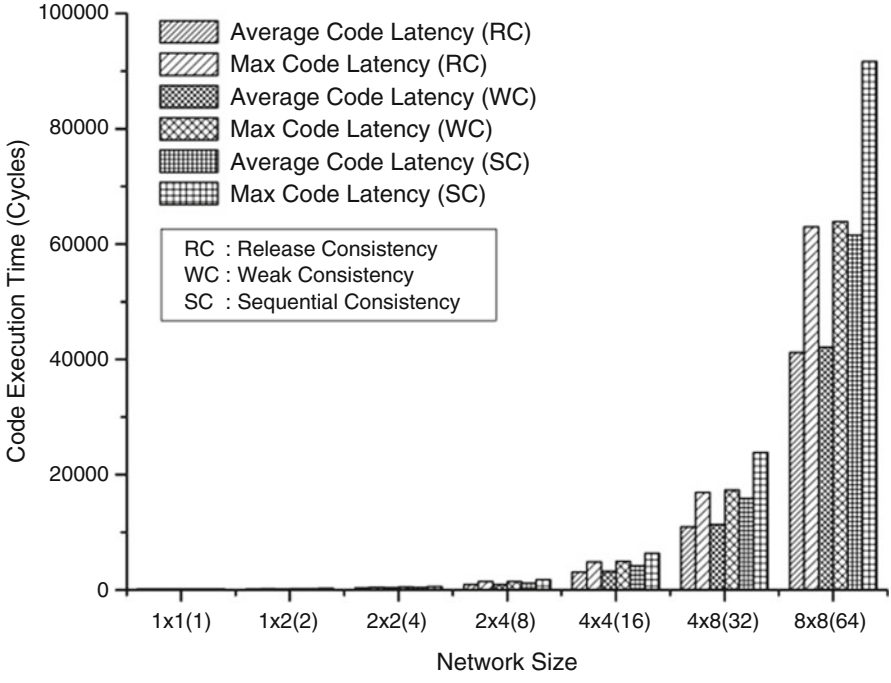

Fig. 1.19 Execution times of a test program on different architectures

consistency models it is 241.96 and 353.67 times, respectively. The difference between the observed code latencies become obvious as the network size grows. It is due to the further overlapping and program order relaxation in the release consistency as compared to the weak consistency model. The sequential consistency does not allow any kind of reordering in the shared memory operations. The weak consistency model allows the reordering among the shared memory operations in the critical section or non-critical section. The release consistency model permits overlapping among the data operations in the critical and non-critical sections. The release consistency improves the performance by 2.3% and 49.5% on average in the code latencies over the weak and sequential consistency models, respectively, as the system grows from a single core to 64 cores.

1.6 Conclusion

Due to the pace of technology development, computation and communication is operated more and more in parallel. Indeed, the degree of parallelism grows at the rate of Moore's law. As a consequence, memory access must follow suit and become more parallel. 3D stacking and other emerging technology facilitates this trend, but has to be matched by a corresponding adaptation in the memory architecture.

We propose a programmable hardware block, called a Data Management Engine, that supports this architectural adaptation in multiple ways. The DME can realize an address space partitioning into private-shared and into local-remote sections. While the local and remote memory is determined by the platform at design time, the partitioning into private and shared sections can be flexibly modified at run-time, and can be used for performance optimizations. Among the many other potential DME applications we present virtual address space management, synchronization, cache coherency, and memory consistency. We also demonstrate the feasibility of the DME in several experiments.

References

1. The Aeroflex Gaisler webpage. http://www.gaisler.com/.
2. Z-RAM technology backgrounder. http://www.innovativesilicon.com/en/pdf/z-ram.pdf.
3. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.
4. Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society.
5. Xiaowen Chen, Shuming Chen, Zhonghai Lu, and Axel Jantsch. Area and performance optimization of barrier synchronization on multi-core network-on-chips. In *3rd IEEE International Conference on Computer and Electrical Engineering (ICCEE)*, Chengdu, China, November 2010.

6. Xiaowen Chen, Zhonghai Lu, Shuming Chen, and Axel Jantsch. Run-time partitioning of hybrid distributed shared memory on multi-core network-on-chips. In *The 3rd IEEE International Symposium on Parallel Architectures, Algorithms and Programming (PAAP 2010)*, Dalian, China, December 2010.
7. Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Handling shared variable synchronization in multi-core network-on-chip with distributed memory. In *International SOC Conference*, Las Vegas, Nevada, September 2010.
8. Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Proceedings of the confernece for Design Automation and Test in Europe*, Dresden, Germany, March 2010.
9. Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Supporting efficient synchronization in multi-core NoCs using dynamic buffer allocation technique. In *Proceedings of the IEEE Annual Symposium on VLSI*, Kefalonia, Greece, July 2010.
10. Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.
11. Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. *SIGARCH Comput. Archit. News*, 33(2):357–368, 2005.
12. David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufman Publishers, 1999.
13. Pierre Guironnet de Massas and Frédéric Pétrot. Comparison of memory write policies for NoC based multicore cache coherent systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 997–1002, New York, NY, USA, 2008. ACM.
14. Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.
15. K. Gharachorloo, D. Lenoski, J. Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Computer Architecture News*, 18(2):15–26, June 1990.
16. J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proceedings of the IEEE*, 87(3):418 –429, March 1999.
17. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.
18. Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM.
19. Axel Jantsch, Matthew Grange, and Dinesh Pamunuwa. The promises and limitations of 3-D integration. In Abbas Sheibanyrad, Frédéric Pétrot, and Axel Jantsch, editors, *3D Integartion for NoC-based SoC Architectures*, Integrated Circuits and Systems, chapter 2. Springer, 2011.
20. C. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 10 2002.
21. L. Lamport. How to make a multiprocessors computer that correctly executes multiprocessors programs. *IEEE Transaction on Computers*, C-28(9):690–691, September 1979.
22. Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and management of 3 D chip multiprocessors using network-in-memory. *ACM SIGARCH Computer Architecture News*, 34(2):130–141, 2006.
23. C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *Design and Test of Computers*, 22(6):556–564, November-December 2005.

24. Gabriel Loh. 3D-stacked memory architectures for multi-core processors. In *Proceedings for the 35th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2008.
25. G. L. Loi, B. Agarwal, N. Srivastava, S.-C. Lin, and T. Sherwood. A thermally-aware performance analysis of vertically integrated 3-D processor memory hierarchy. In *Proceedings of the 43rd Desigfn Automation Conference*, 2006.
26. Abdul Naeem, Xiaowen Chen, Zhonghai Lu, and Axel Jantsch. Scalability of transaction counter based relaxed consistency models in NoC based multicore architectures. *ACM SIGARCH Computer Architecture News*, December 2009.
27. Abdul Naeem, Xiaowen Chen, Zhonghai Lu, and Axel Jantsch. Scalability of weak consistency in NoC based multicore architectures. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, Paris, France, June 2010.
28. Abdul Naeem, Xiaowen Chen, Zhonghai Lu, and Axel Jantsch. Realization and performance comparison of sequential and weak memory consistency models in network-on-chip based multi-core systems. In *Proceedings of the 16th Asian Pacific Design Automation Conference (ASP-DAC)*, Tokyo, Japan, January 2011.
29. T.G. Rauscher and P.M. Adams. Microprogramming: A tutorial and survey of recent developments. *Computers, IEEE Transactions on*, C-29(1):2 –20, January 1980.
30. Chuan Seng Tan. Three-dimensional integration of integrated circuits - and introduction. In Abbas Sheibanyrad, Frédéric Pétrot, and Axel Jantsch, editors, *3D Integartion for NoC-based SoC Architectures*, Integrated Circuits and Systems, chapter 1. Springer, 2011.
31. R. Stanley Williams. How we found the missing memristor. *IEEE Spectrum*, December 2008.
32. W. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.
33. Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.
34. Yuang Zhang, Zhonghai Lu, Axel Jantsch, Li Li, and Minglun Gao. Towards hierarchical cluster based cache coherence for large-scale network-on-chip. In *Proceedings of the 4th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, Cairo, Egypt, April 2009.

Chapter 2
Application-Specific Multi-Threaded Dynamic Memory Management

Sotirios Xydis, Alexandros Bartzas, Iraklis Anagnostopoulos, and Dimitrios Soudris

Abstract This chapter presents the methodology and the corresponding software framework developed to systematically explore the design space of Multi-Threaded Dynamic Memory Management (MTh-DMM). We developed two exploration approaches: (a) a two-phase constraint-orthogonal and (b) a single-phase aggressive exploration methodologies. Pareto optimal configurations are generated considering various design targets. Experimental results evaluate the solution quality delivered by the proposed exploration approaches and by state-of-the art general purpose dynamic memory management solutions, based on a real-life multi-threaded network application.

2.1 Introduction

Recent advances in VLSI process technology enabled the shifting from single-processor System-on-Chip (SoC) to Multi-Processor SoC (MPSoC) architectures. The emerging market of new embedded devices seems to highly adopt the new architectural template [1, 19] to integrate multiple services and heterogeneous applications such as multimedia, telecommunication protocols and wireless network communications. The increased number of processing elements enables the exploitation of parallelism in coarser levels (thread-level) than the Instruction-Level-Parallelism (ILP) found in uni-processor SoCs. Thus, multi-threaded applications are becoming increasingly prevalent for the next generation of embedded systems.

The porting process of multi-threaded applications to MPSoCs is a difficult task. They come from the general-purpose domain require increased interaction with the

S. Xydis • A. Bartzas (✉) • I. Anagnostopoulos • D. Soudris
National Technical University of Athens, 9 Heroon Polytechneiou,
Zographou Campus, Greece
e-mail: sxydis@microlab.ntua.gr; alexis@microlab.ntua.gr; iraklis@microlab.ntua.gr;
dsoudris@microlab.ntua.gr

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_2,

environment, which raises their dynamism [14]. Multiple algorithms that need to run concurrently on such devices impose complex memory access patterns that may result in performance degradation and high energy consumption. For example, in multimedia applications like video games, the unpredicted behavior of the user causes varying inputs, which significantly vary the dynamically allocated memory objects, leading to unexpected memory footprint variations unknown until runtime.

Dynamic memory management (DMM) is a critical component in MPSoCs since the dynamic memory allocation often forms the main performance, and scalability bottleneck of multi-threaded applications [15]. Also, it greatly affects the energy and memory consumption of the overall system [8]. Extensive research has been conducted for general-purpose dynamic memory allocators, which target either the single processor or the multi-processor domain. However, embedded computing involves stricter design constraints than general purpose one due to the limited available resources. Thus, there is a need for application-specific customization of the overall system's management since usually there is a-priori knowledge on the applications' set or the application domain on which the embedded system has to be operative. Customizing the decisions concerning the strategies, policies and architecture of the dynamic memory manager improves performance [5, 9], heavily optimizes power consumption compared to general-purpose dynamic memory managers [8] and avoids memory fragmentation [10]. However, the specification of the available DMM design space (DMM design decisions) and a corresponding methodology for combining these decisions has been elaborated only for single-processor systems running single-threaded applications [2]. Thus, it is reported a semantic gap considering the extended design space of Multi-Threaded DMM (MTh-DMM) and the proper methodologies for designing efficient custom dynamic memory managers for MPSoCs.

In this chapter, we address the open issue of application-specific multi-threaded dynamic memory management by proposing a comprehensive methodology which enables the designer to explore, traverse and evaluate through the decisions of the new design space in an efficient manner. The exploration methodology searches and evaluates the available decisions of the DMM design space and returns decision combinations. Given the multi-threaded application to be executed onto an MPSoC, the proposed exploration methodology delivers a custom dynamic memory manager based on the proper combination of decisions found into the extended DMM design space. Such decisions made both at the intra-thread level and the inter-thread level in order to customize the dynamic memory manager according to both each thread atomic allocation behavior and their existing interaction.

2.2 Heap-Based Dynamic Memory Management

The memory pool responsible for allocation or deallocation of arbitrarily-sized blocks in arbitrary order that will live an arbitrary amount of time is called heap. In order to conceptually position the heap region, Fig. 2.1 depicts a typical memory

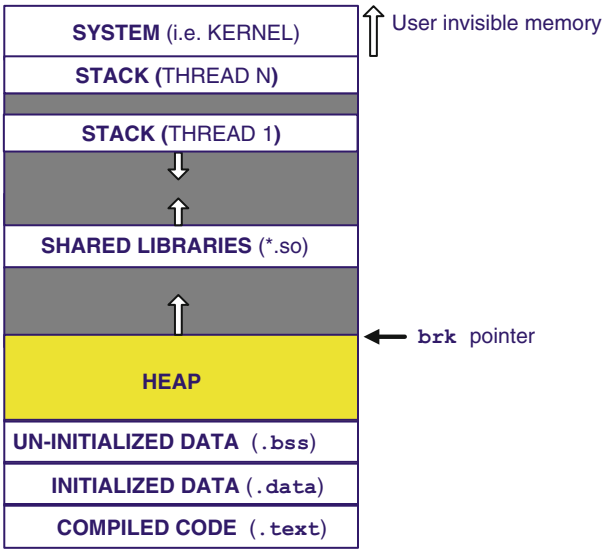

Fig. 2.1 Memory layout of process address space

layout of a running process, assuming a C application running onto a Linux-OS (the organization of process address space is determined by the Operating System (OS) and the programming language). In brief, the ".text" segment stores the compiled code of the program that forms the running process, the ".data" segment contains global and static variables used by the program that are initialized. while the ".bss" segment contains all the uninitialized global variables and static variables that are initialized to zero by default. The stack is the section of memory allocated for variables within functions or temporary storage of information, using a Last In First Out (LIFO) scheme.

Between the stack and the ".bss" segment, it lays the heap memory pool. The heap is the memory region that the dynamically allocated data objects are stored. Since the size of the allocated data is unknown until the runtime, the heap is managed by dividing it into blocks able to service the runtime memory request. The unallocated (free) blocks are organized based on dynamic data structures (i.e. single linked lists, double link lists, trees, etc.), usually called free-lists. In many cases several free-lists exist inside the heap address space. During an allocation request the dynamic memory manager searches the free-lists in order to find an available free block to return. In order to reduce the searching overhead, a common practice is to manage free-lists that handle memory blocks of a specific size, called fix-sized free-lists or fixed-lists. By this way the searching is reduced since the manager knows where to search first. Fixed-lists can be viewed as a caching mechanism of the dynamic memory manager. The size of the heap during the execution is defined by the brk pointer. Thus, each time the application requests for memory allocation,

the dynamic memory manager either returns a pointer to an unused block found into the heap or requests for additional heap memory from the operating system using the sbrk function (actually requests for moving the brk pointer).

Dynamic memory managers are operate on the heap memory space, being responsible for organizing the dynamically allocated data into the heap and servicing the application's memory requests (allocation/deallocation) at run-time. In case of a memory request for allocation of a new object the dynamic memory manager returns to the application the pointer, which refers to memory position of allocated object. In case of a memory request for deallocation of an already dynamically allocated object, the dynamic memory manager returns to the application either a true or a false value in respect to success of the deallocation process. In C/C++ programming language dynamic memory management is performed through the malloc/new functions for allocation and free/delete functions for deallocation, respectively.

2.3 MTh-DMM Performance Metrics

Historically, the efficiency of single-threaded dynamic memory managers was evaluated according to conventional metrics such as performance and memory fragmentation (internal and external). However, in the field of multi-threaded dynamic memory managers both conventional and new defined metrics are required in order to perform accurate evaluation of their efficiency [4]. That is because of new design constraints are taken into account for the case of multi-threaded applications i.e. scalability of the solution and avoidance of false sharing. Furthermore, design constraints that come from the embedded computing community (i.e. energy consumption) have to also be taken into account. Thus, in order to design efficient multi-threaded dynamic memory allocator of the following metrics should have to be considered:

1. Performance: A multi-threaded dynamic memory allocator should perform memory operations (i.e., malloc and free) about as fast as a state-of-the-art serial memory allocator. By this way, performance is guaranteed even when a multi-threaded program executes on a single-processor. While in single-threaded applications performance is affected mainly by the fit and search policies of the allocator, in multi-threaded applications also the synchronization mechanisms and strategies have a great impact in the allocator's performance.
2. Scalability: A scalable multi-threaded dynamic memory allocator should guarantee that as the number of processors in the system grows, the performance of the allocator also grows/scale linearly with the number of processors to ensure scalable application performance. The un-scalable behavior of the memory allocator makes it the bottleneck of the overall application.
3. Heap False Sharing: Heap false sharing refers to the situation in which threads with distinct heaps inadvertently share data. For example, data allocated from

thread 1 into heap 1 are freed from of thread 2 into heap 2. In many access patterns such a situation may impose extensive memory consumption. Considering a platform dependent view, heap corruption resembles the false sharing of cache lines.

4. Memory Fragmentation: In general fragmentation is defined as the maximum amount of memory allocated from the operating system divided by the maximum amount of memory required by the application. In multi-threaded memory allocators there are three types of fragmentation:
   (a) Internal Fragmentation: It happens when the allocator returns a memory block that is larger than the initial size request.
   (b) External Fragmentation: It happens when a memory request cannot be served even if there are available memory blocks that can serve the request if they merged.
   (c) Memory Blowup: This is a special kind of fragmentation found in multi-threaded memory allocators. Specifically, it refers to the situation in which the increase in memory consumption caused when a concurrent allocator reclaims memory freed by the program but fails to use it to satisfy future memory requests. It is defined as the maximum amount of memory allocated by a given allocator divided by the maximum amount of memory allocated by an ideal uni-processor allocator [4].
   (d) Energy Consumption: This design metric comes from the field of embedded computing in which battery lifetime forms a critical resource. Energy consumption for DMM has a direct relation with the number of memory accesses that the allocator performs to service a memory request.

2.4 Related Literature

This section presents a brief classification of the various MTh-DMM solutions [4] based on the organization of the heap memory pool.[1] Each multi-threaded allocator[2] presented in the existing literature can be assigned to one of these classes. Figure 2.2 illustrates in an abstract manner the various heap organizations that each of the dynamic memory management classes imposes. Grey areas depict the heap regions that can be accessed in parallel from each thread. The following sections further elaborated on each of the aforementioned multi-threaded allocator classes and reference relative work and research activities.

[1] A similar taxonomy for single-threaded DMM can be found in the Wilson's extensive report [25].
[2] In the reminder, we use interchangeably the terms allocator and dynamic memory manager.

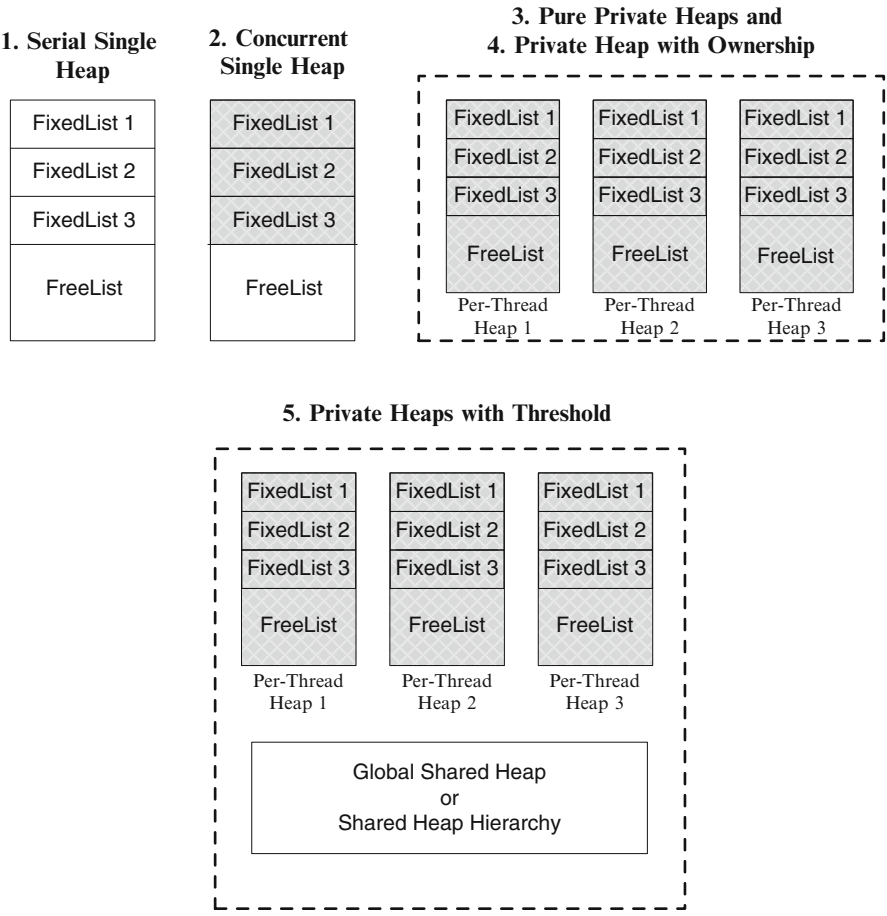

Fig. 2.2 Heap organization schemes for MTh-DMM

2.4.1 Class A: Single Heap MTh-DMM

A.1 Serial Single Heap MTh-DMM assumes a global shared heap, protected through locking before each (de)allocation request. Serialization of memory operations and heavy lock contention are introduced, forming a serious bottleneck in the case of multi-threaded applications. Since all the threads operate on the same heap, false sharing is heavily induced. Many operating systems provide such type of memory allocators in their default library [15]. Existing customization methodologies [2, 5] can be applied in straightforward manner in this class of allocators.

A.2 Concurrent Single Heap Allocators implement the heap as a concurrent data structure, such as a concurrent B-tree [13, 21] or a freelist with locks either on each free block or on the entire freelist [6, 7]. In the general case of random access patterns, they present better scalability and performance characteristics than the serial single heap allocators. The extensive use of locks makes them quite expensive considering the energy and performance. Each synchronization operation requires one or more memory access, so their extensive use contributes in a negative way to overall energy consumption. Customization is expected to have a positive impact in both performance and energy metrics of such allocators.

2.4.2 Class B: Multiple Heap MTh-DMM

B.1 Pure Private Heaps Allocators assume that each thread has its own private heap used for every dynamic memory operation and never accesses any other [11, 18]. In terms of performance and scalability, pure private heaps form a very efficient solution since in an ideal situation minimum lock contention is induced. However, if the application exhibits producer-consumer allocation patterns between different threads, memory consumption is unbounded [4]. Thus, exploration and customization is required to evaluate the efficacy of pure private heaps DMM solutions in an application specific basis.

B.2 Private Heaps Allocators with Ownership extends the pure private heaps class with heap ownership mechanisms [17, 20, 24]. False heap sharing is eliminated, since threads deallocate memory to the heap that originally allocated it. Each thread or group of threads operates on its own heap and the lock contention is small delivering efficient performance and scalability. In an application-specific context, the ownership mechanisms and the various thread groupings (thread-to-heap mapping mechanisms) introduce new trade-off parameters that have to be explored for optimized custom MTh-DMM.

B.3 Private Heaps Allocators with Threshold move blocks of memory between a hierarchy of heaps shared by multiple threads [4, 16, 23]. When a private heap has more than a certain amount of free memory (crossing a threshold value), some portion of the free memory is moved to a shared heap. This strategy bounds memory blowup to a constant factor, since no heap may hold more than some fixed amount of free memory [4]. However, the threshold value need to be carefully determined since a rather small value will initiate a lot of memory movement degrading performance, while a rather large threshold may induce large memory waste. Such trade-offs can be efficiently utilized through customization of the allocator according to the application specific needs.

2.5 A Modular C++ Library for Application-Specific MTh-DMM

In order to design application specific dynamic memory manager a number of decisions and strategies have to been explored. Each decision forms a different implementation choice. Different combination of decisions delivers different dynamic memory managers with different tailoress to the application specific needs. Thus, the specification of all the possible decisions and strategies concerning dynamic memory allocation has to be defined in order to be able to explore various alternatives.

The enumeration of these decisions defines the complete design space of dynamic memory management. All of them should affect as less as possible the other ones, i.e. be as orthogonal as possible. Thus, this set of possible decisions must cover exhaustively any kind of potentially profitable (In the Pareto trade-off sense, where more than one metric is considered and a solution can be very better in general, but not for one axis) dynamic memory scheme that exists currently in the literature. Furthermore, we propose to use taxonomy of decisions based on orthogonal trees at two different abstraction levels.

Based on the above analysis, Atienza et al. [2] have proposed a taxonomy of the available decisions through a set of decision trees to manage DMM customization. They build a parameterizable design space based on the following decision taxonomy:

1. **Intra-Heap Block Structure Decisions** ⇒ Parameters: {*Block Structure*, *Block Sizes*, *Block Recorded Info*}: It handles the data structures which organize the memory blocks inside each heap of the MTh-DMM.
2. **Intra-Heap Pool Organization Decisions** ⇒ Parameters: {*Pool Structure*, *Pool Structure Based on Block Size*, *Pool Structure Based on Blocks Order*}: It define per heap pools' organization i.e. single pool, one pool per size, traversing order etc.
3. **Intra-Heap Block Allocation/Deallocation Decisions** ⇒ Parameters: {*Allocation Search Order*, *Allocation Fit*, *Destination Pool*}: It deals with the operations to satisfy the DM allocation and deallocation requests.
4. **Intra-Heap Splitting/Coalescing Decisions** ⇒ Parameters: {*Block Size*, *Splitting/Coalescing Frequency*, *Splitting/Coalescing Triggering Criterion*}: It formalizes the decisions to handle the current coalescing and splitting blocks techniques [25], i.e. the threshold logic for coalescing and splitting the blocks.

However, the aforementioned taxonomy covers only the design space of single heap DMM for single-threaded applications (intra-heap level), which is rather limiting for the multi-threading case. In order to provide customized DMM implementations concerning multiple heaps, which is the case for multi-threaded applications, we extended the single heap DMM design space to efficiently model and capture decisions found in the field of multi-processor and multi-threaded application domain. We introduced a new decision taxonomy for the inter-heap

design space [26] which models heap decisions shared for all threads such as heap organization, thread to heap mapping policies etc. The union of inter-heap [26] and intra-heap [2] design spaces returns the overall MTh-DMM design space. The inter-heap decision taxonomy along with its parameters is summarized in the following lines:

1. **Architectural Scheme Decisions** ⇒ Parameters: {*Heap Architecture*, *Global Heap*}: It determines the way the dynamic memory allocator organizes and architects its heaps in order to exploit the available thread-level parallelism into memory management.
2. **Data Coherency Decisions** ⇒ Parameters: {*Synchronization Mechanisms*}: It deals with the existence or not and the structure of the synchronization mechanisms in order to ensure the data coherency in each heap.
3. **Inter-Heap Allocation Decisions** ⇒ Parameters: {*Thread to Heap Mapping*}: It manages the way in which threads allocate memory in the inter-thread level. Allocation in this level is strongly connected with decisions which consider both the thread grouping in order to share a heap and the thread to heap mapping. Allocation decisions of finer granularity i.e. fit policies etc are included into the intra-thread design space.
4. **Inter-Heap Deallocation Decisions** ⇒ Parameters: {*Free Block Movement Strategy*, *Destination Heap*}: It includes trees concerning the ownership aware, deallocation of each memory block and placement decisions for the deallocated blocks.
5. **Inter-Heap Fragmentation Decisions** ⇒ Parameters: {*Threshold*, *Number of Moved Free Blocks*}: It manages the potential memory blowup of the multi-threaded application and consider decisions in order to reduce or bound the worst memory blowup.

The MTh-DMM design space is platform independent and applicable to any MPSoC or NoC platform after a platform dependent refinement. In order to enable modular construction of various MTh-DMM configurations, a C++ library has been developed implementing each decision as a separate component. Modular construction enables the hierarchical composition of the MTh-DMM configuration. Thus, heap management is structured hierarchically across differing management layers (each layer implementing a different policy/mechnanism) rather than with monolithic pieces of code. Heap layers has been originally proposed in [5]. As suggested in [5], we also utilize C++ mixins [12] to build the layers found into the inter-heap design space. Mixins introduce the concept of multiple inheritance overcoming the limitation of single class hierarchy. Each heap layer of the inter- and intra-heap design space corresponds to a mixin that provides a malloc and free method interface to its parent class for allocating/deallocating memory enhanced with proper management mechanisms corresponding to its dedicated functionality.

Figure 2.3 depicts a simple exemplary scenario of building application specific MTh-DMM. We assume the threadtest.c multi-threaded benchmark configured to invoke three threads. Each thread allocates memory blocks of size either 8 or 40

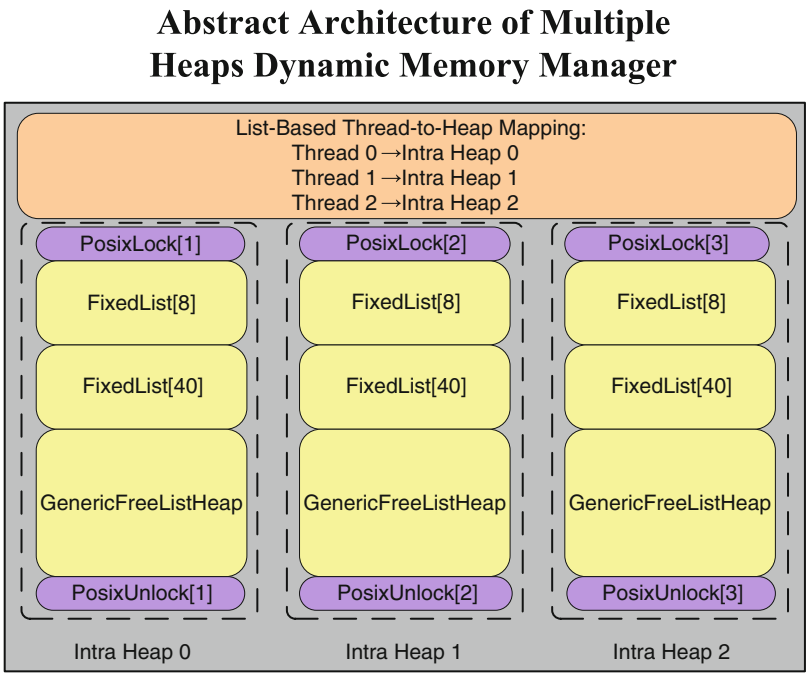

Fig. 2.3 Abstract architectural description of the application specific MTh-DMM

bytes and deallocates only its own blocks. Given the aforementioned allocation behavior, we assume an application specific dynamic memory manager that accommodates a multiple heaps architecture. Specifically, we consider a MTh-DMM with three heaps where each thread is instructed to allocates and deallocate data to only one of the heaps (inter-heap level customization). Each of the heaps is further customized with two fixed size freelists (8-byte and 40-byte respectively) and a generic freelist (intra-heap level customization).

Figure 2.4 shows the code fragment implementation of the application specific allocator composed with the C++ mixins of our library. Specifically, the FIXEDLIST (N) definition implements the structure of generalized fixed size freelist. The function mapFunctionCustom handles the mapping of sizes into the allocator by padding memory sizes requests either to 8 bytes or 40 bytes if the size of allocation request is smaller than 8 or 40 byte respectively. If the requested size is larger than 40 bytes no padding is performed. The RootHeap implements the interface of the allocator to the sbrk() function of the OS in order to request memory when there is no available in the dynamic memory manager. The AdaptLockHeap layer imposes the synchronization interface for memory manager by properly locking and unlocking the malloc and free memory requests of the application. A simple lock mechanism, specifically a Posix mutex [11], handles the synchronization through the PosixLockType component. ThreadHeap0, ThreadHeap1, ThreadHeap3 are the

```
#define NumHeaps 3

#define FIXEDLIST(N) \
SelectorHeap< \
  FIFOSLFixedListHeap<LeaHeader>, \
  SizeSelector<N>, \
  SizeSelector<N> \
>

size_t mapFunctionCustom(size_t sz){
        if (sz <= 8) return 8;
        else if (sz <= 40) return 40;
        else return sz;
}
typedef  SlopHeap<SbrkHeap<EmptyHeader>, LeaHeader> RootHeap;
typedef HeapList<FIFOSLFirstFitHeap<LeaHeader>, RootHeap> GenericFreeListHeap;

typedef ThreadSelectorHeap<AdaprLockHeap<PosixLockType, MapSizeHeap<HeapList<
FIXEDLIST1(8), HeapList< FIXEDLIST1(40), GenericFreeListHeap> >,
mapFunctionPurePrivateHeaps>>, 0 ,NumHeaps> ThreadHeap0;

typedef ThreadSelectorHeap<AdaprLockHeap<PosixLockType, MapSizeHeap<HeapList<
FIXEDLIST1(8), HeapList< FIXEDLIST1(40), GenericFreeListHeap> >,
mapFunctionPurePrivateHeaps>>, 1 ,NumHeaps> ThreadHeap1;

typedef ThreadSelectorHeap<AdaprLockHeap<PosixLockType, MapSizeHeap<HeapList<
FIXEDLIST1(8), HeapList< FIXEDLIST1(40), GenericFreeListHeap> >,
mapFunctionPurePrivateHeaps>>, 2 ,NumHeaps> ThreadHeap2;

typedef  HeapList<ThreadHeap0, HeapList<ThreadHeap1, ThreadHeap2> >
MultipleHeaps;
```

Fig. 2.4 Code fragment implementing the MTh-DMM of Fig. 2.3 based on C++ mixins

private per-thread heaps. The ThreadSelector layer implements a distributed the modulo hash function in order to map thread IDs to heap IDs. The per-thread heaps are combined together and the final MultipleHeaps allocator implements the desired application specific dynamic memory manager. It should be noticed that the three per-thread heaps are identical due to the threadtest application that considers

all the threads to do the same job in parallel. Following the template depicted in Fig. 2.4, customization of each per-thread heap can be done easily by customizing the ThreadHeap0, ThreadHeap1, ThreadHeap2 heaps.

2.6 A Framework for MTh-DMM Exploration

The goal of MTh-DMM exploration is to generate Pareto sets [22] of customized MTh-DMM, tailored to the designer's constraints and the application's specific needs. In this section, we introduce the exploration framework that enables automated code generation and evaluation of customized MTh-DMM configurations. The framework is structured on the partition of the MTh-DMM design space into the inter- and intra-heap design subspaces. The MTh-DMM exploration framework integrates the following components:

1. The inter-heap exploration tool that automatically generates the inter-heap decision vectors and the source code for each MTh-DMM configuration.
2. The intra-heap exploration tool that automatically generates the intra-heap decision vectors and the source code for each MTh-DMM configuration.
3. The MTh-DMM C++ library implementing the decisions of the MTh-DMM design space in a modular manner (Sect. 2.5).
4. The automated analysis tools.

Based on the aforementioned framework, we developed two exploration variants. A two-phase MTh-DMM exploration approach that explores both the inter- and intra-heap design space and an aggressive single-phase MTh-DMM exploration that explores only inter-heap decisions considering pre-configuration of the intra-heap decisions. In the remainder of this section, we present the two exploration variants and we analyze the tools of the MTh-DMM exploration framework.

2.6.1 Two-Phase Constraint-Orthogonal MTh-DMM Exploration

The two-phase exploration is guided by the observation that inter-heap decisions are shared since they concern the management of the overall allocated heaps of the MTh-DMM in a global manner, while intra-thread design space decisions concerns the management of each allocated intra-heap individually. Thus, the inter-thread level includes decisions of globalized and shared policies, while intra-thread level defines heap local customization policies. Since the policy semantics are orthogonal for the inter- and intra-thread design spaces, we partitioned the problem of exploring and designing custom multi-threaded dynamic memory managers into two constraint-orthogonal problems [26]. The first problem (phase 1) refers to the

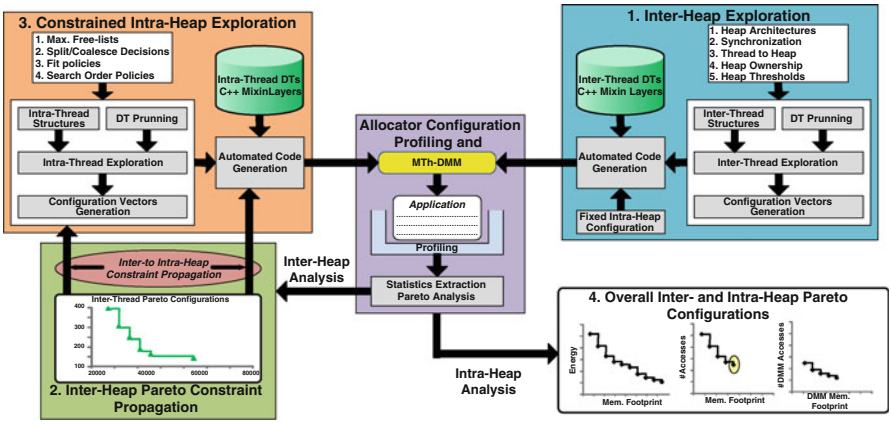

Fig. 2.5 Four-phase constraint-orthogonal exploration methodology

exploration of inter-thread decisions, while the second one refers to exploration decisions available in intra-thread design space (phase 2). The inter-heap MTh-DMM exploration problem generates a Pareto optimal set of solutions (we silently consider multi-objective optimization for the MTh-DMMs). These Pareto configurations are propagated as constraints to the intra-thread level exploration. Thus, local heap customization decisions are explored over global Pareto decisions. In this way, we manage to the handle the complexity of customizing at the intra-heap level each possible inter-thread configuration.

The four-phase constraint-orthogonal exploration methodology is depicted in Fig. 2.5. In an abstract manner, the major steps of the proposed exploration methodology are summarized as follows:

1. Given the dynamic application, its native source code is annotated with proper profiling constructs that capture the dynamic memory behavior of the application. The software designer performs this task manually.
2. Inter-heap level exploration is then performed in order to automatically generate the source code of valid MTh-DMM solutions. The MTh-DMMs configurations are linked with the dynamic application's source code and each solution is compiled and evaluated based on various metrics. A Pareto analyzer is invoked in order to extract the Pareto optimal inter-thread level MTh-DMM solutions. These Pareto optimal solutions form the "inter- to intra-heap" level constraints and they are propagated to the intra-thread level exploration.
3. Intra-thread level exploration is invoked customizing each heap individually found in the propagated inter-thread Pareto solutions. Automatic code generation produces valid intra-heap customized MTh-DMM solutions for each Pareto optimal point. The intra-heap level customized MTh-DMMs are linked again with the dynamic application's source code and each solution is re-compiled and re-evaluated. The Pareto analyzer is invoked again and the combined intra- and

inter-thread level Pareto optimal MTh-DMM solutions are extracted and returned to the designer as the final customized MTh-DMM solutions. Having the Pareto optimal solutions the designer selects the MTh-DMM solution, which satisfies their design constraints.

2.6.2 *Single-Phase Aggressive Inter-Heap MTh-DMM Exploration*

In the two-phase exploration methodology the final Pareto DMM configurations are extracted evaluating DMM configurations twice (one time per each design space level). However, the evaluation procedure (multiple compilations and executions/simulations of the dynamic application) is an extremely time consuming task. Furthermore, the large number of intra-heap decisions imposes a huge number of possible configurations to be explored (i.e. up to 19.000.000 intra-heap configurations for exhaustive exploration of a two heap DMM architecture). Thus, the elimination of the second (intra-heap) exploration and evaluation phase seems a very attractive approach especially in cases that strict design-time constraints are imposed to the design team. However, by simply eliminating the intra-thread level DMM customization far from optimal solutions are generated.

We propose an aggressive access-oriented customization approach which eliminates the need of exploring the intra-thread level design decisions without degrading the quality of final solutions concerning the number of total memory accesses. We achieve this by analyzing the impact of major intra-thread decisions onto the number of memory accesses of the DM manager and by aggressively customize the intra-thread decisions during inter-thread exploration.

We recognize two major decisions that affect the number of memory accesses performed by the DMM: (i) The number of fixed-sized freelists and (ii) the allocation fit strategy. The more the fixed-sized freelists the higher the possibility a memory block to be allocated with only one memory access. In the same sense, in case that malloc requests a memory block of size different than available a free-blocks in the free-lists, then the first fit allocation strategy guarantees that the least number of memory accesses will be performed in order to discover free memory to allocate.

The single-phase exploration strategy relies in the aggressive customization of the fixed intra-thread configuration which is utilized from the code generation module during inter-heap exploration. Figure 2.6 depicts the component, which enables this aggressive exploration approach. Thus, the internal structure of each heap is customized to include fixed-size free-lists of all the dominant block sizes requested by the dynamic application. In addition, each heap adopts a first fit allocation strategy. Independently from the thread-to-heap mapping, the number heaps and the of MTh-DMM's heap architecture decisions, each inter-thread solution produced by this aggressive exploration will include this type of heaps.

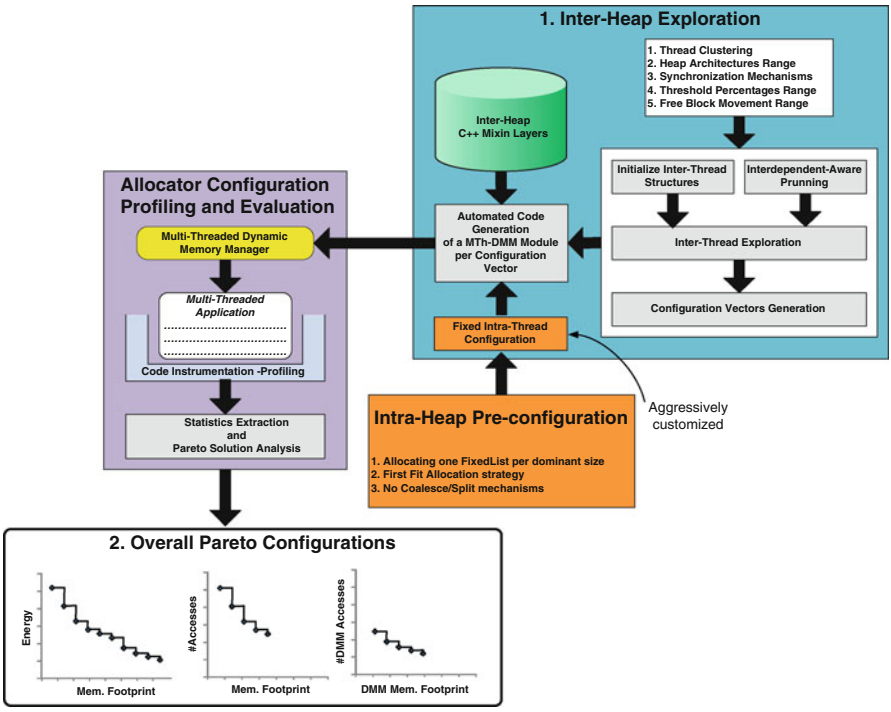

Fig. 2.6 Two-phase constraint-orthogonal exploration methodology

2.6.3 Tool A: Inter-Heap Exploration

Inter-heap exploration searches for optimal combinations of design decisions at the inter-heap level for the dynamic application under study. In order to reduce the huge number search space, the exploration procedure has been extended to incorporate solution pruning based on the interdependencies of the inter-heap decisions [26]. Assuming a straightforward inter-thread level exploration procedure for a dynamic application consisting of five threads, there are 544.320 DMM configurations that have to be explored and evaluated. The incorporation of the solution pruning during the exploration process reduces the number of DMM solutions that are worthwhile to be examined down to 67.655 configurations. Assuming uniform generation and evaluation delays for each DMM configuration, the proposed pruning methodology offers a speedup in exploration time up to x8.

The designer can either guide the exploration procedure by setting proper ranges to the inter-heap exploration parameters or let the tool perform exploration with its default parameter values. For example, in case that a number of thread to heap clusters have been a-priori decided based on platform specific constraints, the designer can force the inter-thread DMM exploration tool to not evaluate

all the available thread-to-heap mappings, reducing significantly the explorations' runtime. After the determination of the exploration parameters (user guided or tool default), the exploration script generates the MTh-DMM solution set. Each solution is a different configuration of a dynamic memory manager. The configuration vectors are fed into an automated code generator module that produces the C++ implementation of each dynamic memory manager according to the specified decisions. The code generator is linked with a C++ software library that contains the software implementations of each decision found into the inter-heap decision trees. During this level of customization, the library includes the decisions concerning the inter-heap design space (intra-heap specific DMM decisions are not taken into account). Each intra-heap defined into the inter-thread level configuration vector is considered pre-defined. We considered a structure consisting of a first-fit based general heap for primary allocation/deallocation services extended with a general FIFO-based freelist. Thus, each heap defined into the inter-thread configuration vector follows the above description. During the intra- thread level exploration step, this heap definition will be refined and customized for each heap individually found into the inter-thread Pareto optimal solutions.

2.6.4 Tool B: Profiling and Evaluation

The tool developed for the evaluation of various DMM configurations is actually used for the evaluation of both inter- and intra-heap customized solutions. We automatically collect and analyze statistics of the dynamic application when various DMM managers are invoked, through proper instrumentation. The instrumentation consists of the insertion of proper profiling constructs, which are implemented as part of an advanced profiling library [3]. Each automatically generated MTh-DMM solution is evaluated according to the following statistics:

1. The number of memory accesses (both of the overall application and of the DMM manager isolated),
2. The maximum memory footprint requested by the both the dynamic application or the DMM manager individually,
3. The execution time for the overall application,
4. The per-thread predominant block size (information used during intra-thread exploration and customization).

After the collection of the proper statistics for each examined DMM solution, a Pareto analyzer module is invoked in order to extract the solutions, which present the most efficient trade-offs (Pareto solutions). In case that the statistics collection is performed to evaluate the inter-heap customized DMM solutions, the extracted Pareto optimal points form the Pareto constraints to be propagated to the intra-thread level exploration tool. In case that the evaluation is performed onto the intra-heap level customized DMM solution, the extracted Pareto curve includes the final Pareto optimal DMM configurations which are returned to the designer.

2.6.5 Tool C: Inter-Heap Exploration

The intra-heap exploration tool has similar software architecture with the inter-heap one. It is invoked only in the two-phase constraint-orthogonal exploration approach and performs the extra refinement/customization of the inter-heap Pareto optimal solutions. Each inter-heap Pareto solution is propagated to the intra-thread exploration tool forming its constraints. Since each inter-heap Pareto solution is formed from differing inter-heap level decisions, intra-heap exploration customizes the internal structure of each instantiated heap. We developed two heuristic strategies considering intra-heap exploration:

1. An access-oriented (AO) heuristic and
2. A footprint-oriented (FO) heuristic.

The intra-heap heuristics are based on the traversing guidelines proposed in [2] for single-threaded and single processor customization. In the memory access-oriented heuristic the main goal of exploration is to find customized intra-heap organizations with low memory accesses overhead. In the memory footprint oriented heuristic the main goal of exploration is to find intra-heap customized solutions with low requirements in memory size.

The AO heuristic excludes from the exploration the intra-heap decisions that inherently increase the number memory accesses. Thus, splitting and coalescing decisions are not taken into account during exploration since they both require an increased number of accesses in order to determine whether they applied and to generate the coalesced/splitted blocks. In the same sense, first-fit strategy is preferred in comparison to the corresponding best-fit strategies which consume many memory accesses to traverse the dynamic data type structures. In addition, the instantiation of fixed-sized free-lists inside each heap has positive impact on the number of memory accesses. In case that there is not a fixed-sized freelist available in the heap instantiation, then the allocation has to traverse the dynamic data type structures in order to find the first available free block.

The FO heuristic excludes from the exploration decisions that inherently increase the memory wastage. Splitting and coalescing decisions are both taken into account for various ranges of interest, since both target to lowering the memory waste by generating at the runtime new proper memory blocks for the dynamic memory requests. In the same sense, best fit and exact de/allocation strategies invoked for exploration in comparison to the first fit strategy, which is excluded. By searching for better block fittings the memory footprint is reduced since the better the fitting the lower the memory waste during allocation (through reducing internal fragmentation). As in the memory accesses-oriented heuristic, the instantiation of fixed-size free-lists inside each heap has positive impact on the maximum requested memory footprint. In case of free blocks being accommodated into fixed-sized freelists then a memory request for allocation of that size will be served with zero memory waste (no invocation of internal fragmentation).

2.7 Case Study: A Multi-Threaded Wireless Application

We experimentally evaluate the proposed exploration methodology and the corresponding tool flow based on a real-life case study of a dynamic multi-threaded wireless application as the one in [3]. The application consists of 5 kernels which are triggered by wireless streams. Each kernel corresponds to a thread and communicates asynchronously with the other threads in a Linux multi-threaded environment. The threads dynamically allocate and deallocate data according to the characteristics of the incoming network trace. The distribution of the incoming packet sizes is shown in Fig. 2.7a, whereas the size distribution of the allocated blocks in Fig. 2.7b.

The exploration methodologies, presented in Sect. 2.6, has been applied to the aforementioned network application. At first, inter-heap level exploration has been performed. A solution space of 67.655 valid and semantically disjoint MTh-DMM configurations has been generated.

Figure 2.8 depicts the three Pareto curves generated after the implication of the two-phase exploration. The customization of inter-heap Pareto solutions is graphically depicted through the inter-heap Pareto curve shifting towards MTh-DMM solutions with either less memory accesses (in case of AO heuristic) or lower required memory footprint (in case of FO heuristic).

In Fig. 2.8b, the Pareto curve generated by the two-phase AO exploration methodology has been overlapped to the one generated by the aggressive one. Following the aggressive exploration, a curve shifting towards higher quality solutions is depicted. This Pareto curve shifting delivered by the aggressive exploration does not mean that the two-phase exploration (both inter- and intra-thread customization) is worthless. Pareto solutions delivered by the aggressive inter-thread exploration are also included into the solution space of the two-phase exploration. However, the intra-thread access-oriented heuristic exploration eliminated the examination of these solutions. By properly adjusting the bounds of the intra-thread access-oriented heuristic to accommodate the examination of a larger number of MTh-DMM solutions of the same or higher quality are expected to be delivered in the expense of larger exploration's runtime.

We have also compared custom implementations of MTh-DM managers with Windows-XP Kingsley [15] and Hoard [4] allocators (Fig. 2.9). We considered four variants of custom MTh-DMMs. Specifically, we considered a custom serial heap (Non-Pareto), a custom pure private heaps allocator (Non-Pareto), a custom Hoard-like DM manager (Non-Pareto) and best custom FO MTh-DMM solution extracted from the final Pareto curves (Fig. 2.8). Figure 2.9 shows the efficiency of custom MTh-DMM solution generated by the two-phase exploration methodology. In terms of number of memory accesses (metric proportional to the energy consumption), the custom FO MTh-DMM delivers an average reduction of 46K in memory accesses (Fig. 2.7). In terms of memory footprint in Fig. 2.7, the custom FO MTh-DM configuration is the second best solution after Hoard allocator with an average footprint reduction of 8.8KBytes. Hoard's efficiency in memory footprint is due to its internal heap organization with a large set of size-bins.

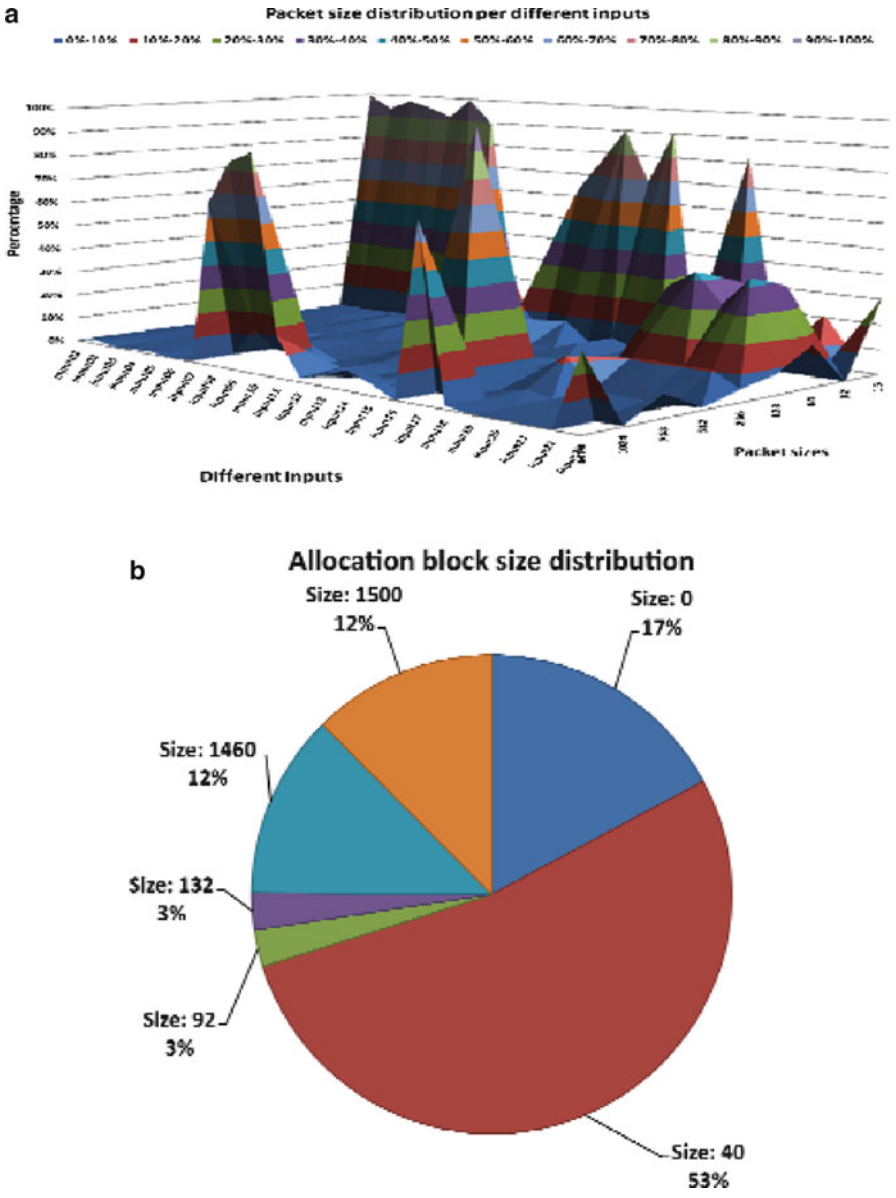

Fig. 2.7 Multi-threaded application characterization. (**a**) Network packet size distribution, (**b**) Block sizes distribution [3]

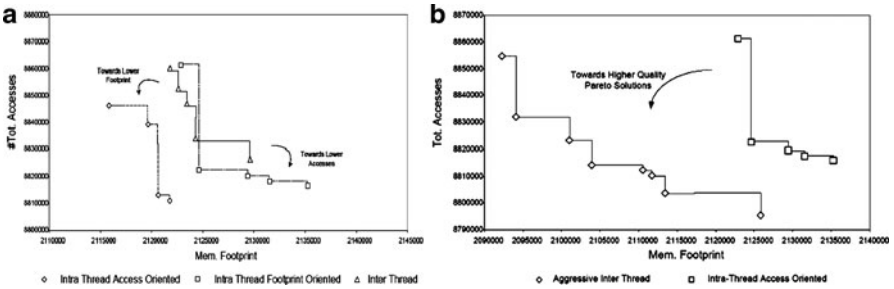

Fig. 2.8 Multi-threaded application characterization. (**a**) Network packet size distribution, (**b**) Comparing the solution quality of two- and single-phase explorations for the AO case

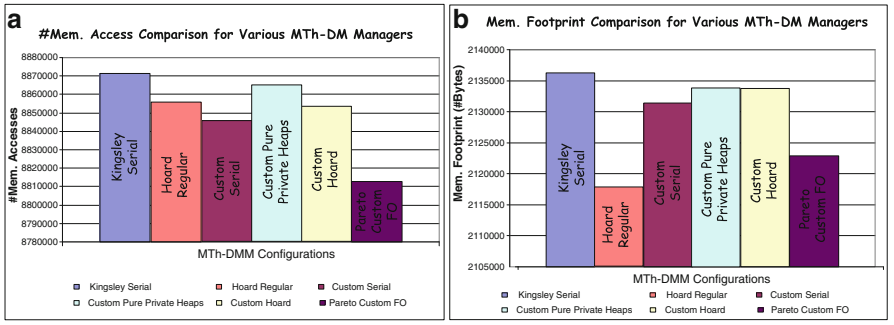

Fig. 2.9 Comparative results for Kingsley-XP [15], Hoard [4] and various custom MTh-DM managers

2.8 Conclusions

In this chapter, we presented an exploration methodology and the software framework for application specific dynamic memory management targeting the multi-threaded embedded applications. A software C++ library has been developed that implements the design decisions regarding customized dynamic memory management. Furthermore, we proposed two exploration variants, namely a two-phase exploration methodology complement with various heuristics and a single-phase aggressive one to efficiently traverse and explore through design space. Specialized software tools have been developed in order to support the full automation of both exploration approaches. The efficiency on generating customized multi- threaded dynamic memory managers have been evaluated through extensive experimental results and comparisons.

References

1. S. Agarwala et al. A 65nm C64x+ Multi-Core DSP Platform for Communications Infrastructure. In *Proc. of ISSCC*, pages 262–601. IEEE Press, 2007.
2. D. Atienza et al. Systematic Dynamic Memory Management Design Methodology for Reduced Memory Footprint. *ACM Trans. Des. Autom. Electron. Syst. (TODAES)*, 11(2):465–489, Apr. 2006.
3. A. Bartzas et al. Enabling run-time memory data transfer optimizations at the system level with automated extraction of embedded software metadata information. In *Proc. of ASP-DAC*, pages 434–439, 2008.
4. E. Berger et al. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not*, 35(11), Nov. 2000.
5. Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *In Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI*, pages 114–124, 2001.
6. B. Bigler et al. Parallel dynamic storage allocation. In *Proc. of ICPP*, pages 272–275, 1985.
7. C. Schlatter Ellis and T. J. Olson. Algorithms for parallel memory allocation. *International Journal of Parallel Programming*, 17(4):303–345, 1988.
8. D. Atienza, S. Mamagkakis, M. Leeman, F. Catthoor, J. M. Mendias, D. Soudris, and G. Deconinck. Fast system-level prototyping of power-aware dynamic memory managers for embedded systems. In *Proc. of PACT*, page, 2003.
9. D. Grunwald, and B. Zorn. CustoMalloc: efficient synthesized memory allocators. *Softw. Pract. Exper.*, 23(8):851–869, 1993.
10. D.Atienza, S.Mamagkakis, F.Catthoor, J.M. Mendias, and D.Soudris. Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications. In *Proc. of DATE*, 2004.
11. F. Garcia, J. Fernandez. POSIX thread libraries. *Linux Journal*, page 36.
12. G. Bracha and W. Hook. Mixin-based inheritance. In *Proc. of OOPSLA*, pages 303–311, 1990.
13. A. Iyengar. Parallel dynamic storage allocation algorithms. In *Proc. of PDP*, 1993.
14. L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test of Computers*, ():74–85, Apr. 2000.
15. M. R. Krishnan. Heap: Pleasures and pains. *Microsoft Developer Newsletter*, 1999.
16. P. E. McKenney and J. Slingwine. Efficient kernel memory allocation on shared memory multiprocessor. In *Proc. of USENIX*, pages 295–305, 1993.
17. P. Larson, M. Krishnan. Memory allocation for long-running server applications. In *Proc. of the ISMM*, 1998.
18. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. of FOCS*, pages 356–368, Nov. 1994.
19. K. Hirata and J. Goodacre. ARM MPCore; The streamlined and scalable ARM11 processor core. In *Proc. of ASP-DAC*, pages 747–748. IEEE Computer Society, 2007.
20. Solaris 9 Reference Manual man pages for mtmalloc. http://docs.sun.com/.
21. T. Johnson. A concurrent fast-fits memory manager. *Technical Report TR91-009, University of Florida, Department of CIS*, (), 1991.
22. V. Pareto. *Manuale di Economia Politica.* Picola Biblioteca Scientifica, Milan, 1906, Translated into English by Ann Schweir (1971), Manual of Political Economy, MacMillan, London, 2008.
23. V. Vee and W. Hsu. A scalable and efficient storage allocator on shared memory multiprocessors. In *Proc. of I-SPAN*, pages 230–235, 1999.
24. W. Gloger. Dynamic memory allocator implementations in Linux system libraries. *http://www.dent.med.uni-muenchen.de/ wmglo/malloc-slides.html*, 2002.

25. P. R. Wilson et al. Dynamic storage allocation, a survey and critical review. In *Proc. of IWMM*, 1995.
26. Sotirios Xydis, Alexandros Bartzas, Iraklis Anagnostopoulos, Dimitrios Soudris, and Kiamal Pekmestzi. Custom mutli-threaded dynamic memory management for multiprocessor system-on-chip platforms. In *ICSAMOS '10: Proceedings of Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 102–109, jul. 2010.

Chapter 3
Power Management Architecture in McNoC

Jean-Michel Chabloz and Ahmed Hemani

Abstract In this chapter we present the power management architecture of the McNoC platform. The power management architecture of McNoC offers distributed Dynamic Voltage Frequency Scaling (DVFS) and power down services to the platform at a fine level of granularity, allowing independent setting of frequency and supply voltage to all switch and resource nodes in the platform. The design style enables hierarchical physical design and solves the clock-domain-crossing problem with a solution based on rationally-related frequencies, which avoids the overhead associated with handshake. The architecture allows arbitrary power management regions to be defined and region-wide power management commands affecting all nodes in a region can be issued by the software layer that we call as Power Management Intelligence (PMINT).

3.1 Introduction

The design of the McNOC power management architecture has been motivated by several factors related to VLSI engineering effort and technology limitations:

1. In modern chips, latency insensitive hierarchical physical design is needed to eliminate the need for global clock balancing and chip level timing closure [1–3]. The state-of-the art practice of flat physical design incurs large turnaround time that not only increases the engineering cost but also restricts effective design space exploration besides preventing reuse in form of hard IPs.

J.-M. Chabloz • A. Hemani (✉)
ES Department, School of ICT, KTH, Isafjordsgatan 39, FORUM 120, 16440 Kista, Sweden
e-mail: chabloz@kth.se; hemani@kth.se

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_3,

2. Efficient and distributed Dynamic Voltage and Frequency Scaling is needed to reduce power consumption [4]. Implementing Dynamic Voltage and Frequency Scaling also requires the ability to safely and efficiently communicate across different clock and voltage domains [5].
3. To enable platform level reuse for widely varying use cases, it is essential to enable the possibility of defining arbitrary power management region boundaries. This decision can be made at build time, boot time or even runtime.
4. Complexity of the next generation McNoC-like platforms requires distributed power management rather than a centralized power management [4].

3.1.1 McNoC Overview

The main features of the power management architecture of the McNoC platform are:

- Latency-insensitive hierarchical physical design style
- Globally-Ratiochronous, Locally-Synchronous clocking
- Dynamic Voltage and Frequency Scaling
- Distributed power management
- Programmable voltage and frequency domains

An overview of McNoC is shown in Fig. 3.1. McNoC is a computing platform based on the Nostrum Network-on-Chip [6, 7]. Every resource node contains a processing unit, a memory unit and a Data Management Engine (DME) which acts as a memory management unit and interfaces the node with the network. The switch nodes are synchronous. A power management architecture has been built on top of McNoC by introducing a wrapper around every node. The wrapper is used to ensure safe communication between nodes and to allow frequency and voltage regulation. The access point to provide the power services is given by the Power Management Unit (PMU), which controls a Voltage Control Unit (VCU) and a Clock Generation Unit (CGU), respectively used to control the voltage and the frequency in the node. Power Management Units are controlled by the Power Management INTelligence (PMINT), a controlling software entity residing in one of the nodes of the platform, which enforces the energetic policies. The Always-on LOgic In Node (ALOIN) unit present in every node remains on even when a node is powered down and is used to wake up the node. McNoCs CGU, VCU, PMU and ALOIN are described in Sects. 3.2.2, 3.2.3, 3.2.5 and 3.2.6. An overview of McNoCs architecture is given in Sect. 3.2.1.

3.1.2 Latency-Insensitive Hierarchical Physical Design Style

The McNoC platform targets next-generation, multi-million gates VLSI systems in which a globally-synchronous assumption is nearly impossible to implement

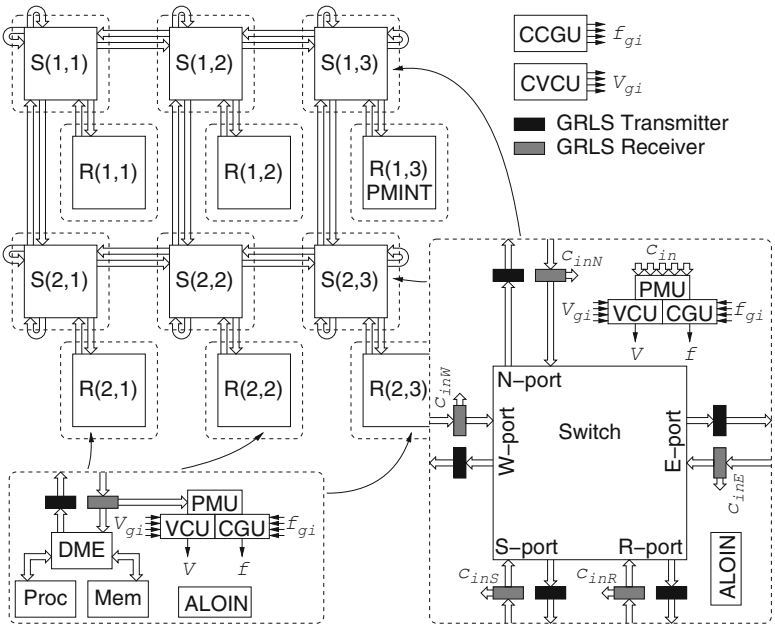

Fig. 3.1 McNoC overview

or comes with a huge penalty in terms of engineering effort, loss of power and performance [8] and most importantly is not compatible with the needs of modern power management based on DVFS [1, 2]. Preserving the globally-synchronous paradigm requires design of globally-balanced clock trees that is proving to be impractical even for present-day 10M+ gates SoCs and will be practically impossible for next-generation 100M+ gates SoCs [5]. These large SoCs are assembled from pre-designed IPs and sub-systems. Architecturally, these IPs are independent but when assembled physically, change in one IP can increase the load on the global clock or cause it to be routed differently, which disturbs the timing relationship of signals to other IPs with respect to clock and requires a chip level timing closure [8]. This chip level timing closure, especially when required to do for multi-modes multi-corners is a significant factor in the NRE cost. The desired solution would be if each IP's timing was not affected by the change in global clock or the timing of other IPs with which it interacts. In other words, a latency insensitive design style would eliminate the chip level timing closure and allow a truly hierarchical physical design methodology.

3.1.3 Globally-Ratiochronous, Locally-Synchronous Clocking

The problem of achieving latency insensitivity in many respects boils down to that of safe data communication between two communicating functionalities – abstracted as IPs above. The truest form of latency insensitivity is achieved by having

asynchronous communication, which completely eliminates the clock. While fully-asynchronous design has been achieved [9] it remains a niche design style that is hard to apply in the general case and is not compatible with the established synchronous-design-style-based SoC design tools and methods used in industry [10]. Globally-Asynchronous Locally-Synchronous (GALS) [11–13] design styles are a compromise and introduce asynchronicity in a restricted sense at global level, while retaining a synchronous design style at local level – local within an island (IP or a sub-system). This design style however comes with a severe performance penalty in the form of round-trip delay that is required for request-acknowledge signalling for every data transfer [14].

The power management architecture of McNoC is based on a clocking strategy that restricts the clock frequencies to be rationally-related, i.e. all clocks on the chip run at frequencies which are submultiple of a physical or ideal frequency f_H. This restriction allows a significant simplification in the implementation of synchronizers used to bridge rationally-related clock domains and, most importantly, it eliminates the round trip delay penalty associated with traditional GALS implementation styles. The clocking strategy is called Globally-Ratiochronous, Locally-Synchronous (GRLS) [14]. It introduces a limited and acceptable loss of flexibility compared to GALS [15], in which no restriction is made on the local clock frequencies. The synchronizers are described in Sect. 3.2.4.

3.1.4 Dynamic Voltage and Frequency Scaling

To deploy Dynamic Voltage Frequency Scaling (DVFS) in McNoC, the GRLS clocking strategy is coupled with quantized voltage levels [16–18], as will be discussed in Sect. 3.2.3. Up to 4 global supply voltages are distributed throughout the chip. The decision to use 4 VDDs is based on our investigation that suggests that the overhead of more than VDDs compared with the increased flexibility is not justifiable. The supply voltage of every node can be switched to any of the four global supply voltages using PMOS power switches built into every node. Multiple supply voltages and power switches are supported by state-of-the-art synthesis/place-and-route flows [19]. Some limited and acceptable loss of flexibility is introduced compared to a solution allowing complete freedom in the choice of the supply voltages [15]. The latter approach would be hard to apply in a real system because it would require a voltage regulator in every node, which would carry a very high overhead and complicate the design process [20].

3.1.5 Distributed Power Management

The combination of the GRLS design style and the quantized supply voltages allows the design of low-overhead communication interfaces [14,21] and frequency/

voltage regulators [15], i.e. all the components needed for effective and efficient power management. These components can be integrated at relatively fine levels of granularity. Unlike GALS, for which the high latency of the communication interfaces is a bottleneck [22, 23], the GRLS design style allows isolation of every switch and resource node into an independent power management domain, so that the frequency and the supply voltage of every node can be independently set. Traditionally, power management techniques in NoC-based platforms have never allowed independent power management for the switch nodes, restricting the benefits of distributed DVFS to the resource nodes only [24, 25].

The power services that are offered to every node are DVFS operating point change and shutdown/wakeup. Because shutting down switches in the NoC disrupts the communication fabric of McNoC, an always-on block is inserted in every node to wake it up.

3.1.6 Programmable Voltage and Frequency Domains

While the power management architecture in McNoC allows potentially to treat every node as a separate power management domain, for performance and complexity reasons the platform can be dynamically organized into power management regions at boot time or even at runtime. Nodes belonging to the same region always run at the same frequency and are always all awake or all asleep. This is justified because nodes belonging to the same power management region have a coherent need from the power management perspective. This also simplifies the working of the Power Management INTelligence (PMINT), the power management controller normally implemented as a software entity residing in one of the nodes of the platform. PMINT issues a single command to one of the nodes in a target region to change the DVFS point of all nodes in the region, or shut them all down.

3.2 Power Management Architecture

3.2.1 McNoC Overview

McNoC is built upon the existing Nostrum Network-on-Chip [6, 7]. Nostrum is based on simple, low-overhead switches employing X-Y routing and supports regular mesh topologies. The output ports of the switches on the border of the network are closed back on the switches themselves (see Fig. 3.1), which allows all switches in the network to have the same structure. As an example, the east output port of a switch on the east border of the network is fed back to the east input port of the same switch. Nostrum switches are bufferless and based on deflection routing. Packets are never dropped but can be misrouted when contention is encountered.

Buffers are only inserted at the resource-network interface, where packets can be delayed for entrance in the network. The properties of the routing algorithm ensure that the Nostrum NoC cannot get into deadlock or livelock [7].

In McNoC, up to 4 global clocks, indicated as $clk_{g0}...clk_{g3}$ and up to 4 global supply voltages, indicated as $V_{g0}...V_{g3}$, are distributed throughout the chip to allow generation of the local frequency f and supply voltage V in every node. The frequencies of the global clocks are all rationally-related, i.e. they are all submultiple of a frequency f_H.

Every node (switch or resource) of the McNoC platform is enclosed into a power management wrapper. The wrapper has the purpose to create the local clock clk running at frequency f and the supply voltage V from the global clocks and global voltages, to give access to the power services and to guarantee safe inter-node communication. A simplified schematic structure of the wrapper is shown in Fig. 3.1. To keep the presentation simple, the figure does not show any connection for the ALOIN blocks. Every power management wrapper contains a Clock Generation Unit (CGU), a Voltage Control Unit (VCU), GRLS Transmitter and one GRLS Receiver for every bidirectional link, Power Management Unit (PMU) and a Always-on LOgic In Node (ALOIN). A brief description of these block is followed by a more detailed description of each of these blocks:

- The Voltage Control Unit (VCU) selects one of the global voltages and generates the voltage V for the node. It is controlled by the PMU.
- The Clock Generation Unit (CGU) selects one of the global clocks, divide its frequency by a programmable integer number and generates the frequency f for the node. It is controlled by the PMU.
- The GRLS Transmitters and Receivers are the two ends of the GRLS interfaces in charge of allowing safe communication between the nodes. They are programmed by the Power Management Unit, which instructs them of the frequencies of both the Transmitter and the Receiver node. The GRLS Receivers also have the responsibility to identify incoming commands for the Power Management Unit and deliver them to it. The Power management Unit can use a GRLS Transmitter to send a command to a PMU in a neighboring node.
- The Power Management Unit (PMU) acts as an access point for the power services. It is accessed by the Power Management INTelligence (PMINT), a controlling entity which can be positioned in any resource node of the network. The Power Management Unit implements the power management services by controlling the CGU and the VCU, and coordinates with PMUs in neighboring nodes to ensure that power management services are implemented correctly and without packet loss.
- The Always-on LOgic In Node (ALOIN) is a block that is always kept operational, even when the node is powered down. The ALOINs are connected in a network which runs parallel to the Nostrum NoC and which utilizes one-bit links. The function of the ALOIN network is to wake up the nodes that have been powered down.

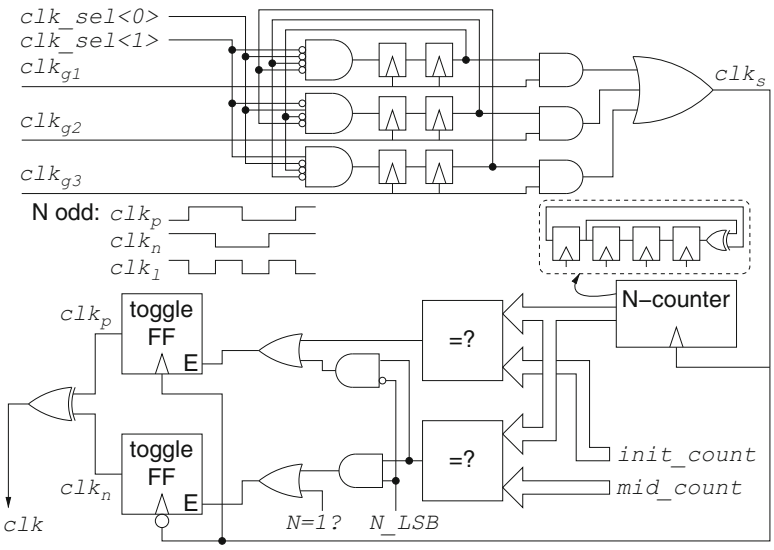

Fig. 3.2 Structure of a Clock Generation Unit supporting three global clocks

3.2.2 Clock Generation Unit

In McNoC, there are two types of CGUs, one central and the other local. The central CGU generates up to four global clocks $clk_{g0}...clk_{g3}$ running at rationally-related frequencies $f_{g0}...f_{g3}$ and distributed throughout the chip using unbalanced clock trees. If more than one clock is present, the frequencies of the global clocks should all be submultiple of a frequency f_H. f_H is not necessarily a physical frequency but is derived from the frequencies of the global clocks: as an example, if two global clocks are present, with $f_{g0} = 200\,\text{MHz}$ and $f_{g1} = 300\,\text{MHz}$, then $f_H = 600\,\text{MHz}$ and f_H does not need to be distributed in the system. In McNoC, division ratios $\frac{f_H}{f_{gi}}$ up to 15 are supported.

In every node, a Clock Generation Unit (CGU) selects one among the global clocks and divides its frequency by a programmable integer value to generate the local clock *clk* running at frequency f. As a result, all local frequencies are submultiple of f_H. The area overhead of distributing multiple global clocks is negligible compared to the area overhead of the local clock trees because the global clock trees have a fanout equal only to the number of islands and, being unbalanced, can be routed in the most convenient way. The structure of a CGU supporting three global clocks is shown in Fig. 3.2.

The structure of the selection stage ensures that, when *clk_sel* is changed to select a new global clock, the previously-selected clock is first gated; only when all clocks are gated, the newly-selected clock is ungated. Double-stage synchronizers are used to make the system metastability-safe and glitch-free. This is necessary because the global clock trees are unbalanced and no assumption is made on the arrival time

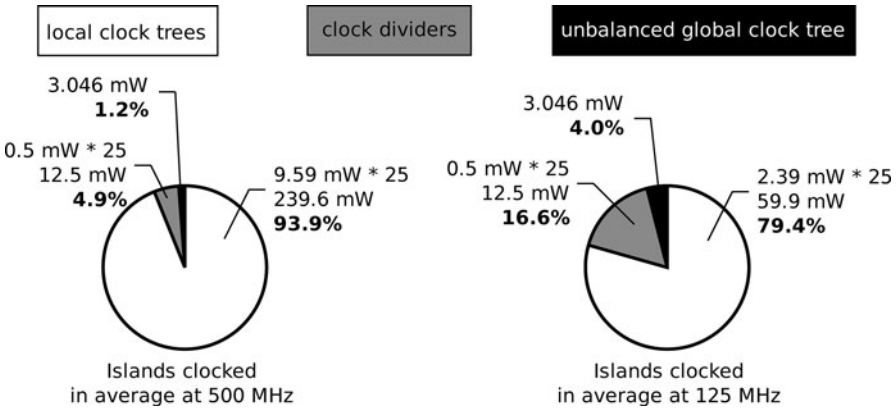

Fig. 3.3 Power breakdown of a 2 mm × 2 mm, 90 nm chip with 25 equal-sized GRLS islands, each containing 3,600 flipflops, with a single, 1 GHz global clock

of the global clocks at the local CGUs. The synchronizers introduce a two-cycles latency when the global clock is changed but introduce no latency during normal operation.

The divider stage divides the frequency of the selected clock clk_s by an integer number N between 1 and 15. It is composed of a 4-bits LFSR counter having a sequence of N states. *init_count* contains the value of the counter right after it is reset and *mid_count* contains the value of the counter $\lceil \frac{N}{2} \rceil$ cycles later. The comparators outputs are used to generate the enable signals for a positive- and a negative-edge-triggered toggle flipflops. When N is even, all edges of clk are synchronous with rising edges of clk_s; the negative-edge-triggered flipflop never toggles while the positive-edge-triggered flipflop toggles twice during a count sequence of the counter. When N is odd, clk is obtained by combining the two clocks clk_p and clk_n, each running at $\frac{f}{2}$ and in opposition of phase.

3.2.2.1 Overhead and Performance Analysis

Other implementations of the CGU are possible, but this solution was selected because of its low overhead. In 90 nm ASIC technology, its area occupation is 140 NAND-equivalents. The CGU is built using only standard cells and is very fast in changing its output frequency. In 90 nm ASIC technology, the power consumption of a CGU is 0.5 mW when it is locked on a global clock running at 1 GHz. Experiments were conducted to estimate the power overhead of the CGU and the frequency distribution system by considering a very fine-granularity, 2 mm × 2 mm, 90 nm chip divided in 25 synchronous islands. With a single 1 GHz global clock and every island containing 3,600 flipflops, the power breakdown depends on the average frequency at which the islands are clocked. The power breakdowns for two cases are shown in Fig. 3.3. The percentage overhead is higher when the average island

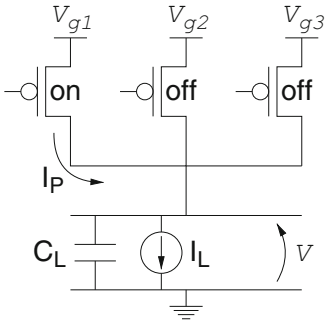

Fig. 3.4 Simplified model of a Voltage Control Unit

frequency is lower, which translates to lower power consumption in the local clock net. The power breakdown shows that the power overhead of GRLS is reasonably low even for very fine-granular systems and should be significantly better for more realistic coarse granular system.

3.2.3 Voltage Control Unit

Up to 4 global supply voltages are generated in the Central Voltage Control Unit (CVCU) using up to 4 voltage regulators and distributed throughout the chip. Because of the reduced number of voltage regulators needed, a good solution for the Central VCU is to use switching voltage regulators, very efficient but expensive if they were implemented in each island. Alternatively, the voltages can be generated off-chip. The number of supply voltages is limited to 4 because of the pin requirements for multiple Vdds and the area overhead of the multiple Vdd distribution grids.

A Voltage Control Unit (VCU) is instantiated in every node which contains power switches and the logic necessary to drive them. Power switches and multiple power rails are part of the state-of-the-art ASIC/SOC methodology and are supported by all commercial place and route tools e.g. [19]. Having four Vdd rails instead of one increases the area of the chip; a solution, consisting in distributing the Vdds in different metal layers, was proposed in [15]. The metal stripes in the local islands can be switched to one of the supply voltages or disconnected from the power supply. A simplified model of a VCU is shown in Fig. 3.4. The node is modeled as a current sink in parallel with a capacitance.

3.2.3.1 Overhead and Performance Analysis

The PMOS switches for power selection are designed as a trade-off between area occupation and performance penalty.

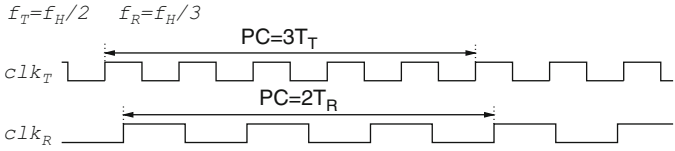

Fig. 3.5 Definition of the Periodicity Cycle *PC*

The maximum voltage drop on the single PMOS transistor that is on at any time is given, in a linearized model, by $\frac{I_P}{g_{on}}$, where I_P is the peak current draw of the node and g_{on} the conductivity of the PMOS switches, which is proportional to the number of power switches. Using the linearized alpha-model [26], the voltage drop introduces a performance penalty that translates to a reduction of the maximal working frequency from f to

$$f\left(1-\alpha\frac{I_P}{g_{on}\left(V-V_{TH}\right)}\right)$$

where α is the velocity saturation index of the technology, V is the nominal supply voltage and V_{TH} is the threshold voltage of the transistors.

Considering a 44.4 mm^2 node with a peak current of 18.48A, in 90 nm ASIC technology a 15% performance hit can be guaranteed with a 5% area overhead for each supply voltage. Other area-performance trade-offs are possible depending on the node constraints. The parameter that drives the trade-off is the ratio $R = \frac{I_P}{A}$, the peak current consumption per unit of area. R is a parameter that does not necessarily depend on the area of the node, which is roughly proportional to the number of gates that constitute it.

Once the power switches are dimensioned, the conductivity of the PMOS switches determines the time necessary for the node to change its supply voltage. SPICE simulation shows that the same 44.4 mm^2 node can change its supply voltage in ∼27 ns.

3.2.4 GRLS Transmitter and Receiver

The GRLS Transmitter and Receiver are based on the GRLS interface presented in [14]. The GRLS interface exploits a property of rationally-related frequencies, i.e. that the alignment between the clocks is the same every Periodicity Cycle *PC*, where *PC* is the least common multiple between the periods of the Transmitter clock (clk_T, running at frequency $f_T = \frac{1}{T_T}$) and the Receiver clock (clk_R, running at frequency $f_R = \frac{1}{T_R}$), as shown in Fig. 3.5. This property allows to design communication interfaces that are much more efficient compared to GALS interfaces. In literature, several attempts at designing interfaces for rationally-related frequencies can be

found [27–29]; however, the only one [27] that constitutes a truly latency-insensitive interface cannot be easily applied in a real environment because the interface is very complex and requires transistor-level design [14].

3.2.4.1 Background

All GALS communication techniques that can be found in literature [12, 30, 31] are based on the idea that the Transmitter should inform the Receiver before sending it a data item, so that the Receiver can prepare for data reception [21]. The data items can be sent only when the Transmitter knows that the Receiver is ready to accept data. This inherently involves some form of handshake and carries a high latency penalty because the analysis by the Receiver of the signals coming from the Transmitter takes time and data cannot be transmitted until the analysis is not complete.

Mesochronous communication is a subset of the GALS communication problem in which the frequencies of the Transmitter and the Receiver are perfectly matched, but their alignment is unknown [32]. There have been several mesochronous synchronizers proposed in literature which are not based on handshake [33–35]. These solutions employ a fully-synchronous Transmitter which outputs data in every clock cycle. The Receiver can sample data on both rising and falling clock edges: one of the two is guaranteed to be secure for data sampling. To know on which edge data should be sampled, a synchronization mechanism is used. The synchronization mechanism, or learning phase, can be activated once upon reset or continuously during operation.

The advantage of the learning phase approach is that the learning phase, which normally takes time to complete, determines a clock edge on which data can be sampled; when data is sent, the Receiver already knows when to sample it and data is accepted with low latency. Unfortunately, a learning phase approach cannot be used in GALS synchronizers because in a GALS communication problem no assumption can be made on the frequencies of the Transmitter and the Receiver clocks. Thanks to the periodic properties of rationally-related frequencies (the alignment between the clocks is periodic with period PC), however, it is possible to build a learning-phase-based interface for a GRLS system.

3.2.4.2 Structure

The conceptual structure of the GRLS Transmitter and Receiver are shown in Fig. 3.6.

The interface [14] is based on a synchronous Transmitter which outputs data on a subset of its rising clock edges. The clock edges on which data is output are determined by a regulation algorithm [14] which is also periodic with period PC: if a data item is sent at time t, another data item is sent at time $t + PC$. The regulation algorithm also guarantees maximal throughput, i.e. one data item is output per clock

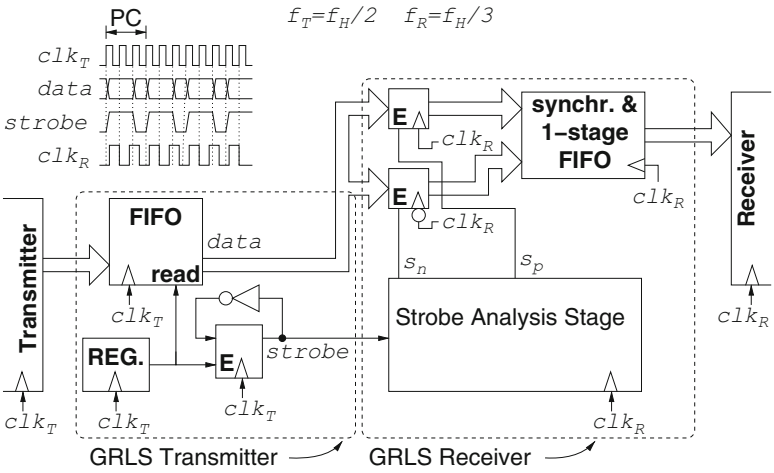

Fig. 3.6 GRLS interface and relevant signals

cycle of the slowest of the two units. A zero-waitstate FIFO buffer is used to store the data items until they can be output. If the Transmitter does not have anything to send out on a output clock edge, then it sends out a dummy data item, marked by setting an additional data line *valid* to zero. A strobe toggles between 0 and 1 every time a new data item is sent out. The strobe travels through the channel and reaches the Receiver, where it is continuously sampled on both positive and negative edges of the Receiver clock. The strobe samples are synchronized to the Receiver clock domain and analyzed. Using a specific strobe analysis mechanism [14], the Receiver detects strobe transitions, i.e. it identifies when the strobe toggled and which was the first clock edge on which the new data item that was sent out when the strobe toggled could have been safely sampled. Because of the periodic properties of rationally-related systems and the regulation algorithm, if a new data item was available to be safely sampled at time t, then a new data item will be available for safe sampling also at time $t + KPC$ with K integer. The Receiver uses this knowledge to sample the data items, i.e. it samples data items a time KPC after a strobe toggle was detected. When the Transmitter sends out a data item, it finds the Receiver ready to accept it and transmission happens with low latency. A communication example is shown in Fig. 3.7.

3.2.4.3 Overhead and Performance Analysis

The overhead of the interface is 4 flipflops per data line [14], which is equal to the overhead of state-of-the-art mesochronous communication interfaces such as STARI [32]. The strobe analysis stage of the strobe requires only 140 Gate Equivalents in 90 nm technology.

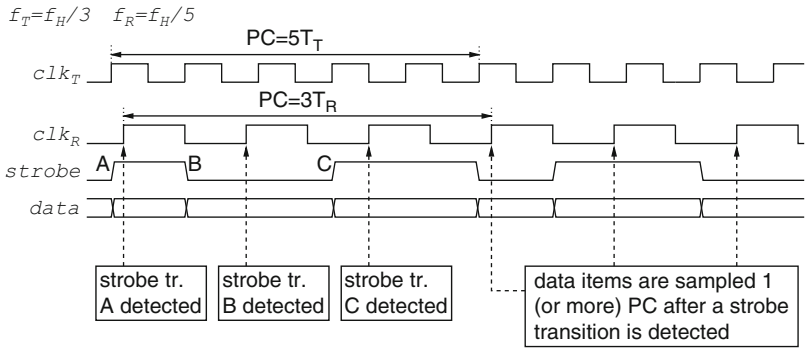

Fig. 3.7 Communication in a GRLS interface

Table 3.1 Average- and worst-case latencies for different values of $N_T = \frac{f_H}{f_T}$ and $N_R = \frac{f_H}{f_R}$, in terms of Receiver clock cycles

$N_T : N_R$	Worst-Case latency (T_R)	Average-Case latency (T_R)
$N_T = N_R$	1.000	0.500
$N_T > N_R$	1.000	0.500
2:3	1.667	0.722
2:5	1.800	0.820
3:7	1.857	0.806
5:11	1.909	0.789
4:17	1.941	0.888
12:17	1.706	0.708
16:17	1.941	0.555

The GRLS interface has ideal throughput, i.e. it allows to send out one data item per clock cycle of the slowest of the two units [14].

Because a GRLS system does not rely on global synchronicity, latency figures depend on the alignment between the clocks. Best-case, average-case and worst-case latencies can be defined. For asynchronous FIFOs [36], the most widely-accepted GALS interface [3, 37], best-case latency corresponds to 2 Receiver clock cycles, average-case latency corresponds to 2.5 clock cycles and worst-case latency corresponds to 3 clock cycles. For the GRLS interface, average- and worst-case latencies are reported in Table 3.1.

3.2.5 Power Management Unit

The Power Management Unit (PMU) acts as an access point for the power management commands issued by the Power Management INTelligence (PMINT). One PMU is inserted in every node, and can be accessed using a 32-bit command

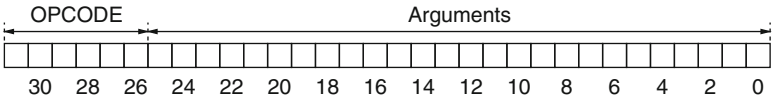

Fig. 3.8 Structure of the power management commands

register. The DME in every node knows the physical location of every PMU and encapsulates memory transactions bound for one PMU in a Nostrum packet bound for the node in which the PMU resides. Once the packet reaches the node, the GRLS Receiver on which the packet is received delivers it to the PMU. This allows the physical structure of the network to be totally transparent to PMINT. An API is provided to configure the Power Management Units and issue the power management commands.

Power Management Units (PMUs) are accessible by PMINT software and also by the other PMUs in the neighboring nodes. This is necessary for implementation of the region-wide power management commands that requires coordination between Power Management Units in different nodes. To contact a neighboring PMU, a PMU issues a memory transaction bound for the neighboring PMU, and outputs the packet on the port on which it neighbors the target PMU.

Power management commands are organized as follows (see Fig. 3.8):

- bits 31-26: OPCODE
- bits 25-0: ARGUMENTS – dependent on the command.

3.2.6 ALOIN Network

When a node is powered down, a small portion of the GRLS wrapper remains operational. This block is called ALOIN, for Always-on LOgic In Node. The ALOINs are interconnected with a NoC structure that runs parallel to the Nostrum NoC and are kept functional even when a region is powered down. ALOINs are necessary because power down modes, when applied to switches, disrupt the structure of the network and do not allow the wake-up command to take the same path as all other commands, i.e. to be encapsulated in Nostrum packets and delivered by the Nostrum network.

To justify the introduction of a new NoC running in parallel with Nostrum, the ALOINs network is kept as simple as possible. The ALOIN signals have a one-bit width, which is sufficient to implement the simple functionality of ALOIN, i.e. to wake up nodes that were powered down. Unlike multi-bit signals, single-bit signals can be safely synchronized using simple multistage synchronizers. By introducing additional synchronization stages, the MTBF (Mean Time Between Failures) of the

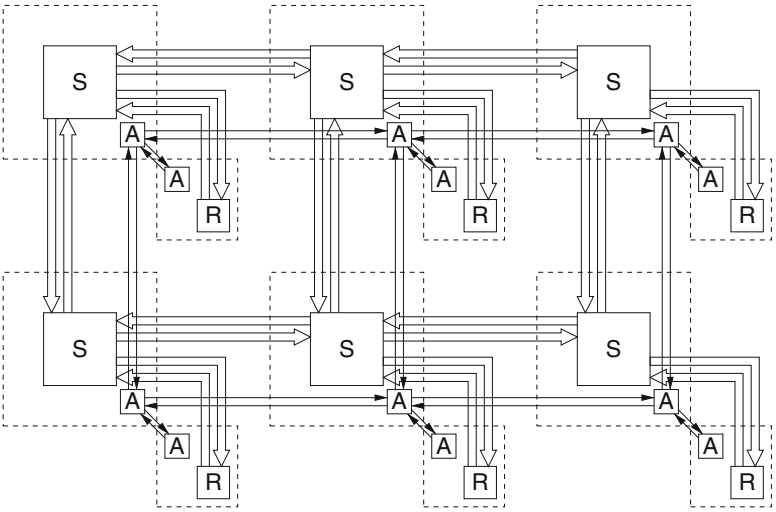

Fig. 3.9 The ALOIN network (A indicates ALOINs)

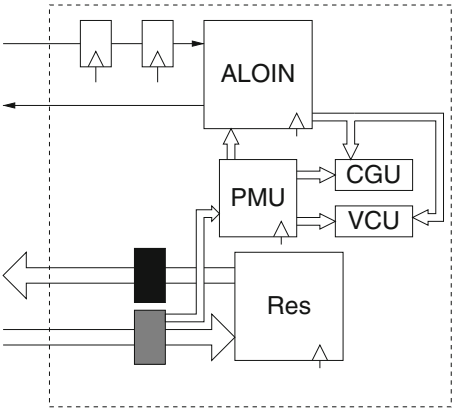

Fig. 3.10 The ALOIN block in a resource node

synchronizers can be arbitrarily increased at the expense of latency [38]. Signals that toggle too often do not propagate correctly in multi-stage synchronizers. However, because the functionality of the ALOIN network is very simple, the signals toggle only once when a node is powered down and once when it is woken up.

Figure 3.9 shows how the ALOIN network runs in parallel with the Nostrum NoC. Some details of the network, such as the synchronizers, are not shown to keep the illustration simple.

The GRLS wrapper of a resource node is shown in Fig. 3.10.

3.3 Power Management Services

The power management API of McNoC provides four power management commands which can be directly accessed by the Power Management INTelligence through the power management API. Other valid PMU commands are meant for inter-node coordination and can be issued only by other Power Management Units.

The four commands that can be accessed by PMINT are:

- SETOPTION: used to set one configuration option of the Power Management Unit. The command takes as a parameter a 6-bit value indicating the code of the configuration option, and the value to which the configuration option should be set.
- DVFSCHANGE: used to change the DVFS point of one region. The command takes as a parameter the global clock which the CGU should select and the new clock divider ratio.
- POWERDOWN: used to send to a power-down mode (clock gating, hibernation or shutdown) a region. The command takes as argument a parameter indicating the type of power-down mode (clock gating, hibernation or shutdown).
- WAKEUP: used to wake up a region from a power-down mode. Because power-down modes disrupt communication in the NoC fabric (some switches may have been powered down), the WAKEUP command is not sent to a node in the region that should be woken up, but to a node that neighbors it, and then propagated using the ALOIN network. It takes as argument the port on which the node receiving the command neighbors the region that should be woken up.

The four Power Management Commands are discussed in the following sections, along with different aspects of the functionality of McNoC.

3.3.1 SETOPTION: Power Management Unit Configuration Options

Different configuration options can be set in the Power management Unit during run time. The configuration options are set by PMINT issuing a SETOPTION command, which is organized as follows:

$$SETOPTION(ID, VALUE)$$

The parameters of the SETOPTION command are:

- ID (6 bits): configuration option identifier
- VALUE (W bits): value to which the configuration option should be set

where W is the width in bits of the configuration option.

The configuration options that can be set in the Power Management Unit are shown in Table 3.2.

Table 3.2 Configuration options for the Power Management Unit

Option	W	Usage
gclkdiv[0]	4	Specifies using 4 bits (values 1-15) the ratio between f_H and f_{g0}
gclkdiv[1]	4	Specifies using 4 bits (values 1-15) the ratio between f_H and f_{g1}
gclkdiv[2]	4	Specifies using 4 bits (values 1-15) the ratio between f_H and f_{g2}
gclkdiv[3]	4	Specifies using 4 bits (values 1-15) the ratio between f_H and f_{g3}
Vdiv[0]	8	Specifies using 8 bits (values 1-255) the minimal clock division ratio between f_H and f that can be supported by V_{g0}.
Vdiv[1]	8	Specifies using 8 bits (values 1-255) the minimal clock division ratio between f_H and f that can be supported by V_{g1}.
Vdiv[2]	8	Specifies using 8 bits (values 1-255) the minimal clock division ratio between f_H and f that can be supported by V_{g2}.
Vdiv[3]	8	Specifies using 8 bits (values 1-255) the minimal clock division ratio between f_H and f that can be supported by V_{g3}.
PortType	5	Specifies using 5 bits the type of neighboring ports (S or N). For a switch node, the five bits 4, 3, 2, 1 and 0 represent respectively: the resource port, the north port, the east port, the south port and the west port. For a resource node, bit 4 represents the switch port and all other bits are unused. An S port is indicated with 0 and an N port is indicated with 1.
RT	8	Number of cycles to which correspond the slowest round-trip to a neighboring node, i.e. the time that it takes for a packet to reach a neighboring node and for an ack to come back.
CWait	8	Number of cycles that it takes for all packets to evacuate the power management region if no packet enters it.

A PMU that receives a SETOPTION command sets the value of the configuration option indicated by ID to VALUE, then responds to PMINT with an acknowledgment after the operation is completed.

3.3.2 Power Management Regions

McNoC is organized into power management regions, which can be modified at runtime by setting the PortType configuration option. Every bi-directional port of every node in the network can be configured to be of two different types, type N or type S. S indicates that the neighboring node belongs to the same region, while N indicates that the neighboring node belongs to a different region. An example configuration is shown in Fig. 3.11. The platform is organized into three different regions:

- The first includes switch nodes $S(1,1)$, $S(1,2)$, $S(1,2)$, $S(2,3)$, and resource nodes $R(1,1)$, $R(1,2)$, $R(1,2)$, $R(2,3)$;
- The second includes switch nodes $S(2,1)$, $S(2,2)$, $S(3,1)$, $S(3,2)$, $S(3,3)$ and resource nodes $R(2,1)$, $R(2,2)$, $R(3,2)$, $R(3,3)$;
- The third includes the single resource node $R(3,1)$.

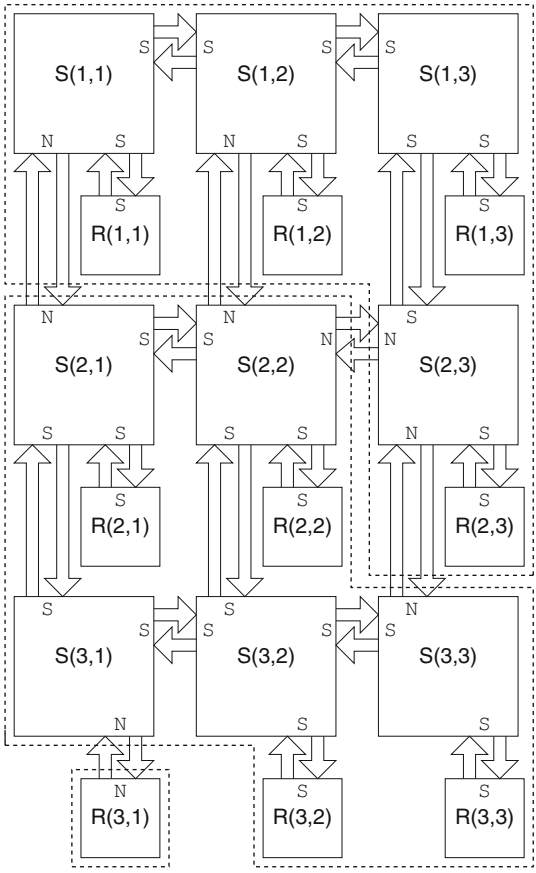

Fig. 3.11 A 3×3 platform organized into three different GRLS regions

The settings in the different nodes should be coherent: as an example, if the west port of switch node $S(2,3)$ is configured as a N port, the east port of switch node $S(2,2)$ must also be configured as a N port. Regions can take any shape, the only constraint being that all nodes in a region must be contiguous. For maximum flexibility, every node can be enclosed into its own power management region. Region-wide power management commands can be issued to any among the nodes in a region and the command is automatically propagated to all nodes that belong to it.

3.3.3 Boot Sequence

Upon reset, McNoC automatically boots in a mesochronous configuration, i.e. a configuration in which all nodes run at the same frequency. Specifically, all CGUs set $f = f_{g0}$ and all VCUs set $V = V_{g0}$. Also, all GRLS Transmitters and

Receivers are set for mesochronous communication. This solution guarantees that the system achieves immediate synchronization and that the network becomes immediately operational, which provides the basic infrastructure necessary for PMINT to configure the Power Management Units.

Different configuration options need to be set before any power management command can be issued:

- The network needs to be correctly configured in power management regions, by setting the PortType configuration option in every node.
- The options $gclkdiv[0]$ to $gclkdiv[Nclk-1]$, where $Nclk$ is the number of global clocks, must also be set to indicate the ratio between f_H and the frequencies of all the global clocks. If, for example, the first global clock runs at 200 MHz and the second global clock runs at 300 MHz, then $f_H = lcm(200MHz, 300MHz) =$ 600 MHz, $gclkdiv[0]$ should be set to 3 and $gclkdiv[1]$ should be set to 2. The values of $gclkdiv$ are expressed on 4 bits, and can take all values between 1 and 15 (the value 0 is illegal).
- Denoting as $Nvdd$ the number of global supply voltages, the options $Vdiv[0]$ to $Vdiv[Nvdd-1]$ must also be set to indicate the minimal ratio $\frac{f_H}{f}$ that can be supported by the global supply voltages. It is assumed that $V_{g0} > V_{g1} > V_{g2} > V_{g3}$. $Vdiv[3]$ expresses, on 8 bits, the minimal clock division ratio $\frac{f_H}{f}$ that is supported by voltage V_{g3}. If the ratio $\frac{f_H}{f}$ is lower than $Vdiv[3]$, then V_{g3} is too low to support operation of the node at frequency f. $Vdiv[2]$, $Vdiv[1]$ and $Vdiv[0]$ have a similar definition. Because the voltage levels are ordered, it must be $Vdiv[3] > Vdiv[2] > Vdiv[1] > Vdiv[0]$.
- RT should be set, for all nodes, as the slowest round-trip to a neighboring node expressed in terms of clock cycles. It is the responsibility of PMINT, who knows the ratio between the clocks of all nodes, to calculate a value for RT.
- $CWait$ must be set to a time interval, in terms of clock cycles, necessary for all packets in the region to reach their destinations or exit the region when the region is isolated and no packet enters it. Again, it is the responsibility of PMINT to set the value of $CWait$ according to the size and characteristics of the region.

Once all these steps have been concluded, the system is fully-configured and can accept power management commands from PMINT.

3.3.4 DVFSCHANGE Command: Changing the DVFS Point

3.3.4.1 Interface

A DVFSCHANGE command has the following structure:

$$DVFSCHANGE(SEL, GDIV)$$

The parameters are the following:

- SEL (2 bits): Id of the global clock to select
- GDIV (4 bits): Clock divider ratio

PMINT sends the DVFSCHANGE command to any node in the region for which it wants to change the DVFS point. The command takes several cycles to execute. Upon completion an acknowledgment is sent back to PMINT from the node that received the request. Upon completion, the clocks of all nodes in the region will be obtained by dividing the frequency of clock clk_{gSEL} by the dividing factor $GDIV$: $f = \frac{f_{gSEL}}{GDIV}$. The voltage of all the nodes in the region will be set to the lowest V_{gi} which can support operation at frequency f, according to the $Vdiv[i]$ configuration options. All nodes in the region will communicate mesochronously. The GRLS Receivers and Transmitters in the nodes neighboring the region that changed its DVFS point will also be programmed so that they can continue to communicate with the region once its DVFS point has changed. No packet will be lost while the DVFS point is changed.

3.3.4.2 Internals

Once a DVFSCHANGE command is received by a Power Management Unit in a node, the node immediately stops to send data items to all neighbors by disabling output in all GRLS Transmitters.

The node also informs all neighbors about the change in DVFS point by sending a DVFSCHANGE-N command to all neighbors. The node then waits RT cycles. After RT cycles, all neighbors will have received the DVFSCHANGE-N command and will also have stopped to send data items to the node. Then, the node calculates the new ratio $\frac{f_H}{f} = gclkdiv[SEL] * GDIV$ based on the selected global clock and the clock dividing ratio. Based on $\frac{f_H}{f}$ and the values of $Vdiv$, the lowest global supply voltage V_{gL} that can tolerate operation at frequency f is determined. Commands are issued to the CGU to change the global selected clock and the dividing ratio, and to the VCU to change the voltage to $V = V_{gL}$. The GRLS Transmitters and Receivers are informed of the change in frequency. Then, the unit waits again RT cycles before restarting to send packets to all neighbors and sends an acknowledgment back to PMINT to inform it about the successful completion of the operation.

Nodes receiving a DVFSCHANGE-N command from another node residing in the same power management region propagate a DVFSCHANGE-N command to all neighboring nodes and then initiate a similar course of action, with the difference that no acknowledgment is sent to PMINT. If the command is received from a node that resides in a different power management region, then the node stops sending packets to the node from which it received the command; it then updates its GRLS Transmitter and Receiver based on the new frequency of its neighbor; finally, it waits $2RT$ cycles before restarting to send data to the neighboring node, which by then will have changed its DVFS point.

3.3.5 POWERDOWN Command: Power Down Modes

3.3.5.1 Interface

The POWERDOWN command has the following structure:

$$POWERDOWN(MODE)$$

The parameters are the following:

- MODE (2 bits): Power-down mode identifier (0: clock gating; 1: hibernation; 2: shutdown)

When a POWERDOWN command is received by a Power Management Unit, the PMU sends immediately an acknowledgment back to PMINT. The command takes several cycles to execute. Upon completion of the operation, all nodes in the GRLS region will be powered down, i.e. they will have a gated clock; the supply voltage in the node will remain unchanged (with $MODE = 0$), set to the lowest global supply voltage (with $MODE = 1$), or disconnected from the global power supply (with $MODE = 2$). PMU, GRLS Transmitters and Receivers and switch/resource will be powered down, the only elements remaining active being the ALOIN, the CGU and the VCU. All the packets that were moving inside the region when the command was first received will have been consumed by the resources or evacuated from the region. The switches neighboring the powered-down region will consider the ports going to the powered-down region as border ports, i.e. they will feed the output port on the input port.

3.3.5.2 Internals

Once a POWERDOWN command is received by a Power Management Unit in a node, the node immediately sends an acknowledgment back to PMINT. It then sends a POWERDOWN-N command to all neighboring nodes. Nodes receiving a POWERDOWN-N command from a node in the same power management region propagate the POWERDOWN-N command to all neighbors. Nodes receiving a POWERDOWN-N command from a node in a different region stop sending data items to the region that is going to be powered down. The region which is going to be powered down remains thus isolated from the rest of the network. If the network was configured correctly, in CWait cycles all packets traveling in the region will have been consumed or will have left it.

After *Cwait* cycles, the node that received the POWERDOWN command sends a POWERDOWN-S command to all neighboring nodes. Again, the POWERDOWN-S command is propagated from node to node until it reaches all nodes in the region and all nodes neighboring it. If a node receives a POWERDOWN-S command from

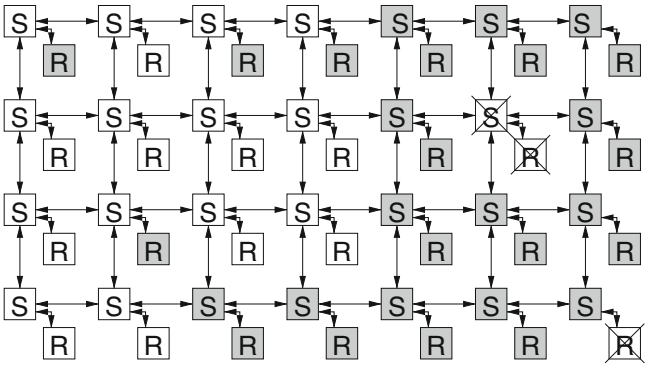

Fig. 3.12 Incorrect partitioning of the network in powered-down and awake regions

a node belonging to a different region, then the node closes the loop on that port, i.e. it feeds the output port to the corresponding input port and resumes data output on the port.

The original node that received the POWERDOWN command and all the nodes that received the POWERDOWN-S command from a node in the same region gate the clock and then fix the voltage to the level determined by MODE. The ALOIN, which also contains all the configuration options of the PMU, remains operative.

3.3.6 Operation with Regions Powered Down

Powering down a region may divide the platform into multiple regions of contiguous non-powered-down nodes, called awake regions. Communication between different awake regions is not possible. As long as only resources are powered down, a single awake region exists. If switches are also powered down, then the network structure of the platform is disrupted. The Nostrum switches, based on X-Y routing, require that all awake regions remain rectangular in shape.

For this reason, when powering down switch nodes, it is necessary for PMINT to follow some guidelines to ensure correct operation. In particular:

- All resource nodes can be powered down without restrictions.
- Switch nodes can be powered down as long as the awake regions remain rectangular in shape and all nodes in any region communicate only with nodes within the awake region in which they belong.

As an example, Fig. 3.12 shows an incorrect configuration of the network. A non-rectangular awake region is present, and the two crossed-out nodes cannot contact or be contacted by any other resource node.

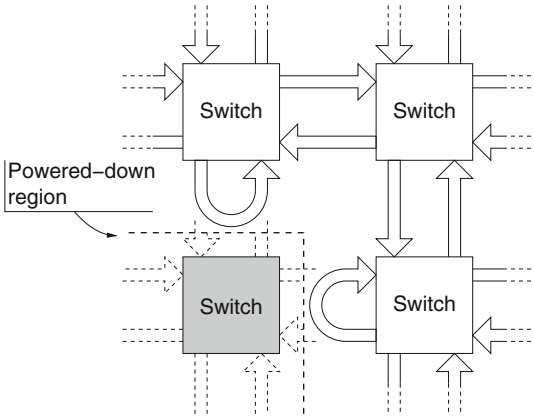

Fig. 3.13 Loop-closing on the border of a powered-down region

To avoid the loss of misrouted packets, McNoC closes the outgoing links going to a powered-down region on the node itself, as is done with nodes on the border of a network (see Fig. 3.13).

3.3.7 WAKEUP: Waking Up From a Power-Down State

3.3.7.1 Interface

The WAKEUP command is distinguished from the other commands because it is not sent to a node inside the region that is woken up, but instead to a node that neighbors it.

The WAKEUP command has the following structure:

$$WAKEUP(PORT)$$

The parameters are the following:

- PORT (5 bits): Id of the port on which the WAKEUP command should be implemented

When a WAKEUP command is received by a Power Management Unit, the PMU instructs the ALOIN, the Always-on-LOgic In Node, to wake up the region neighboring it on the port indicated by PORT: for a switch node, $PORT = 10000$ indicates the resource port, $PORT = 01000$ indicates the north port, $PORT = 00100$ indicates the east port, $PORT = 00010$ indicates the south port and $PORT = 00001$ indicates the west port; all other values are illegal. For a resource node, $PORT = 10000$ indicates the switch port and all other values are illegal. The command takes several cycles to execute. Upon completion, all nodes in the region that neighbors the node on the specified port are restored to awake state, i.e. the clocks are restored

to the frequencies that they had before the region was powered down and the supply voltages are restored to the values they had before the region was powered down. Communication from the neighboring nodes to the region, which was interrupted when the region was powered down, is re-allowed. An acknowledgment is sent back to PMINT.

3.3.7.2 Internals

When a region is powered down, the ALOINs of all nodes in the region are sent to sleep. An ALOIN in a sleep state sets all its outputs to zero. When an awake ALOIN sees an input port going to zero, it sets the corresponding output port to zero. *RT* cycles after an ALOIN went to sleep, it is ready to be woken up. The ALOIN is woken up when it sees one of its input ports going to one. When an ALOIN wakes up, it sets all its output ports to one, so that the wakeup command is propagated throughout the region. It also wakes up the CGU and the VCU. The PMU is woken up and sends a WAKINGUP command to all neighboring nodes. Nodes receiving a WAKINGUP command from nodes in a different region restart communication with the region that was powered down.

3.4 Conclusion

In conclusion, the power management architecture of McNoC enables hierarchical physical design by relying on rationally-related frequencies, thus avoiding the overhead associated with handshake. Quantized voltage scaling is realized by distributing multiple Vdd throughout the chip and allowing every node to select one of the Vdds as supply voltage. Arbitrary power management regions can be defined at configuration time or at run time. region-wide DVFS point change, power down and wakeup services are provided.

References

1. International Technology Roadmap for Semiconductors Report, 2009
2. J. M. Rabaey, “Digital Integrated Circuits: A Design Perspective,” Prentice Hall, 1995
3. A. P. Niranjan and P. Wiscombe, “Islands of synchronicity, a design methodology for SoC design,” Design, Automation and Test in Europe Conference and Exhibition, 2004
4. S. Herbert and D. Marculescu, “Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors,” ISLPED 2007
5. P. Teehan et al., “A Survey and Taxonomy of GALS Design Styles,” Design & Test of Computers, IEEE , vol.24, no.5, pp.418-428, Sept.-Oct. 2007
6. Nostrum home page - http://www.ict.kth.se/nostrum

7. E. Nilsson, “Design and implementation of a hot-potato switch in a network on chip,” Master’s thesis, Department of Microelectronics and Information Technology, KTH, 2002
8. A. Hemani et al., “Lowering power consumption in clock by using globally asynchronous locally synchronous design style,” Design Automation Conference, 1999
9. I. E. Sutherland and J. Ebergen, “Computers Without Clocks,” Scientific American, Aug. 2002
10. S. Borkar, “Does asynchronous logic design really have a future?,” EE Times, 2003
11. D. M. Chapiro, “Globally Asynchronous Locally-Synchronous Systems,” PhD thesis, Stanford University, Oct. 1984
12. K. Y. Yun and R. P. Donohue, “Pausible clocking: a first step toward heterogeneous systems,” IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1996
13. J. Muttersbach et al., “Practical design of globally-asynchronous locally-synchronous systems,” International Symposium on Advanced Research in Asynchronous Circuits and Systems, 2000
14. J. M. Chabloz and A. Hemani, “A Flexible Interface for Rationally-Related Frequencies,” ICCD 2009
15. J. M. Chabloz and A. Hemani, “Distributed DVFS with Rationally-Related Frequencies and Quantized Voltage Levels,” ISLPED 2010
16. L. H. Chandrasena et al., “An Energy Efficient Rate Selection Algorithm for Voltage Quantized Dynamic Voltage Scaling,” ISSS 2001
17. M. Putic et al., “Panoptic DVS: A Fine-Grained Dynamic Voltage Scaling Framework for Energy Scalable CMOS Design,” ICCD 2009
18. E. Beigne et al. “Dynamic Voltage and Frequency Scaling Architecture for Units Integration within a GALS NoC,” NOCS 2008
19. Cadence SoC Encounter User Guide
20. V. Gutnik and A. Chandrakasan, “Embedded power supply for low-power DSP,” in IEEE Transactions on VLSI Systems, 1997
21. J. M. Chabloz and A. Hemani, “Lowering the Latency of Interfaces for Rationally-Related Frequencies,” ICCD 2010
22. I. Miro Panades et al., “Physical Implementation of the DSPIN Network-on-Chip in the FAUST Architecture,” NoCS 2008
23. D. Kim et al., “Asynchronous FIFO Interfaces for GALS On-Chip Switched Networks,” International SoC Design Conference, 2005
24. G. Liang and A. Jantsch, “Adaptive Power Management for the On-Chip Communication Network,” Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on
25. S. R. Vangal et al., “An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS,” IEEE Journal of Solid-State Circuits, vol.43, no.1, Jan. 2008
26. T. Sakurai and A. R. Newton, “Alpha-Power Law MOSFET Model and its Applications to CMOS Inverter Delay and Other Formulas,” IEEE J. of solid-state circuits, 1990
27. A. Chakraborty and M. R. Greenstreet, “Efficient self-timed interfaces for crossing clock domains,” International Symposium on Asynchronous Systems and Circuits, 2003
28. J. Mekie et al., “Interface Design for Rationally Clocked GALS Systems,” International Symposium on Asynchronous Systems and Circuits, 2006
29. L. F. G. Sarmenta, “Synchronous Communication Techniques for Rationally Clocked Systems,” Master’s thesis, MIT, 1995
30. J. Carlsson et al., “A Clock Gating Circuit for Globally Asynchronous Locally Synchronous Systems,” Norchip Conference, 2006
31. E. Amini et al., “Globally asynchronous locally synchronous wrapper circuit based on clock gating,” Symposium on Emerging VLSI Technologies and Architectures, 2006
32. M. R. Greenstreet, “Implementing a STARI chip,” International Conference on Computer Design, 1995
33. F. Mu and C. Svensson, “Self-tested self-synchronization circuit for mesochronous clocking,” IEEE Transactions on Analog and Digital Signal Processing, vol.48, no.2, pp.129-140, Feb. 2001

34. D. Mangano et al., “Skew Insensitive Physical Links for Network on Chip,” 1st International Conference on Nano-Networks and Workshops, Sep. 2006
35. I. Loi et al., “Developing Mesochronous Synchronizers to Enable 3D NoCs,” DATE, 2008
36. C. E. Cummings and P. Alfke, “Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons,” Synopsys Users Group Conference, 2002
37. N. Wingen, “What If You Could Design Tomorrow’s System Today?,” Design, Automation & Test in Europe Conference & Exhibition, 2007
38. R. Ginosar, “Fourteen ways to fool your synchronizer,” International Symposium on Asynchronous Systems and Circuits, 2003

Chapter 4
ASIP Exploration and Design

Jari Kreku, Kari Tiensyrjä, Andreas Wieferink, and Bart Vanthournout

Abstract ASIP exploration uses the mappability method for the selection of processor core and algorithm combinations for multi-core designs. The mappability estimation is based on the analysis of the correlations of algorithm and core characteristics. This information is used for narrowing the exploration space of the subsequent ASIP design that exploits commercial ASIP design environment, Synopsys Processor Designer. According to simulation results the proposed ASIPs are able to achieve up to 96% of maximum performance with a clear reduction in complexity.

4.1 Introduction

The requirements of embedded processors vary a lot depending on the targeted application. Therefore specialised processors for particular application domains have been developed. In addition to specific functional requirements they can be designed for fulfilling various non-functional requirements like performance, power and cost. Due to the well-defined and limited scope of embedded systems, designing an optimised processor (ASIP) can reduce production costs and increase flexibility and efficiency.

J. Kreku • K. Tiensyrjä (✉)
VTT Technical Research Centre of Finland, Kaitoväylä 1 FI-90570 Oulu, Finland
e-mail: jari.kreku@vtt.fi; kari.tiensyrja@vtt.fi

A. Wieferink
Synopsys, Team4 Building, Kaiserstrasse 100 D-52134 Herzogenrath, Germany
e-mail: andreasw@synopsys.com

B. Vanthournout
Synopsys, Interleuvenlaan 15A B-3001 Leuven, Belgium
e-mail: bartv@synopsys.com

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_4,

The development of ASIPs is a complex task that requires profound knowledge of both the application at hand and the design of processor architectures. This often leads to an iterative design process to find an optimal ASIP configuration to a set of representative application algorithms. Typically a set of design alternatives are evaluated to obtain relevant performance, power etc. data.

The typical alternatives for evaluating the mapping of an algorithm to an architecture have been static estimation, cosimulation or execution. Chen et al. [1] have examined retargetable static timing analysis techniques of embedded software. Software performance estimation techniques in [3, 12] try to approximate the amount of required computation and the resulting minimum execution time. Lazarescu et al. [7] uses the number of instructions derived from source code as a basis for SW estimation. In [6] a method for evaluation performance using benchmark sets and performance vectors is presented. Retargetable estimation scheme by [2] uses parameterised architecture models for studying compiler-independent architecture potential. In [9] neural networks and cycle-accurate training simulations are used to estimate performance. Marcon et al. [8] considers mapping of applications to homogeneous NoCs from the communication point of view.

In ASIP exploration, depicted in Fig. 4.1, the algorithms allocated to each ASIP and the properties of that ASIP are extensively cross-studied in order to define architectural parameters for the ASIP so that the execution efficiency of the algorithms on the ASIP is maximised. The key factors affecting the efficiency of an algorithm–processor architecture mapping are performance and resource utilisation. The processor has to be able to execute the required computation quickly enough to meet the timing requirements. On the other hand, the mapping is unsuccessful if the algorithms or programs can not exploit the provided resources efficiently, leading to increased cost and power consumption.

Core and Algorithm Mappability Analysis Approach (CAMALA) is a method for calculating the mappability estimate of a processor architecture and algorithm pair ([5, 10, 11]). Mappability denotes the degree of matching between the resources provided by the processor core and requirements described by the algorithm. Mappability estimation can be used for the evaluation or selection of existing processor cores for the execution of an algorithm (and vice versa), or identification of "optimal" architecture to be implemented as a custom processor. It does not require that the implementations of the algorithm or processor exist.

The mappability estimation approach tries to alleviate the processor or algorithm selection problem by taking into account both aspects of efficiency. Instead of predicting the performance in clock cycles, it tries to indicate when the number of resources is right so that utilisation is maximised and adding more resources would not improve the performance.

Early evaluation of the efficiency of the algorithm-processor mapping requires that the algorithm and processor models are straightforward to create directly from the specifications. Moreover, the evaluation approach should not require execution or simulation of execution.

With mappability estimation it is possible to evaluate which ASIP out of several will be the most suitable for the execution of the algorithm or application, with

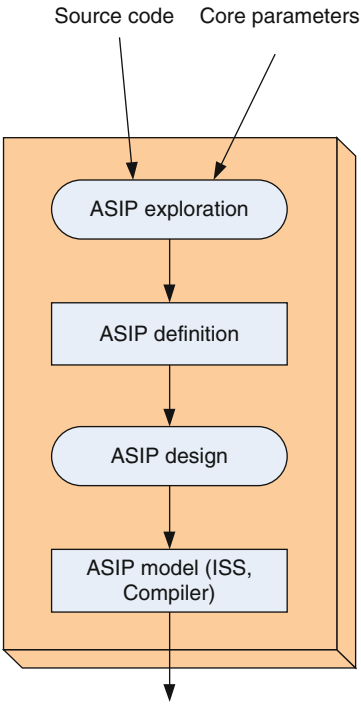

Fig. 4.1 MOSART design flow at the ASIP exploration level

respect to the properties of the processor core. The properties include super-pipelining degree, number of execution units and number of registers, for example. Besides the calculation of the mappability estimate for a set of processor–algorithm pairs, the approach can be used to estimate the optimal values for the processor core properties. It can also reveal, which intermediate language operations are used in the algorithm and how often, which is beneficial for designing the instruction set extensions for the ASIP.

The information of the processor properties and instruction set is provided to the designer, who can take it into account in ASIP design with the Synopsys Processor Designer. The possibilities for transferring this information from CAMALA to Synopsys Processor Designer are considered in Sect. 4.2.3.1. The ASIP exploration can be done iteratively so that the changes implemented with the Processor Designer are optionally fed back to mappability estimation or system level exploration for a new round of evaluations.

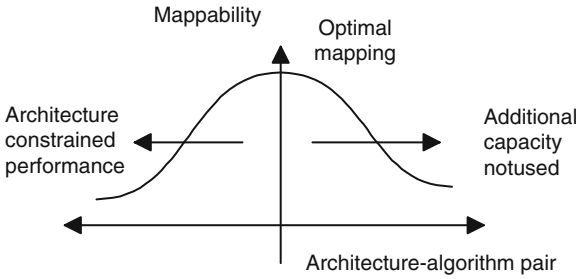

Fig. 4.2 Mappability function

4.2 Mappability Estimation

The figure of merit proposed for architecture-algorithm pair is mappability, e.g. $M=(c,a)$, where c is core architecture and a is algorithm. Mappability is optimal when the hardware architecture does not constrain the execution and it does not have any unused capacity, as depicted in Fig. 4.2. In an ideal case all the hardware in the core is participating in the execution of code all the time, but additional resources cannot be used because of the nature of the code.

Instruction capability, instruction execution speed and dynamic parallelism are core characteristics that affect performance. Instruction capability depends on instruction set, internal registers and memory interface. Execution speed depends on superpipelining degree, pipelining support and instruction types. Dynamic parallelism depends on execution architecture and the core's capability to utilize the potential in the algorithm's control flow.

The computation constraints from the algorithm are caused by its control flow structure and data dependencies. The control flow determines the executed program paths. It depends on input data or events. Data dependencies constrain the order and the amount of parallelism in which the operations can be executed.

CAMALA extracts essential characteristics of the core and algorithm and analyses how much correlation exists between the algorithm's computation requirements, e.g. data dependencies and control flow structure, and core properties, e.g. instruction set, execution architecture and memory interface. The algorithm characterization is based on compilation, profiling and bound checking. The result is an intermediate representation graph. The core model consists of the instruction models described using weighted data dependency graphs, and interface and architecture parameters.

The algorithm a can be modeled as a control flow graph $A=(V,G)$, where nodes $v_j \in V$ are basic blocks (BB) and arcs $g_j \in G$ are branches. Each BB v_j has an execution weight w_j and its operation can be modeled as a data dependency graph $W_j=(O_j,U_j)$. In W the nodes $o_k \in O_j$ are primitive operations of target independent representation and arcs $u_k \in U_j$ are data dependencies. Each arc u_k has a label describing its data type dt_k. Each arc $g_j \in G$ has a branching probability P_j.

Table 4.1 Parameters of the processor core model

Execution architecture parameter	Value range
Branch prediction technique P_T	none, static or dynamic
Dynamic prediction efficiency P_e	0–100%
Pipeline depth D_p	$1-N$
Superpipelining degree D_s	$1-N$
Number of execution paths E	$1-N$
Number of registers R	$1-N$
Data buses B_D	$0-N$
Program bus B_P	yes or no
Combined data/program buses B_C	$0-N$
Bus width B_w	$1-N$
Floating point cost factor C_f	$1-N$
Word length cost factor C_w	$1-N$

The core c consists of instructions $i_j \in I$ and execution architecture parameters (Table 4.1). Each instruction i_j is able to replace one or more operations o_k and has an execution and implementation cost $C_{\text{ex,j}}$ and $C_{\text{im,j}}$. If the core is not capable of implementing every primitive operation in O, then virtual instructions are added to I to model the required subroutines.

Correlation is calculated by obtaining the number of a core resources e_c of type R from the processor model and estimating the algorithms need for such resources e_a from the algorithm model. The difference between the available resources and resource requirements is calculated for each basic block at a time and the weighted standard deviation formula is used to generalise the results for the entire algorithm. Here, e_c acts as the mean value and e_a of each basic block as the sample value. The CAMALA tool uses the rapid standard deviation calculation method to calculate correlation iteratively one basic block at a time:

First the running sum Ω_i of basic block weights w_i is calculated:

$$\Omega_0 = 0 \tag{4.1}$$

$$\Omega_i = \Omega_{i-1} + w_i \tag{4.2}$$

The ordering of the basic blocks is not significant. Next the running sums of mean A_i and squared difference Q_i are calculated:

$$A_0 = 0 \tag{4.3}$$

$$A_i = A_{i-1} + \frac{w_i}{\Omega_i}(e_{a,i} - A_{i-1}) \tag{4.4}$$

$$Q_0 = 0 \tag{4.5}$$

$$Q_i = Q_{i-1} + w_i(e_{a,i} - e_c)^2 \tag{4.6}$$

where $e_{a,i}$ is the algorithm's resource need in basic block i. Correlation after n basic blocks (deviation of basic blocks' optimal resource requirements from the number of resources in the core) is then

$$c = c_{fn}(w_i, e_{a,i}, e_c) = \sqrt{\frac{Q_n}{\Omega_n}} \tag{4.7}$$

and the value for the core property, which would produce optimal correlation

$$e_{a,opt} = A_n \tag{4.8}$$

Good mappability between an algorithm and a processor core requires that the instruction set is suitable for the required computation, the execution architecture supports the logical and effective ordering of operations and the data is available when needed. In order to manage the complexity of estimation, we have divided the correlation problem into seven orthogonal parts (Sect. 4.2.1). Experiments with a large set of algorithms and cores indicate that the correlation values have a log normal distribution. The overall mappability is calculated as a sum of the correlations:

$$M = \sum e^{\frac{\ln c_i - \mu_i}{\sigma_i}} \tag{4.9}$$

where μ_i and σ_i are the mean and standard deviation of correlation i's natural logarithm respectively.

4.2.1 Correlations

The calculation of the mappability estimate for an algorithm-processor pair has been divided into seven independent parts, which evaluate the problem from different points of view. The points of view include the relationship between algorithm's operations and processor's instruction set, exploitation of parallelism in the algorithm with pipelining and execution units, control flow, and data availability.

4.2.1.1 Instruction Set Effectiveness Correlation

The instruction set effectiveness (ISE) correlation depends on how effectively the core's instruction set can be used for the given algorithm from the performance point of view. In the algorithm model the graphs $W_j = (O_j, U_j)$ will be replaced with the graphs $W_j^{'} = (O_j^{''} U_j^{'})$ by replacing the primitive operations $o \in O_j$ with implemented instructions $i \in I_j$. The procedure is basically similar to what happens in a compiler's back-end. The data types and available accuracy needs to be checked. If they do not match, the cost of instructions are multiplied with floating point cost factor C_f or word length cost factor C_w, respectively.

Unlike the other correlations, ISE correlation is calculated for the entire algorithm at a time. First, the reference cost C_0 of the algorithm is calculated by assuming that each operation in the algorithm will be performed in one clock cycle:

$$C_0 = O_{tot} \tag{4.10}$$

where O_{tot} is the total number of operations in the algorithm. Next, the operations are replaced with the core's instructions and the cost with core's instructions C_I is calculated. The ISE correlation is then calculated using the following formula:

$$c_{ISE} = \ln \frac{100 C_I}{C_0} \tag{4.11}$$

4.2.1.2 Instruction Set Coverage Correlation

The instruction set coverage (ISC) correlation analyses, how extensively the processor core's instruction set can be used for the execution of the algorithm. Instruction set coverage correlation is performed after instruction set effectiveness correlation and thus it uses the modified algorithm model, where target-independent operations have been replaced with core's instructions. First, the relative implementation $I_{r,i}$ cost of each non-virtual instruction i is calculated:

$$I_{r,i} = \frac{I_i}{I_{tot}} \tag{4.12}$$

where I_i is the implementation cost of instruction i given in the core model and I_{tot} is the sum of the costs of all instructions. Next, the probability P_i of the instruction, i.e. how commonly the instruction has been used in the modified algorithm model, is calculated:

$$P_i = \frac{N_i}{N_{tot}} \tag{4.13}$$

where N_i is the number of times instruction i has been used and N_{tot} is the total number of instructions.

ISC correlation considers that instructions with large implementation cost must be used more often than the others to justify their existence in the processor core. Thus, using the correlation calculation method given in Sect. 5.1 iteratively for each instruction we get

$$c_{ISC} = c_{fn}(1, I_{r,i}, P_i) \tag{4.14}$$

which gives the optimal correlation when each instruction's probability equals its relative implementation cost.

4.2.1.3 Internal Data Availability Correlation

The internal data availability (IDA) correlation expresses how effectively registers can be used. In program execution the registers store intermediate results and frequently-used operands. Knowing how many registers can be used requires that we know the number of intermediate results $r(a)$ during algorithm execution. In the algorithm model the intermediate results are the arcs v in data dependency graphs W, so for each node we calculate the maximum number of arcs between two scheduling step using ASAP and ALAP schedules, so $e(a) \sim \max |U_i|$. Because the schedules are not constrained by resources, they express the extent to which parallelism and registers can be exploited.

The nodes belonging to a loop must be combined to one node for this correlation. The external dependencies inside the loop are considered internal dependencies and the new node weight is the smallest weight of the nodes belonging to the loop. The correlation for one node and number of registers can be calculated iteratively using the correlation function, where $e(c) \sim R$ is the number of registers in the core model:

$$c_{IDA} = c_{fn}(W_i, \max |U_i|, R) \tag{4.15}$$

4.2.1.4 External Data Availability Correlation

The external data availability (EDA) correlation deals with bus efficiency. The bus usage should correlate to the bus capacity. The number of instruction bus operations is assumed to be equal to the number of executed instructions, i_i. The number of data operations is estimated using external dependencies in w and operations without successors. It is assumed that external dependencies outside a loop require bus operations because the program flow is non-deterministic. The results of the operations without successors must be written into memory because the whole operations are useless otherwise. The estimated algorithm characteristic is then the bus usage, e.g. $e(a) \sim i_i + i_r + i_w$. The estimated processor characteristic is the bus capacity, e.g. $e(c) \sim B$, which depends on the number of buses and bus width in bits.

For the Von Neumann architecture the EDA correlation is defined as follows:

$$c_{EDA} = c_{fn}(W_i, i_i + i_r + i_w, B_i) \tag{4.16}$$

where B_i is the capacity of the combined instruction and data bus. For the Harvard architecture instruction and data buses are handled separately:

$$c_{EDA} = c_{fn}(W_i, i_i, B_i) + c_{fn}(W_i, i_r + i_w, B_d) \tag{4.17}$$

where B_i and B_d are the capacities of the instruction and data buses respectively.

4.2.1.5 Control Flow Continuity Correlation

The control flow continuity (CFC) correlation depends on the number of branch instructions and pipeline depth. The branch instruction ratio B is calculated by dividing the number of branch instructions $|i_b|$ by the number of all instructions $|I_j|$ in a program path:

$$B = \frac{|i_b|}{|I_j|} \tag{4.18}$$

Branch prediction reduces the number of executed branch instructions and branch penalties. If static branch prediction is used, it is assumed that the compiler can optimize all branches correctly and, if dynamic prediction is used, the value must be given as a parameter P_e. Effective branch instruction ratio B_{eff} is calculated by subtracting the correctly predicted branches from B, thus:

$$B_{eff} = (1 - P_e)B \tag{4.19}$$

The core characteristic in estimation is the pipeline depth, $e(c) \sim D$.

The best mappability value is achieved when there is a long pipeline and few branches, because the execution of instructions can be overlapped effectively. If there is a short pipeline and a lot of branches, the branch penalties are small. If there is a short pipeline and few branches, we do not exploit all the overlapping possibilities of the algorithm, and if there is a long pipeline and lot of branches, the overlapping benefits are wasted because of branch penalties.

The CFC correlation analyses the entire algorithm at a time like the ISE correlation. The CFC correlation begins with the estimation of the optimal pipeline depth D_{opt} for the algorithm. Assuming no prediction is used this is given by

$$D_{opt,no-pred} = \frac{1}{kB} - 1 \tag{4.20}$$

and with static or dynamic prediction

$$D_{opt,pred} = \frac{1}{kB_{eff}} - 1 \tag{4.21}$$

where k is a scaling factor (default $k = 9$).

If the core supports branch prediction, the CFC correlation is given by

$$c_{CFC} = c_{fn}(1, D_{opt,no-pred}, D) \tag{4.22}$$

Otherwise the following formula is used:

$$c_{CFC} = c_{fn}(1, D_{opt,no-pred}, D) + c_{fn}(2, D_{opt,pred}, D) \tag{4.23}$$

to avoid situations, where the optimal correlation would always be given by pipeline depth of 1 without branch prediction in the core.

4.2.1.6 Data Flow Continuity Correlation

In the data flow continuity (DFC) correlation the idea is that if the execution order of instructions is fixed by data dependencies, it will cause data hazards and degrade the pipeline efficiency. The degree of data dependency can be estimated by analyzing the number of instructions in a schedule step in unconstrained ASAP or ALAP schedule of w_j. The instructions that can be scheduled on the same step can fill the pipeline without data hazards. The topology of w and bypassing support of the processor has an effect on the mobility of instructions, d, and a higher mobility makes it easier to exploit the pipeline more efficiently. The estimated algorithm characteristic is then $e(a) \sim d\bar{i}_i$, where $\bar{i}_i$ is average number of instructions in a scheduling step. The core characteristic is the superpipelining degree, $e(c) \sim D_s$. Thus,

$$c_{DFC} = c_{fn}(W_i, d\bar{i}_i, D_s) \tag{4.24}$$

4.2.1.7 Execution Unit Availability Correlation

The execution unit availability (EUA) correlation compares operation level parallelism in an algorithm to the number of parallel execution units. The parallelism of algorithm is constrained by the data dependencies and it can be studied by dividing the number of instructions in W by the number of steps in the shortest possible schedule, so $e(a) = \bar{i}_j$.

The parallelism of the architecture is constrained by the available execution units in each parallel execution path. The parallel execution units in the core are, typically, not alike and all the execution units cannot execute all the instructions. We estimate the number of parallel execution units by calculating the coverage of execution unit k by dividing the possible instruction/unit by all instructions/core and then adjusting the number of execution paths E with this value, i.e. $e(c) \sim kE$. The value of EUA correlation is calculated iteratively using the following formula:

$$c_{EUA} = c_{fn}(W_i, \bar{i}_j, kE) \tag{4.25}$$

4.2.2 Algorithm Modeling Front-End

The algorithm models can be written manually, but in complex cases it is rather cumbersome. Therefore, a GNU Compiler Collection (GCC) 4.3.1 based front-end has been developed to create models automatically from source code. It extends GCC with an additional compilation pass *pass_camala* (Fig. 4.3), which can be enabled with a single compilation flag. Pass_camala is located approximately in the middle of all the passes, between pass_dcc and pass_update_address_taken in the current implementation. At this phase, GCC is still using the high-level

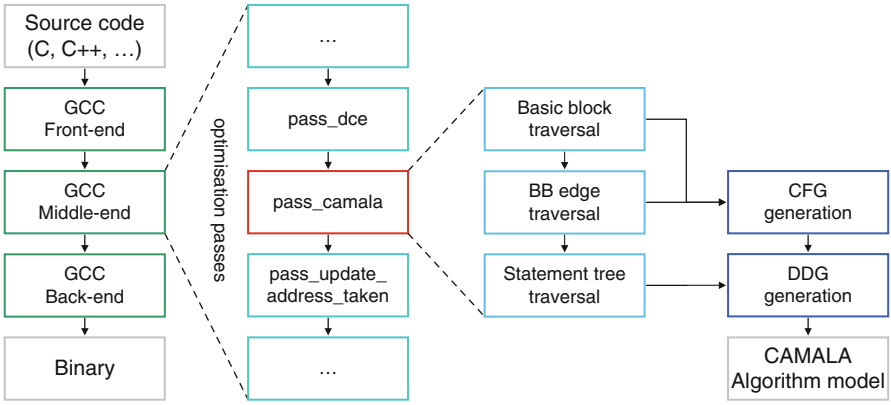

Fig. 4.3 Block diagram of the CAMALA front-end implemented by extending GCC

GIMPLE intermediate language and generating the models there will not introduce dependences on a particular target architecture. The front-end should be able to generate workload models from any language supported by GCC. However, only C has been tested in experiments.

CAMALA algorithm models contain weights for control flow nodes and probabilities for branches. GCC can obtain these if code profiling is used. Thus, the model generation flow includes the following three phases: (1) First the source code is compiled with profiling, (2) then the compiled binary is executed with a suitable dataset, and (3) finally, the source code is recompiled with profile-guided optimisation and model generation enabled.

The pass_camala extracts the control flow by traversing the BBs and BB edges in the GCC's intermediate representation and generates the control flow graph (CFG) for the algorithm model accordingly (Fig. 4.3). The format for the generated nodes is the following:

```
node_j {
  weight = wj;
  # Successors (0 - N)
  target = node_g1;
  prob = P1;
  ...
  # DDG nodes
  ...
}
```

where `node_j` is a unique identifier for the CFG node. For each BB the generator will extract the number of times the block has been executed from the profiling data and use it as the node weight `wj`. It will also obtain the identifiers of successor nodes `node_g1, node_g2, ...` by walking through the BB exit edges and the probabilities of branching to those nodes `P1, P2, ...` from the profiling data.

Table 4.2 CAMALA operation set for data dependence graphs

Class	Operations
arithmetic	$x+y$, $x-y$, $x\cdot y$, x/y, $-x$, $\lvert x\rvert$, $\min(x,y)$, $\max(x,y)$
logical	$x\wedge y$, $x\vee y$, $x\oplus y$, $\neg x$
bitwise	$x\wedge y$, $x\vee y$, $x\oplus y$, $\neg x$
shift	left or right shift or rotate x by y
comparison	$x<y$, $x\leq y$, $x>y$, $x\geq y$, $x=y$, $x\neq y$
other	branch, call, no-op

In addition, pass_camala traverses all GIMPLE statements within each basic block to generate a data dependence graph (DDG) of the contents of the block. The nodes in the DDG represent intermediate language operations and have the following format:

```
r_type r_precision op_id op_type(arguments);
```

where `r_type` and `r_precision` denote the result data type (`int`, `float`) and precision (in bits) of the operation respectively. `op_id` is a unique identifier, which is obtained with GCC's `DECL_NAME()` macro whenever possible. Otherwise, a unique temporary id is generated. `op_type` corresponds to the type of the operation and is one of the types listed in Table 4.2. Within GCC the data and operation types can be obtained by examining the tree expression with `TREE_CODE()` macro and precision with `TYPE_SIZE()` macro.

`arguments` is a comma-separated list of the dependences of the node. Each argument has the following format:

```
dep_type dep_precision dep_id
```

where `dep_type` and `dep_precision` are optional and denote the data type and precision of the dependence. `dep_id` is the dependence identifier and must correspond to the `op_id` of predecessor node. However, `dep_type` and `dep_precision` may override the result type and precision of the predecessor node. Call operations have an additional argument for the name of the function.

4.2.3 CAMALA Tool

A command-line CAMALA tool has been implemented with C++ for mappability estimation. It expects the algorithm model in its own textual format, which resembles the C programming language to a degree. The properties of the processor core are given in a text file, which describes the instructions supported by the core, instruction parameters (e.g. latency), and core parameters (Table 4.1).

After the mappability estimation has been completed, CAMALA will present the following information to the user:

- Optimal values for processor core properties per each correlator, which are estimated from the algorithm model. Depending on the correlator and verbosity level, one or more intermediate values used in the calculation of the estimate can be displayed.
- Processor core properties obtained from the core parameter file.
- Values for each correlation, which are obtained by comparing core properties to the corresponding algorithm properties. A value of zero represents high correlation (optimal match), whereas the higher the correlation value is the worse is the match.
- Overall mappability estimate, which is a weighted average of the correlations.

4.2.3.1 Interface to Synopsys Processor Designer

CAMALA is also able to estimate the optimal selection of instructions for the execution units of the processor core. First, the optimal number of parallel execution units (i.e. VLIW slots) is obtained from the EUA correlation. Next, the ISA correlation needs to be performed to analyse, how often each instruction implemented in the core can be used for the execution of the algorithm. More commonly used instructions are given more slots and vice versa.

Initial guess N_i of the number of slots for instruction i is given by

$$N_i = N_{opt} I P_i \tag{4.26}$$

where N_{opt} is the optimal number of execution units, I is the number of instructions in the core and P_i is the probability of instruction i:

$$P_i = \frac{U_i}{U_{all}} \tag{4.27}$$

where U_i is the number of times instruction i can be used in the execution of the algorithm and U_{all} is the total number of instructions required by the algorithm. Thus the average number of slots given to instructions equals the optimal number of execution units given by the EUA correlation.

Next, N_i is limited so that there is at least 1 slot for all instructions regardless whether they are used by the algorithm or not. There is also an upper bound for the number of slots, N_{max}, which by default is 6:

$$N_i = \begin{cases} 1 & N_i < 1 \\ N_i & 1 \leq N_i \leq N_{max} \\ N_{max} & N_i > N_{max} \end{cases} \tag{4.28}$$

After limitation the number of slots allocated for instructions is adjusted by increasing or decreasing them evenly. This is done in order to keep the average at the optimal level, since limitation may decrease or increase the number of slots. Difference from the optimal D is:

$$D = N_{avg} - N_{opt} \tag{4.29}$$

where N_{avg} is the average number of slots after limitation. Correction C is:

$$C = \begin{cases} \frac{DI}{I - I_u} & D > 0 \\ \frac{DI}{I - I_o} & D < 0 \end{cases} \tag{4.30}$$

where I_u is the number of instructions below 1 slot and I_o is number of instructions above N_{max} slots. Thus we get:

$$\hat{N}_i = N_i - C \tag{4.31}$$

Finally, the number of slots is rounded to the nearest integer, which is still within the limits.

There are two different methods for allocating the slots: simple and balanced. In the simple allocation the first execution unit always contains all instructions, the second execution unit contains instructions with at least two slots, and so on until unit N contains instructions with N slots. In balanced allocation the instructions are allocated to slots starting from the most common instruction. The least used slot will be selected from the slots available to the instruction. The list of available slots is limited by interdependences between instructions, e.g. instruction i_0 may use partially or completely the same hardware as instruction i_1 and thus needs to use the same slots.

4.3 Synopsys Processor Designer

4.3.1 The Commercial Tool: Processor Designer

Processor Designer is used to develop a wide range of processor architectures, including architectures with DSP-specific and RISC-specific features as well as SIMD and VLIW architectures. The design flow is depicted in Fig. 4.4. Processor Designer's generated software development environment enables the commencement of application software development prior to silicon availability, thus eradicating a common bottleneck in embedded system development. The key to Processor Designer's automation is its Language for Instruction Set Architectures, LISA 2.0. In contrast to SystemC, which has been developed for efficient specification of systems, LISA 2.0 is a processor description language that incorporates all necessary processor-specific components such as register files, pipelines, pins, memory and caches, and instructions. It enables the efficient creation of a single golden processor specification as the source for the automatic generation of the instruction set simulator (ISS) and the complete suite of software development tools, like Assembler, Linker, Archiver and C-Compiler, and synthesizable RTL code. The development tools, together with the extensive profiling capabilities of the debugger, enable rapid analysis and exploration of the application-specific processor's instruction set architecture to determine the optimal instruction set

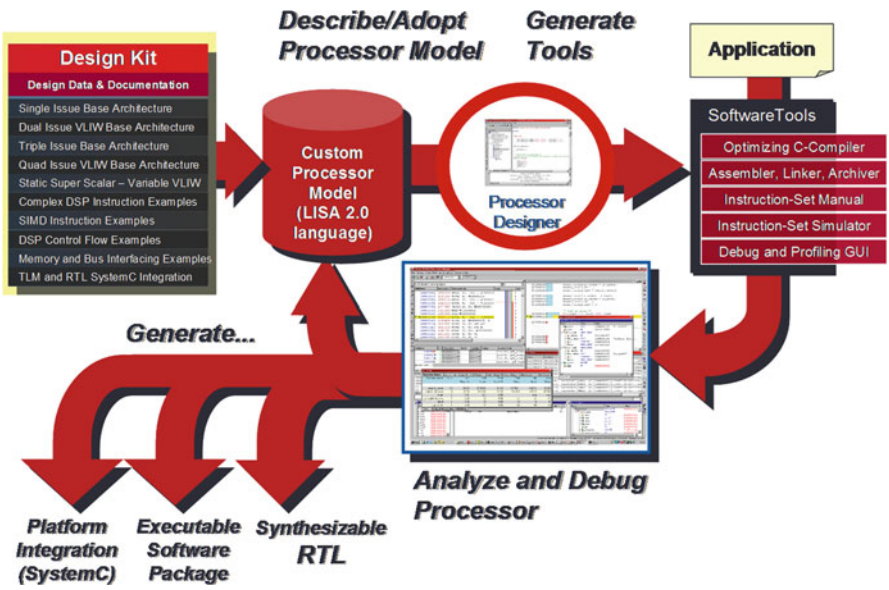

Fig. 4.4 Synopsys processor design flow

for the target application domain. Processor Designer enables the designer to optimize instruction set design, processor micro-architecture and memory subsystems, including caches.

4.3.2 The Extension: Processor Designer Starter Kit

The Processor Designer framework provides flexibility to develop highly optimized ASIPs, respectively programmable accelerators. However, this flexibility comes at the cost of high development effort, especially when the processors are developed from scratch.

In order to ease this effort, a starter kit has been created, which contains a set of fully functional processor architecture templates. The basic architecture template only provides what is absolutely necessary to provide a fully working set of software development tools including C-compiler and cycle accurate instruction set simulator.

By compiling the target application for this basic processor architecture, the user can immediately start with a functionally correct simulation. Profiling the application already executing on the target architecture should then point the designer the spots which are promising candidates for optimization.

There are several independent approaches that can significantly increase performance, which can of course be combined. The template kit contains examples for these kinds of extensions:

- Instruction-set extensions: Fusion of multiple (simple) instructions into a highly specialized complex instruction. Typically, additional functional units are allocated to execute these instructions.
- Special purpose registers: Highly optimized functional units need to read and write their data efficiently. Additional customized register banks can be allocated, e.g. to hold 4×4 blocks of pixels for an H.264 video encoder.
- Data parallelism (SIMD): A huge set of data is processed in parallel by a vector type functional unit. In an H.264 video encoder, an instruction could perform the same operation on a block of 16 pixels simultaneously. This type of processing is often called SIMD (Single Instruction Multiple Data).
- Instruction-level parallelism, software pipelining (VLIW): Subsequent operations should be executed in parallel wherever possible. For example, the next burst of input data should already be loaded from memory while the current burst is still computed. For embedded applications the parallel instructions are typically scheduled by the compiler. This kind of architecture is also often referred to as VLIW (Very Long Instruction Word).
- Pipelining: By pipelining processor instructions, the data-path of a functional unit can be distributed over multiple pipeline stages. This breaks up the critical path of the functional unit as it results in faster combinatorial logic.

4.3.3 Using CAMALA as Front-End

Even when simplifying the ASIP design by using the PD kit, there is still a huge design space left which cannot be explored exhaustively. It makes sense to pre-explore the ASIP design space on a higher abstraction level in order to detect promising ASIP configurations that are worthwhile to focus on.

For some key ASIP parameters, even an automated transfer of CAMALA configuration proposals into Processor Designer is possible. As already mentioned in Sect. 4.2.3.1, the ASIP's VLIW configuration is such a key property.

VLIW processor architectures are a powerful vehicle for meeting future performance requirements. It is plausible that design space exploration concerning the most optimal number and population of VLIW slots has a huge impact on overall performance and chip area of a processor. The PD starter kit contains a rich collection of VLIW processor templates. The following design options are implemented independently from each other in different LISA model files:

- Number of VLIW slots (1 . . . 6)
- Variable length instruction encoding (yes / no)
- Synchronous Memory (yes / no)

- Extra instructions (none, Multiply, MAC, ABS, SIMD-ABS,..., and combinations of those)
- Exact population of available VLIW slots with instructions (many permutations possible).

The final configuration option is the exact population of available VLIW slots with instructions. All current template kit configurations are written such that they just allocate all known computational instructions in all slots. This means that chip area may increase significantly, because a maximum number of functional units need to be instantiated on the chip. However, it is most likely not necessary to allocate expensive units like a multiplier once for each VLIW slot. Thus, an important design parameter is the VLIW slot configuration: Which instructions should be available in which slot, and how many slots are required?

As indicated in Sect. 4.2, CAMALA is able to analyze the target application on an abstract level, and it can estimate to which degree instruction level parallelism is possible. From that, CAMALA then proposes promising configurations for the VLIW slots. The idea is that these configuration proposals are automatically imported by the Synopsys environment and further processed in order to select and configure a LISA processor template.

The text file format that has been agreed on is looking like this (simplified):

```
#define VLIW_MAX_SLOTS 4
#define GROUP_SLOT_0 alu_arithmetic<0> || alu_logic<0> || smul<0>
#define GROUP_SLOT_1 alu_arithmetic<1> || alu_logic<1> || smul<1>
#define GROUP_SLOT_2 alu_arithmetic<2>
#define GROUP_SLOT_3 alu_arithmetic<3>
```

This example would allocate the basic arithmetic instructions in all 4 of the 4 requested VLIW slots, whereas the basic logic instructions as well as a multiply instruction are only allocated in 2 of the 4 slots.

The syntax is already in the correct format as expected for LISA models to define the coding root, which is the central instance that spans the operation tree. The file contains preprocessor defines, because for a LISA template with variable length coding, the same slot definitions may be required multiple times throughout the LISA code.

Unfortunately, the traditional way of setting up the coding tree for a LISA model does not match the way it is required for a CAMALA coupling. As depicted in Fig. 4.5, it is usually the signature of the instruction encoding that is defined at the top level of the operation tree. This way, instructions with the same operand footprint share the same sub-branch. For example, the coding tree branch below alu_rri contains all instructions that operate on two register references and require one immediate value. Only in lower nodes of the operation tree, the branches split according to functional differences. The advantage of this approach is a higher modelling efficiency especially for processor architectures having a very regular instruction encoding. However, this approach is not suitable to optionally instantiate instruction branches according to their functionality, as required for the CAMALA

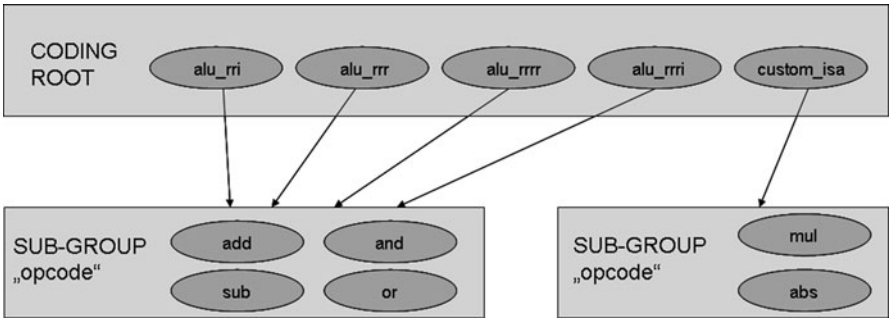

Fig. 4.5 "Traditional" organisation of a LISA model

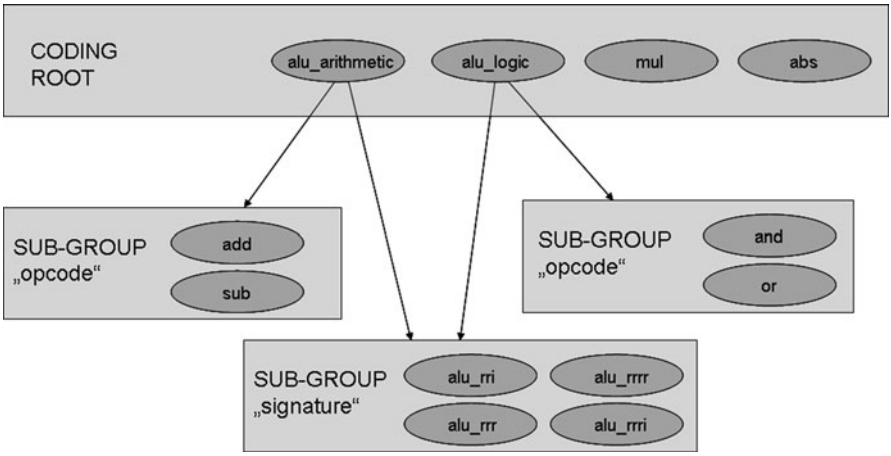

Fig. 4.6 "Functional" organisation of a LISA model

coupling. In the traditional operation tree, it would only be possible to remove or add instruction groups according to their encoding, which is not useful.

That is why all VLIW templates in the library need to be organized as an operation structure shown in Fig. 4.6. Only in this way it is possible to optionally remove entire sub-branches according to instruction functionality rather than their encoding.

This VLIW slot configuration information can now be provided by CAMALA in a few lines of text file.

Table 4.3 Processor core parameters

	Value	
Parameter	PdKit-RISC	PdKit-VLIW
Branch prediction	none	
Pipeline depth	5	
Instruction and data buses	32-bit instruction & data bus	
General purpose registers	16 32-bit registers	
Execution units	1	4
Instruction set	ALU arithmetic, ALU logic, compare, shift, fused shift logic, fused shift arithmetic, smul, abs, mac	

4.4 Case Example

4.4.1 Algorithm and Core Models

MiBench ([4]) is a freely available benchmark suite which consists of several applications in the fields of automotive control, networking, security, consumer devices, office automation, and telecommunication.

The algorithm model generator was used to create models of several MiBench benchmark applications: as a result, one model was obtained per each function in each automotive benchmark. The larger option of the MiBench data sets was selected for each application. Two processor core models were created manually (Table 4.3): They model two variants of the PdKit ASIP template of the Synopsys Processor Designer. The first one is a singe unit RISC, whereas the second one has four parallel execution units in a VLIW configuration.

The applications were first compiled with profiling code. Next, they were executed to obtain the branch probabilities and execution weights for the control flow from the profiling data. Finally, the applications were recompiled with profile-guided optimisation and model generation. As a result, one algorithm model was created per each function in each automotive benchmark.

4.4.2 Mappability Estimation

Table 4.4 shows the overall mappability for all benchmarks. A value of zero would indicate optimal correlation between the properties of the algorithm and the processor core.

It seems that the higher data processing capacity of the VLIW is favourable in only a few benchmarks: Rijndael, Rsynth, SHA, CRC32 and the encoding part of GSM. However, when the mappability estimate of the RISC is better than that of the VLIW, the difference is quite small. The only exception here is the Lame

Table 4.4 Mappability estimates for MiBench benchmarks and PdKit cores

			Mappability	
Set	Benchmark	Algorithms	PdKit-RISC	PdKit-VLIW
Automotive	Basicmath	3	3.41	3.65
	Bitcount	8	4.10	4.31
	Quicksort	2	17.28	17.55
Consumer	Lame	211	7.59	12.39
Network	Dijkstra	3	3.99	4.32
Office	Ghostscript	3774	5.60	5.79
	Rsynth	77	6.33	6.20
Security	Rijndael	7	36.35	34.81
	SHA	7	6.33	3.68
Telecomm	CRC32	2	3.17	2.97
	FFT	6	2.60	2.69
	GSM encode	77	39.86	14.09
	GSM decode	77	4.53	4.90

benchmark. On the other hand, the VLIW is clearly better in SHA and GSM encoding.

Based on the estimates, neither of the cores is really suitable for the Quicksort, Rijndael and GSM encoding benchmarks. Quicksort consists of mostly comparison operations and thus the ISC correlation is high. The same applies to the Gsm_Long_Term_Predictor() function in GSM encoding. Furthermore, the function Gsm_RPE_Encoding() has high IDA and EDA. The cores have too many registers and not enough bus capacity for this algorithm. The ISC correlation is not bad, but a smaller instruction set consisting of comparison, shift, multiplication and arithmetic instructions would suffice. In the Rijndael benchmark the decrypt () function would benefit from more registers and it could also utilise a longer pipeline.

The cores fare much better with Basicmath, Bitcount, Dijkstra, CRC32 and FFT. Basicmath and Bitcount are rather simple benchmarks, which match the properties of the PdKit cores well in all respects. The cores obtain low correlation values for everything except the dijkstra () function in Dijkstra. The ISC correlation is higher than the rest there due to the limited operation set consisting of mostly comparisons and arithmetic. The CRC32 and FFT require a closer inspection: In FFT, most of the execution time is spent in the C library functions sin () and cos (), but these functions were not compiled during the compilation of the benchmark. Thus the model generator did not generate models for them. In CRC32 the same happens with getc (). As a result the mappability estimation for the two benchmarks is dominated by their main() functions, which are not too complex and therefore suitable for the PdKit cores. It would be possible to generate models of the sin (), cos () and getc () functions by compiling the C library source code. However, it was not done for this case example.

The correctness of the mappability estimation approach was not measured in this case, but the results are in-line with the expectations based on the nature of

the algorithms and processor core properties. Earlier versions of correlators have been compared to measurements from real implementations in [5, 11]. CAMALA was able to detect the best matrix multiplication algorithm for two DSP cores and effectively predicted the best variant of ARM architecture for several MiBench benchmarks in those experiments.

4.4.3 VLIW Slot Configuration

The VLIW slot configuration with CAMALA was evaluated with simulations using the Basicmath, Bitcount and Quicksort benchmarks. CAMALA was used to propose two configurations for each benchmark: a normal and a 25% more parallel alternative. The Processor Designer was used to create these ASIPs and, in addition, several more generic variations of the PdKit template with 1 to 6 execution units. Finally, the benchmarks were simulated with the ASIPs and the execution time was measured.

The ASIPs and execution times of the benchmarks are shown in Table 4.5. The ASIPs in the table are named *N*i-full,alu,noalu where *N* is the total number of parallel execution units and the prefix denotes, whether there are *N* slots for all instructions (full), only for ALU instructions (alu), or only for non-ALU instructions (noalu). The table presents the number of slots for the different instruction groups for each ASIP core.

The execution time is displayed in millions of cycles for all the benchmarks. It is likely that the maximum attainable clock frequency would vary between the ASIPs and that the less complex variants could reach higher frequencies. However, the clock frequencies were not considered in this comparison.

In addition, there is a simplified complexity estimate C_e, which is calculated as follows:

$$C_e = S + 5S_m \tag{4.32}$$

where S_m is the number of slots for the multiplier and S is the number of slots for other instruction groups.

The normal ASIP proposed by CAMALA for the Basicmath benchmark achieved 95% of the performance of the most complex ASIP with 6 units. On the other hand it has only 22% of the complexity of that highest-performing ASIP. The 25% more parallel proposal achieved 96% of the performance with 37% of the complexity. However, from the comparison ASIP set, the 2i-full is able to achieve 98% performance with 35% complexity.

In Bitcount, the normal CAMALA proposal is able to achieve a 95% level of performance with 24% of the complexity. The 25% more parallel alternative reaches 99% and 31% respectively. This is clearly the best obtained performance for the number of resources.

Finally, with Quicksort we are able to see that CAMALA has proposed adding more compare units to the core since comparisons form the bulk of the benchmark.

Table 4.5 Mappability estimates for MiBench benchmarks and PdKit cores

	Execution units						Execution time (M cycles)		
Core	ALU arithmetic	ALU logic	Comp.	Smul	Abs	Complex.	Basic-math	Bit-count	Quick-sort
6i-full	6	6	6	6	6	49	12247	44	178
4i-full	4	4	4	4	4	33	12264	44	178
4i-noalu	1	1	4	4	4	27	14499	55	201
4i-alu	4	4	1	1	1	15	12384	45	180
2i-full	2	2	2	2	2	17	12508	46	180
2i-noalu	1	1	2	2	2	15	14600	55	201
2i-alu	2	2	1	1	1	11	13012	47	198
1i	1	1	1	1	1	9	15211	57	220
Basicmath	3	1	1	1	1	11	12887		
Basicmath +25	4	1	2	2	1	18	12786		
Bitcount	3	1	2	1	1	12		46	
Bitcount +25	4	2	3	1	1	15		44	
Quicksort	1	1	3	1	1	11			200
Quicksort +25	1	1	5	1	1	13			200

However, it is obvious from the simulation results that the compiler is not able to utilise the additional units, since the number of cycles for both CAMALA proposals is the same. They are able to achieve 89% of the maximum performance with 22% and 27% of the resources. This is still a respectable performance with low complexity, even if the 4i-alu from the comparison set gets 99% of the performance with barely more resources (31%).

The specialised ASIPs proposed by CAMALA would probably perform worse if used to execute one of the other benchmarks, e.g. if the Quicksort core was used with the Basicmath or Bitcount benchmarks. One option could be to utilise several specialised cores, one for each different application at the cost of increased complexity. However, CAMALA is able to propose a single ASIP for a set consisting of any number $(1 - N)$ of applications in the same way it was used with sole MiBench benchmarks. The resulting ASIP would be less optimised for any single application and more optimised for the common calculations in all the applications.

4.5 Conclusions

This chapter described the method and tool approach for doing ASIP exploration within the MOSART design flow. It consists of two parts: CAMALA by VTT is aimed at algorithm-core mappability estimation as the front-end. It allows the

designer to explore extensively algorithms and processor architectures in order to select the best candidates for more profound studies in Synopsys Processor Designer, where the actual design and model/tool generation for the ASIP is done. Currently CAMALA is able to estimate mappability between an algorithm or a set of algorithms and a single processor core. An extension for the estimation multi-core systems consisting of several applications mapped to a set of processor cores has been developed and will be tested in the future.

References

1. K. Chen, S. Malik, and D. I. August. Retargetable static timing analysis for embedded software. In *Proceedings of the 14th International Symposium on System Synthesis*, pages 39–44, 2001.
2. N. Ghazal, R. Newton, and J. Rabaye. Retargetable estimation scheme for dsp architectures. In *Asia and South Pacific Design Automation Conference*, pages 485–489, 2000.
3. J. Gong, D. Gajski, and A. Nicolau. Performance evaluation for application-specific architectures. *IEEE Transactions on VLSI*, 3(4):483–490, December 1995.
4. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, December 2001.
5. J. Kreku and J.-P. Soininen. Mappability estimate: A measure of the goodness of a processor-algorithm pair. In *International Symposium on System-on-Chip Proceedings*, pages 119–122, Tampere, Finland, November 2003.
6. U. Krishnaswamy and I. D. Scherson. A framework for computer performance evaluation using benchmark sets. *IEEE Transactions on Computers*, 49(12):1325–1338, December 2000.
7. M. Lazarescu, J. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *Proceedings of High Level Design Validation and Test Workshop*, pages 167–172, November 2000.
8. C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner. Time and energy efficient mapping of embedded applications onto nocs. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 33–38, January 2005.
9. M. Oyamada, F. R. Wagner, M. Bonaciu, W. Cesario, and A. Jerraya. Software performance estimation in mpsoc design. In *Asia and South Pacific Design Automation Conference*, pages 38–43, January 2007.
10. J.-P. Soininen, J. Kreku, Y. Qu, and M. Forsel. Fast processor core selection for wlan modem using mappability estimation. In *Proceedings of the 10th International Symposium on Hardware-Software Codesign (CODES)*, pages 61–66, Estes Park, Colorado, 2002.
11. J.-P. Soininen, J. Kreku, Y. Qu, and M. Forsell. Mappability estimation approach for processor architecture evaluation. In *Proceedings of the 20th IEEE Norchip Conference*, pages 171–176, 2002.
12. K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *33rd Design Automation Conference*, pages 605–610, 1996.

Part II
System-level Exploration

Chapter 5
System Exploration

Jari Kreku and Kari Tiensyrjä

Abstract Future embedded system products, e.g. smart handheld mobile terminals, will accommodate a large number of applications that will partly run sequentially and independently, partly concurrently and interacting on massively parallel computing platforms. Already for systems of moderate complexity, the design space will be huge and its exploration requires that the system architect is able to quickly evaluate the performances of candidate architectures and application mappings. The mainstream evaluation technique today is the system-level performance simulation of the applications and platforms using abstracted workload and processing capacity models, respectively. These virtual system models allow fast simulation of large systems at an early phase of development with reasonable modelling effort and time. The accuracy of the performance results is dependent on how closely the models used reflect the actual system. This chapter gives a description of the ABSOLUT modelling and simulation approach. Firstly, it gives an outline view of the approach and its evolution. Secondly, it describes how to create different models. Thirdly, it describes the means for simulation.

5.1 Introduction

Both the application and platform designers are facing an abundant number of design alternatives and need systematic approaches for the exploration of the design space. Efficient methods and tools for early system-level performance analysis are necessary to avoid wrong decisions at the critical stage of system development.

According to [11] performance evaluation methods can be classified to three main classes: analytical methods, simulation methods and monitoring methods.

J. Kreku • K. Tiensyrjä (✉)
VTT Technical Research Centre of Finland, Kaitoväylä 1 FI-90570 Oulu, Finland
e-mail: jari.kreku@vtt.fi; kari.tiensyrja@vtt.fi

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_5,

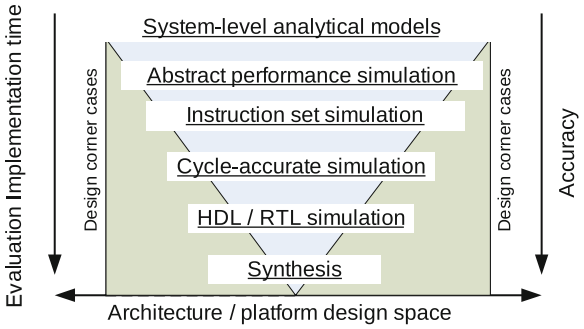

Fig. 5.1 Trade-offs of different evaluation methods

Analytical performance modelling is typical in early phases of design and methods are based on mathematical models of the workload and the system architecture. Markov chains, queuing models and Petri-nets are typical examples of analytical modelling techniques. Analyses are normally based on solving the equations, but also simulation is used as a supporting tool.

In performance simulation, the execution of a workload with a model of execution platform is simulated. Workload modelling can be based on several alternatives, e.g. executable programs (real application or benchmark programs), execution traces of programs and stochastic models. Execution platform modelling can be based on e.g. abstract resource capacity models, or virtual platform models where instruction-set simulators are used to simulate programmable processors.

Monitoring/measurement based approaches need working prototypes of hardware. The prototype is instrumented to gather performance information during the execution of the software.

Performance analysis models are required to capture both the characteristics of the application functionality and the architectural resources needed for the execution. Although giving accurate results, creating such models at too low level of abstraction, e.g. Register-Transfer-Level (RTL) or Instruction-Set-Simulation (ISS), is not feasible due to the vast amount of details needed, heavy modelling effort and long simulation times. Some high-level abstraction approaches like Queuing Networks (QN) and its variants fail to exhibit the characteristics of the execution platforms.

Performance evaluation has been approached in many ways at different levels of refinement. The trade-offs involved by choosing an appropriate evaluation method are shown in Fig. 5.1 (see [9]). Analytical models allow a fast evaluation of a relatively large fraction of the design space, enabling the identification of corner cases of the design. Over several possible steps of refinement with increasing effort for implementation and evaluation the design space can be bound to one particular design point.

SPADE ([22]) implements a trace-driven, system-level co-simulation of application and architecture. The application is described by Kahn process networks using

YAPI ([4]). Symbolic instruction traces generated by the application are interpreted by architecture models to reveal timing behaviour. Abstract, instruction-accurate performance models are used for describing architectures.

The Artemis work by [28] extends the work described in [22] by introducing the concept of virtual processors and bounded buffers. One drawback of restricting the designer to using Kahn process networks is the inability to model time-dependent behaviour. In the developed Sesame modelling methodology a designer first selects candidate architectures using analytical modelling and multi-objective optimization. The system-level simulation environment allows for architectural exploration at different levels of abstraction while maintaining high-level and architecture-independent application specifications by applying dataflow graphs in its intermediate mapping layer. These dataflow graphs take care of the run-time transformation of coarse-grained application-level events into finer grained architecture-level events that drive the architecture model components.

The basic principle of the TAPES performance evaluation approach in [31] is to abstract the involved functionalities by processing latencies and to cover only the interaction of the associated sub-functions on the architecture, represented as inter-SoC-module transactions, without actually running the corresponding program code. This abstraction enables higher simulation speed than an annotated, fully-fledged functional model. Each sub-function is captured as a sequence of transactions, also referred to as trace. The binding decision for the sub-functions is considered by storing the corresponding trace in the respective architectural resource. A resource may contain several traces, one per each sub-function that is bound to it. The application is then simulated by forwarding packet references through the system and triggering the traces that are required for processing particular data packets in the respective SoC modules.

MESH [27] looks at resources (hardware building blocks), software, and schedulers/protocols as three abstraction levels that are modelled by software threads on the evaluation host. Hardware is represented by continuously activated, rate-based threads, whereas threads for software and schedulers have no guaranteed activation patterns. The software threads contain annotations describing the hardware requirements, so-called time budgets that are arbitrated by scheduler threads. Software time budgets are derived beforehand by estimation or profiling. The resolution of a time budget is a design parameter and can vary from single compute cycles to task-level periods. The advance of simulation time is driven by the periodic hardware threads. The scheduler threads synchronize the available time budgets with the requirements of the software threads.

SpecC ([7]) defines a methodology for system design including architecture exploration, communication synthesis, validation, and implementation. These phases must mainly be carried out manually as there are no tools that could automatically perform them. Therefore, SpecC can be considered more as a specification and modelling language that has a rich support for many system design phases. Similar properties can be found also in SystemC language that is more widely adopted in high-level system modelling. Especially, the transaction-level modelling using

SystemC has been adopted for performance modelling and simulation ([8]), and OSCI has published the version 2.0 of its SystemC Transaction-Level Modeling standard.

Posadas et al. [29] presents a C++ library for timing estimation at system level. The library is based on a general and systematic methodology that takes as input the original SystemC source code without any modification and provides the estimation parameters by simply including the library within a usual simulation. As a consequence, the same models of computation used during system design are preserved and all simulation conditions are maintained. The method exploits the advantages of dynamic analysis, that is, easy management of unpredictable data dependent conditions and computational efficiency compared with other alternatives (ISS or RTL simulation), without the need for software (SW) generation, compilation and hardware (HW) synthesis.

Koski ([12]) is an automated SoC design methodology focusing on abstract modelling of application and architecture for early architecture exploration, methods to generate the models from the original design entry, system-level architecture exploration performing automatically allocation and mapping, tool chain supporting the defined methodology utilizing a graphical user interface, well-defined tool interfaces, a common intermediate format, and a simulation tool that combines abstract application and architecture models for co-simulation.

5.2 ABSOLUT System Modelling

The performance modelling and evaluation approach of VTT's ABSOLUT as used within the MOSART methodology follows the Y-chart model as depicted in Fig. 5.2 ([14, 15]).

The layered hierarchical workload models represent the computation and communication loads the applications cause on the platform when executed. The layered hierarchical platform models represent the computation and communication capacities the platform offers to the applications. The workload models are mapped onto the platform models and the resulting system model is simulated at transaction-level to obtain performance data.

ABSOLUT, depicted in Fig. 5.3, is a model-based approach for system-level design that is capable of performance evaluation of future real-time embedded systems and provides early information for development decisions.

The earlier SystemC-based workload-platform performance modelling and simulation approach in [17, 18] was extended in [19] by introducing layering and hierarchy to both the workload and platform models. This enables combining the top-down refinement type application modelling and bottom-up composition type platform modelling. Both are based on service-oriented approach with defined service interfaces ([13]), which brings scalability to both of the sides. The approach takes service orientation into focus, i.e. applications are modelled in terms of services they deliver to the user and request from the execution platform, and

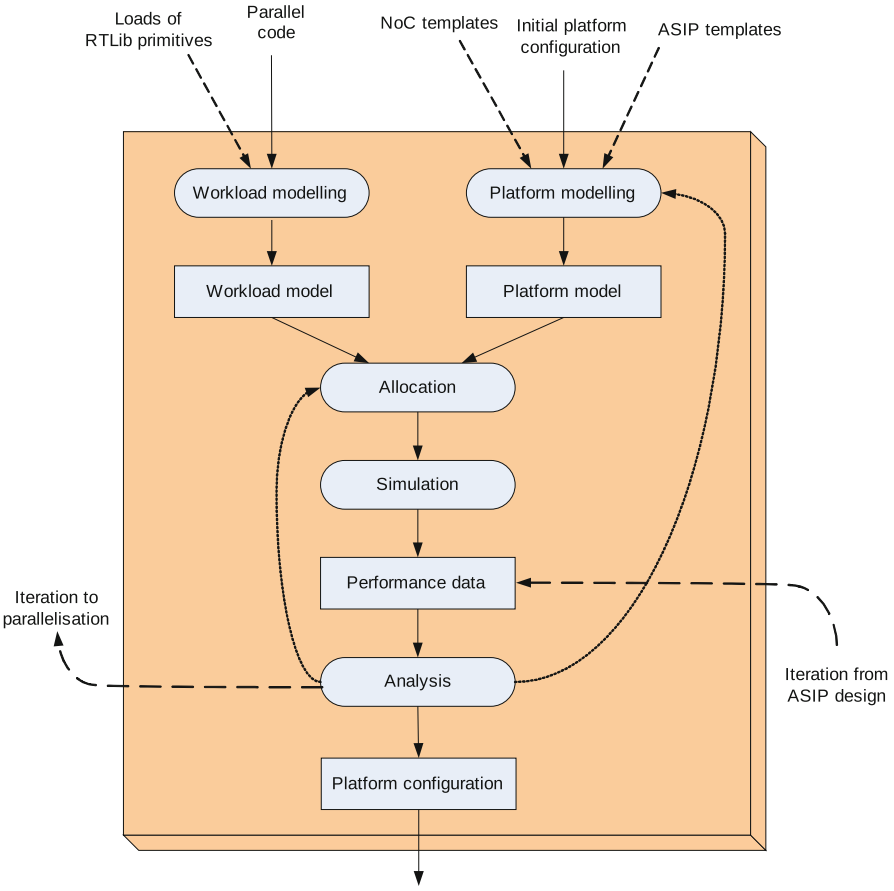

Fig. 5.2 MOSART design flow at the system-level exploration level

the execution platforms are modelled in terms of services provided. The layered hierarchical workload models represent the computation and communication loads the applications cause on the platform when executed. The layered hierarchical platform models represent the computation and communication capacities the platform offers to the applications. The workload models are mapped onto the platform models ([16]) and the resulting system model is simulated at transaction-level to obtain performance data. The tool support is based on open source SystemC2 simulation library of Open SystemC Initiative (OSCI). The approach enables performance evaluation early, exhibits light modelling effort, allows fast exploration iteration, reuses application and platform models, and provides performance results that are accurate enough for system-level exploration ([14, 15]). Recently, an automatic tool based on a modified compiler has been developed to enable easy workload generation from source code ([20]).

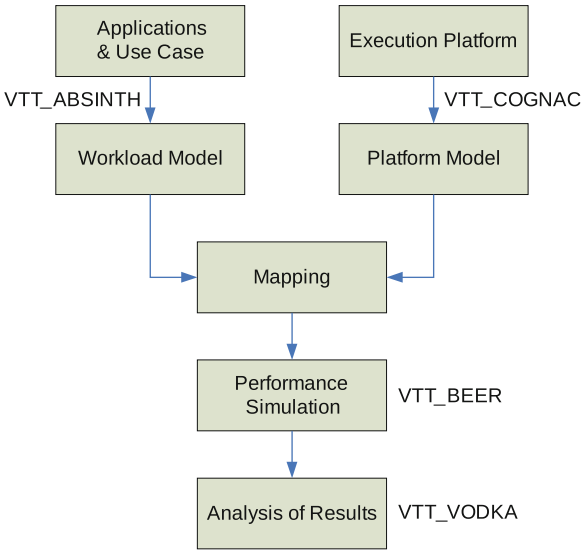

Fig. 5.3 Y-chart model of ABSOLUT

The starting points for the performance modelling are the end-user requirements of the system. These are modelled as a service-oriented application model, which has a layered hierarchy. The top layer consists of system level services visible to the user that are composed of sub-services and divided further to primitive services.

The purpose of workload modelling is to illustrate the load an application causes to an execution platform when executed. Workload models are non-functional in the sense that they do not perform the calculations or operations of the original application. Workload modelling enables performance evaluation already in the early phases of the design process, because the models do not require that the applications are finalised. Workload modelling also enhances simulation speed as the functionality is not simulated and models can be easily modified to quickly evaluate various use cases.

Platform modelling comprises the description of both hardware and platform software (middleware) components and interconnections that are needed for performance simulation. Like workload modelling, platform modelling considers hierarchical and repetitive structures to exploit topology and parallelism. The resulting models provide interfaces, through which the workload models use the resources and services provided by the platform ([13]).

After mapping the workloads to the platform, the models can be combined for transaction-level performance simulation in SystemC. Based on the simulation results, we can analyse e.g. processor utilisation, bus or memory traffic and execution time.

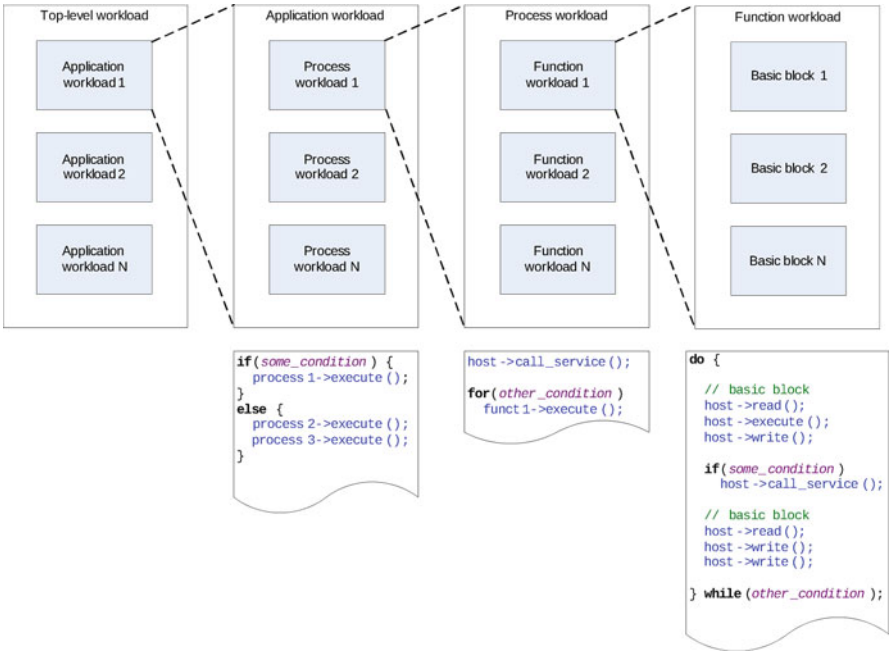

Fig. 5.4 Workload models have a hierarchical structure

5.2.1 Application Models

Applications are modelled as workloads, which characterise the control flow and the loads of the data processing and communication of applications on the execution platform. Therefore the models can be created and simulated before the applications are finalised, enabling early performance evaluation. As opposed to most of the performance simulation approaches, the workload models do not contain timing information. It is left to the platform model to find out how long it takes to process the workloads. This arrangement results in enhanced modelling and simulation speed. It is also easy to modify the models, which facilitates easier evaluation of various use cases with minor differences. For example, it is possible to parameterise the models so that the execution order of applications varies from one use case to another.

The workload models have a hierarchical structure, where top-level workload model W divides into application workloads A_i, $1 \leq i \leq N$ for different processing units of the physical architecture model (Fig. 5.4):

$$W = C_a, A_1, A_2, \ldots, A_n \tag{5.1}$$

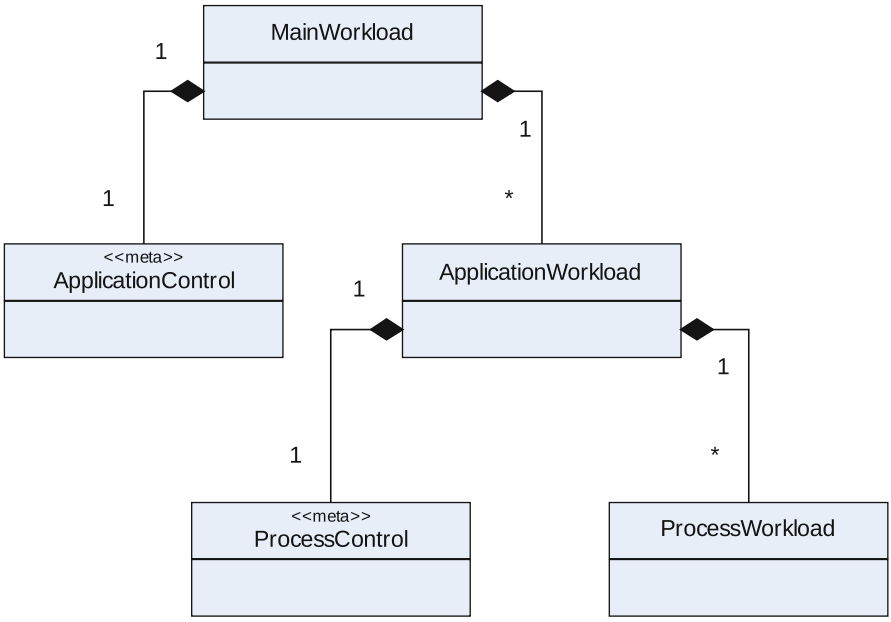

Fig. 5.5 Top-level and application workloads consist of lower level items and control

where C_a denotes the common control between the workloads, which takes care of the concurrent execution of loads mapped to different processors. n is the number of application workloads under the top-level workload.

The structure of the main workload model and the application workloads is depicted in the UML diagram of Fig. 5.5. The application and process control are shown as classes in the diagram; however, they may be implemented using e.g. standard C++ control structures in SystemC based workload models.

Process and function workload models can also be statistical. In this case the model will describe the total number of different types of load primitives and the control is a statistical distribution for the primitives (Fig. 5.6). This is beneficial in case the chosen load extraction method is not accurate enough so that functions and/or basic blocks could be modelled in detail. Less important, e.g. background, workloads can also be modelled this way for reducing the modelling effort. Workload models using deterministic process models but statistical function models are more accurate than those using statistical process models. Models, which are deterministic down to basic block level are of course the most accurate.

5.2.1.1 Application Layer

Each application workload A_i is constructed of one or more processes P_i:

$$A_i = C_p, P_1, P_2, \ldots, P_n \tag{5.2}$$

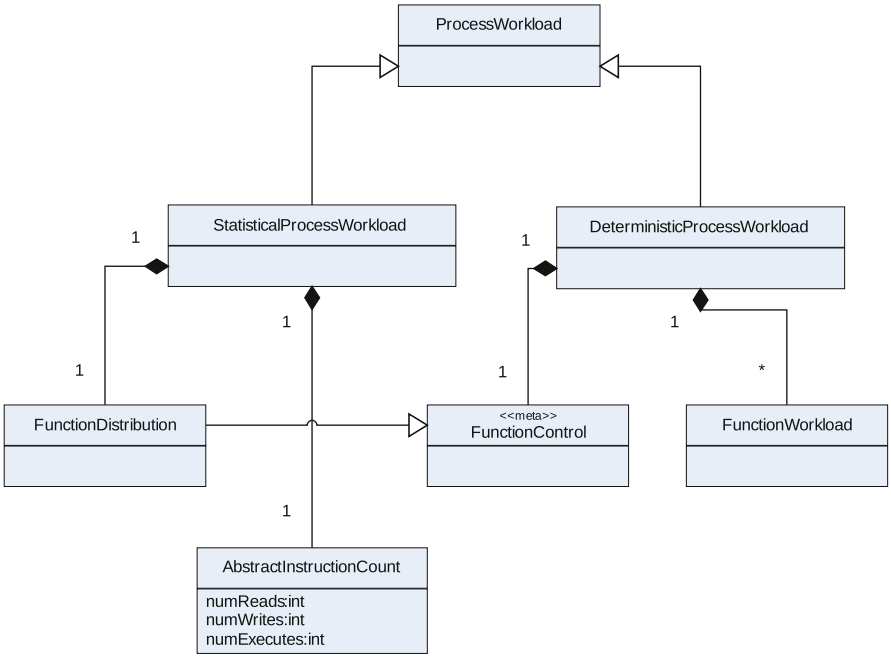

Fig. 5.6 The process workloads can be either statistical or deterministic

Table 5.1 Public Application interface

Methods and members	Description
Application(sc_module_name _name, Subsystem* host)	Constructor
virtual void exec_app() = 0	Pure virtual method, which triggers the simulation of the application model

where C_p corresponds to the control between the processes. Sequential applications consist of a single process, whereas parallel (multi-threaded) applications contain several processes.

In the SystemC implementation the base classes of all layers, including Application (Table 5.1), are derived from a generic Workload class. The constructor parameters of Application are name, which is a unique SystemC name for the created object, and host, which is the subsystem to which the application is mapped. The application behaviour, i.e. starting and stopping of processes, should be implemented in derived models inside the exec_app() function.

Table 5.2 Public Process interface

Methods and members	Description
Process(sc_module_name _name, Proc_ctl_IF * host, **bool** loop = PROCESS_NORMAL)	Constructor

5.2.1.2 Process Layer

The processes are comprised of function workloads F_i:

$$P_i = C_f, F_1, F_2, \ldots, F_n \tag{5.3}$$

where C_f is control and describes the relations of the functions. The operating system models of the platform handle workload scheduling at the process level.

The public interface of the Process base class consists of only the constructor (Table 5.2). The constructor parameters are:

- sc_module_name _name, which is unique SystemC name for the Process object (as is usual for SystemC models)
- Proc_ctl_IF * host, which is the operating system model handling the scheduling of the process
- **bool** loop, which defines whether the process is executed in an infinite loop (PROCESS_INFINITE) or not (PROCESS_NORMAL, the default).

Looped execution can be useful for modelling constant background load, which is e.g. started at the beginning in the simulation and left running until the end.

Internally Process contains wrappers around the SystemC wait() functions so that the process workloads are automatically stopped when waiting for SystemC events and started again after reception. There is also a SystemC thread, whose contents must be implemented by each process workload model derived from the base class. The thread should contain the control, which defines how the simulation of the process progresses from one function to another.

5.2.1.3 Function Layer

Function workload models are basically control flow graphs ([18])

$$F_i = (V, G) \tag{5.4}$$

where nodes $v_i \in V$ are basic blocks and arcs $g_i \in G$ are branches. Real, high-level applications implement control inside functions as for, do, or while loops and if statements. In low-level code these are implemented with comparison instructions (less than, greater than, equal to, ...) and conditional branches. It is decided dynamically during the execution of the application if the branches are taken or not and the decision depends on the data. However, with the ABSOLUT approach

Table 5.3 Public Function interface

Methods and members	Description
Function(sc_module_name _name, Primitive_IF * host, Address read_addr, Address write_addr)	Constructor
virtual void exec_fn() = 0	Pure virtual method, which triggers the simulation of the function model
int branch_test()	Used for statistically selecting the next basic block for simulation

the actual data processing of the application is abstracted away and, as a result, data dependent selection of branches can not be done. Instead, the branches are simulated statistically and each branch has a probability *P*, which defines the likelihood of the branch being taken.

The basic blocks are ordered sets of load primitives used for load characterization and are described in the next section.

Function() is the constructor of the SystemC base class for function models (Table 5.3) and it takes a sc_module_name object as a parameter as is usual for SystemC models. The second parameter, host, is a pointer to the part of the platform (operating system) that will run the function. Finally, read_addr and write_addr define the addresses for read and write primitives within the function.

branch_test() is a helper function, which invokes the random number generator for simulating the branches. The return value is between 0 and 2^{32} based on which the proper branch should be selected. exec_fn() is a virtual function, which must be implemented by the derived models. It should contain the implementation of the control flow using the branch_test(). For example,

```
bb_49:
  node_49();

  b = branch_test();
  // target bb 50 (P = 70.000000%)
  if(b < 7000)
    goto bb_50;
  // target bb 51 (P = 30.000000%)
  goto bb_51;
```

where node_49() would contain the load primitives of the 49th basic block. exec_fn() may be called by process workloads and other function workloads to trigger the execution of the function.

5.2.1.4 Load Primitive Layer

Load primitives form the lowest layer of application models. They are generic, abstracted versions of the instructions of processing units. Different processing units may have a different set of instructions, whereas the load primitives are the same for

all workloads. As a consequence, the workload models can be mapped to different processing units without changes.

The instructions of processors can be categorised as data processing instructions (arithmetic, boolean, shifts, ...), control flow instructions (jumps, branches, calls), and load/store instructions. The load and store instructions cause contention to the interconnections and memories of the system, which are typically shared resources. On the other hand, the effects of the rest of the instructions stay within the processor at least if the instruction fetches are not considered, i.e. the impact on the other parts of the system is limited. Therefore, the load primitives consist of the following:

- Read, which simulates a memory load instruction;
- Write, which simulates a memory write instruction;
- Execute, which simulates all the other instructions.

The load primitive layer consists of a number of consecutive primitives. The primitives may be coalesced: e.g. instead of five consecutive execute primitives there is a single `execute(5)`. The primitives are transferred from the workload side to the platform side through the blocking primitive interface (Sect. 5.2.5.1), which contains a separate interface function for each primitive. Typically, the load primitive layer is implemented in the same source file as the function layer. For example, the following code extract shows the implementation (primitives) of the 49th basic block of the previous example:

```
inline void main_WL::node_49()
{

  m_host->read(m_read_addr, 4, 8);
  m_host->execute(2);

}
```

where m_host is the operating system, which is hosting the function.

5.2.2 *Platform Models*

The platform model is an abstracted hierarchical representation of the actual platform architecture. It contains cycle-approximate timing information along with structural and behavioural aspects. The platform model is composed of three layers: component layer, subsystem layer, and platform architecture layer (Fig. 5.7).

Each layer has its own services, which are abstraction views of the architecture models. They describe the platform behaviours and related attributes, e.g. performance, but hide other details. Services in the subsystem and platform architecture layers can be invoked by workload models. High-level services are built on low-level services, and they can also use the services at the same level. Each service might have many different implementations. This makes the design space exploration

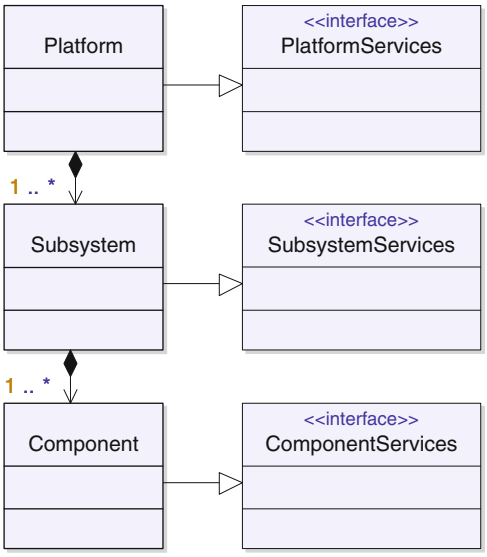

Fig. 5.7 The execution platform model consists of platform, subsystem and component layers

process easier, because replacing components or platforms by others could be easily done as long as they implement the same services.

5.2.2.1 Platform Layer

The platform architecture layer is built on top of the subsystem layer by incorporating platform software and serves as the portals that link the workload models and the platforms in the mapping process. Platform-layer services consist of service declaration and instantiation information. The service declaration describes the functionalities that the platform can provide. Because a platform can provide the same service with quite different manners, the instantiation information describes how a service is instantiated in a platform.

The platform-layer services are divided into several categories with each category matching one application domain, e.g. video processing, audio processing and encryption/decryption. The OS system call services are in an individual domain, and as mentioned earlier they can also be invoked by other services at the same level. A number of platform-layer services are defined for each domain and more could be added if necessary.

Application workloads typically call platform or subsystem level services, process workloads call subsystem services, and function workloads call component-level services. Ideally, all services required by the application are provided by the execution platform and there is a 1:1 mapping between the requirements and provisions. However, often this is not the case and the workloads need to use several lower level services in combination to produce the desired effect.

5.2.2.2 Subsystem Layer

The subsystem layer is built on top of the component layer and describes the components of the system and how they are connected. The services used at this layer could include e.g. video preprocessing, decoding and postprocessing for a video acceleration subsystem.

The model can be presented as a composition of structure diagrams that instantiates the elements taken from the library. The load of the application is executed on processing elements. The communication network connects the processing elements with each other. The processing elements are connected to the communication network via interfaces.

5.2.2.3 Component Layer

This layer consists of processing (e.g. processors, DSPs, dedicated hardware and reconfigurable logic), storage, and interconnection (e.g. bus and network structure) elements. An element must implement one or more types of component-layer services. For example, a network interface component should implement both master and slave services. In addition, some elements need to implement services that are not explicitly defined in component-layer services, e.g. a bus shall support arbitration and a network shall support routing.

The component-layer read, write and execute services are the primitive services, based on which higher level services are built. The processing elements in the component layer realise the low-level workload-platform interface, through which the load primitives are transferred from the workload side. The processing element models will then generate accesses to the interconnections and slaves as appropriate.

All the component models contain cycle-approximate or cycle-accurate timing information. Specifically, the data path of processing units is not modelled in detail; instead the processor models have a cycles per instruction (CPI) value, which is used in estimating the execution time of the workloads. For example, the execution time for data processing instructions is the number of instructions to execute times CPI (). Furthermore, caches and SDRAM page misses, for example, are modelled statistically since the workload models typically do not include accurate address information.

The SystemC implementation of the generic ABSOLUT base class for all component models, Component, provides only the constructor as its public interface. It has a single parameter for the SystemC module name. Internally there is a method for simulating cycle-based latencies in the operation of the models derived from Component. Finally, a number of macros define parameters common to all components, including base address and clock period. The base address is only required for identification of the components: an accurate memory map is not needed by ABSOLUT.

ABSOLUT base classes exist also for master and slave models with OCP TL2 protocol interfaces. These are derived from Component and can be used to quickly

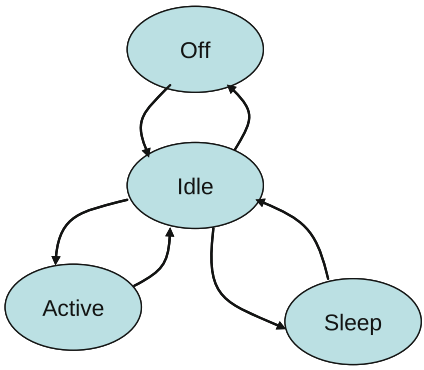

Fig. 5.8 Power state machine

implement new components using the OCP interface. The Master and Slave classes include an OCP master and slave port respectively. The Master provides helper methods for sending requests to the OCP port and obtaining responses from the port. Correspondingly, the Slave class has methods for obtaining requests from the slave port, processing them and sending responses back to the initiating master model.

5.2.2.4 Modelling of Power Consumption

The concepts of power-managed system modelled as a set of interacting power manageable components (PMC) controlled by a power manager were introduced in [1]. A PMC can be managed internally, which means that a component model has an internal power manager, or externally, where information for power control comes basically from the OS and/or application. The power consumption states are presented in the component model as a power state machine (PSM) depicted in [2].

The ABSOLUT power modelling extension divides in practice into two: One is included in the resource simulation model that facilitates the registration of time the resource spends in each of its power states during simulation through instrumentation called power probes. The other is associated to the power analysis for calculating the power consumed at each state off-line the actual simulation.

The power state machine used in modelling is depicted in Fig. 5.8. In the Off state the resource does not contribute to power consumption. When operational, the Active state power is consumed when a resource of the platform is excited by transactions of the workload model, otherwise the resource consumes Idle state power or Sleep state power. The transitions between Idle and Active states are internal. The transitions between Idle and Sleep states can be controlled internally or externally. The transitions between Idle and Off states are controlled externally.

The three main approaches for controlling power consumption include:

- Change of component power mode: Active, Idle, Sleep and Off.
- Dynamic voltage and frequency scaling (DVFS).
- Application-driven dynamic power management (DPM).

The external control in ABSOLUT can be implemented through exploiting the high-level service interface when the respective service is available in the resource model.

The power consumption values for processor cores can be obtained from e.g. data sheets or reference designs ([6]) for power state model approaches. The energy consumption of a CPU for a time period T is calculated as:

$$E_c = P_a T_a + P_s T_s + P_i T_i \tag{5.5}$$

where P_a, P_s and P_i are power consumption in active, sleep and idle states and, correspondingly, T_a, T_s and T_i are the time spent in the same states. The time period $T = T_a + T_s + T_i$ can be selected according to the interests, e.g. start and end of a service or some other functionality.

The memory power state machine is basically the same as in Fig. 5.8, where the Active state refers to read or write to/from the memory and the Sleep state refers to taking the memory to standby. ABSOLUT simulation probes register the time (as numbers of cycles) spent in each of the power states that is then used to calculate power consumed. Energy consumption of memory for a time period T is calculated as:

$$E_m = P_r T_r + P_w T_w + P_i T_i \tag{5.6}$$

where P_r and P_w are power consumption in read and write states and T_r and T_w are the time spent in the corresponding states.

Power consumption values for memories can be found in data sheets, e.g. [25]. There are also power calculators available for different DRAMs ([24]). Applicable to different memory types, e.g. SRAM and eDRAM, CACTI 5.0 is a tool for modelling the dynamic power, access time, area, and leakage power ([30]). The special features of memories should be taken into account when defining the power values for different power states.

For a bus, the active and idle states are registered. The power model is based on average power consumption in these states. Energy consumption of bus for a time period T is calculated as:

$$E_b = P_a T_a + P_i T_i \tag{5.7}$$

Power consumption values for buses are not so straightforward to obtain due to configuration complexity. Several papers describe using of the Hamming distance to account for changes of states of bus wires, e.g. [3]. Gate-level power simulation is used in [5] to characterise CoreConnect based system for transaction-level power estimation. An analysis of AMBA bus architecture is presented in [21] giving breakdown information about how much different bus elements contribute to power consumption.

Calculating Power and Energy Consumption

If we assume that the power consumption of the components in each of the different states is given as parameters to the models, we can estimate the total power consumption P_i at time i:

$$P_i = \sum_n P_{n,i} \tag{5.8}$$

where $P_{n,i}$ is the instantaneous power consumption of component n. The total energy consumption of the entire system during the execution of the use case is calculated using the following formula

$$E = \sum_n E_n = \sum_n \sum_i P_{n,i} t_i \tag{5.9}$$

It is not reasonable for the system simulation model to calculate P for each time delta due to the huge amount of data it would generate and because it might have an adverse effect on simulation speed. Instead, it should be sampled like the component utilisation is at the moment.

Furthermore, the components may support dynamic voltage and frequency scaling (DVFS). In general, frequency has a linear effect on power consumption, whereas doubling the voltage quadruples power consumption. Thus,

$$P_{f,V} = P_n \frac{f V^2}{f_n V_n^2} \tag{5.10}$$

where $P_{v,f}$ is the new power consumption of the component using frequency f and voltage V, with normal power consumption of P_n at frequency f_n and voltage V_n.

It should be noted that the energy required for doing a specific amount of computation does not scale directly with the frequency. With a lower frequency the computation simply lasts longer:

$$E = Pt = fCV^2 \frac{N}{f} = CV^2 N \tag{5.11}$$

where N is the number of cycles required by the computation. Frequency scaling can still be a useful mechanism, e.g. in cases where the application workload can be somehow scaled, or in case the voltage can be scaled due to scaling of the frequency.

5.2.3 Mapping Workloads to the Platform

In order to facilitate simulation of the system model consisting of both the application and platform models, the applications need to be mapped to the platform. Mapping involves choosing the part of the platform, which will host (execute) each

particular workload model, and which instruction and data memory or memories they will utilise. Mapping the execution is done during the initialisation of the workload model, i.e. each workload model receives a pointer to its host as a constructor parameter. The mapping is done in several layers in such a way that:

- Application workload models are mapped to subsystem models,
- Process workload models are mapped to operating system models (inside subsystems), and
- Function workload models are mapped to processing unit models.

For example, the following code sequence initialises a subsystem and a video player application. The second constructor parameter of the video player application model maps the application to the subsystem.

```
ss = new Simple_subsystem("ss");
sva = new Simple_video_app("sva", ss);
```

Then, the application model maps its processes to the operating system model first in the constructor of the process model and then by registering the processes to the OS model through the process control interface (Table 5.6). Inside the Simple_video_app this would be implemented as follows:

```
svp = new Simple_video_process("svp", m_host->os());
m_host->os()->register_process(svp, REGISTER_PROCESS_RUNNING);
```

The operating system model is already mapped to a fixed set of one or more processors in its initialisation. Thus, the process model will just distribute its pointer to the OS model when mapping the functions to the processing units. Specifically inside the Simple_video_process:

```
video_postprocess_fn = new Video_postprocess_fn("
    video_postprocess_fn", host, m_video_data_addr,
    m_framebuffer_addr);
```

The second parameter, host, of the function constructor call above defines the processing unit that will execute the function during the simulation.

The memory mapping is done via memory addresses, which are used as parameters to high- and/or low-level interface calls. These addresses could for example be hard-coded in the models. However, the recommended way is to set these up as model parameters in the process workloads, and the process workloads then distribute the addresses to function workloads during their initialisation. For example, in the Simple_video_process there could be three memory address parameters:

```
PARAM_DECL(Address, video_app_addr)
PARAM_DECL(Address, video_data_addr);
PARAM_DECL(Address, framebuffer_addr);
```

which define the location for the video player application (code), video data, and video framebuffer (display) respectively. These parameters are given as parameters to the function workload constructors:

```
video_main_fn = new Video_main_fn("video_main_fn", host,
    m_video_app_addr, m_video_app_addr);
video_postprocess_fn = new Video_postprocess_fn("
    video_postprocess_fn", host, m_video_data_addr,
    m_framebuffer_addr);
video_exit_fn = new Video_exit_fn("video_exit_fn", host,
    m_video_app_addr, m_video_app_addr);
```

The last two parameters of the function workload constructor define the addresses for read and write primitives respectively.

5.2.4 Service Models

The services inside the platform can model either hardware or software services. In ABSOLUT, software services are modeled as workload models, but unlike application models, they are integrated in the platform model and easily reusable by the applications. If the service is provided by a process or a set of processes running in the system, the service model consists of application or process layer workloads. If the service is implemented as a library, the model will be at the function layer. Service models can utilize other services, but eventually they consist of the same read/write/execute load primitives as the application models.

There are two alternatives how to implement a HW service: It can be implemented simply as a delay in the associated component, if the processing of the service does not affect the other parts of the system at all. In this case the service must not perform I/O operations or request other services. The second alternative is to implement the service as read, write and possibly execute primitives like the SW services, but in this case they are executed inside the HW component and not inside a process workload running on one of the processor models.

5.2.4.1 Service Registration

Services can be registered in either the subsystem or platform layer of the execution platform model. Subsystem-local services are registered only to the local OS model, whereas system-wide services are further registered by the local OS to the master OS, which implements global service registry.

5.2.4.2 Service Use

Process or function workload models can request high-level services from the platform model through the generic service interface (Table 5.4). The services are requested from the local operating system model, which will relay the service

Table 5.4 The services of the platform model are exploited via the high-level interface

Interface function	Description
Serv_id use_service (std :: string name, Serv_attr attr)	Request service name using attr as parameters
void wait_service (Serv_id id)	Wait untile the completion of service id

Table 5.5 The low-level interface consists of functions intended for transferring load primitives between workload and platform models

Interface function	Description
void read (Address a, **unsigned** W, **unsigned** B)	Read W words of B bits from address A
void write (Address a, **unsigned** W, **unsigned** B)	Write W words of B bits to address A
void execute (**unsigned** N)	Simulate N data processing instructions

request to the service provider if the implementation of the service can be found in the same subsystem. Otherwise, the OS model relays it to the master OS, which will call the correct subsystem.

The workload model does not need to know where implementation of the service resides. It will just request the service, pass it some service-dependent attributes and optionally wait for the completion of the service.

5.2.5 Interfaces

5.2.5.1 Application to Platform Interfaces

Generic Service Interface

The high-level interface enables workload models to request services from the platform model (Table 5.4). These functions can be called from workload models between the function and application layers. The use_service () call is used to request the given service and it is non-blocking so that the workload model can continue while the service is being processed. use_service () returns a unique service identifier, which can be given as a parameter to the blocking wait_service () call to wait until the requested service has been completed, if necessary.

Load Primitive Interface

The low-level interface is intended for transferring load primitives and is depicted in Table 5.5. The functions of the low-level interface are blocking – in other words a load primitive level workload model is not able to issue further primitives before the previous primitives have been executed. Load extraction methods are typically

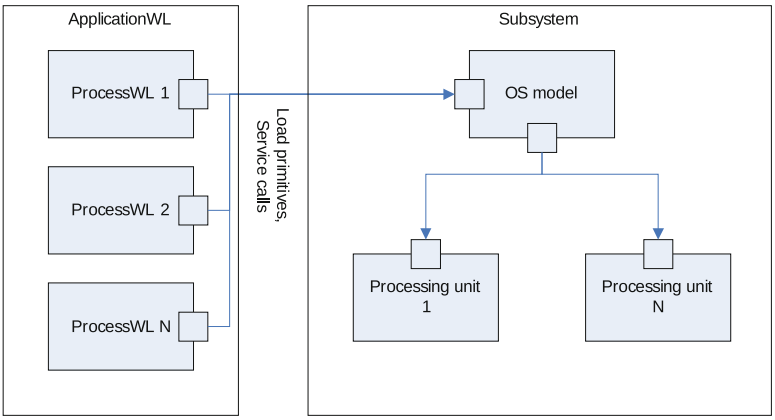

Fig. 5.9 During simulation process workloads send load primitives and service calls to the platform model

not able to extract accurate address information for the load primitive layer of the workloads. Hence, the address parameter of the read and write functions often just corresponds to the base address of the memory block, which is used by the workload.

5.2.5.2 Operating System and its Interfaces

In simple cases the execution of workloads can be scheduled manually by hard-coding it to the models. However, it is not recommended. Typically the platform model includes one or more operating system (OS) models, which control access to the processing unit models of the platform by scheduling the execution of process workload models (Fig. 5.9). The OS model provides both low- and high-level interfaces to the workloads and relays interface function calls to the processor or other models which realise those interfaces. The OS model will allow only those process workloads which have been scheduled for execution to call the interface functions. Rescheduling of process workloads is performed periodically according to the scheduling policy implemented in the model.

Process Control Interface

The OS model extends the primitive interface with a number of functions for registering, starting, stopping and killing process workloads (Table 5.6). For mapping a process workload to an OS model, the process must be registered with

Table 5.6 The process control interface extends the primitive interface with a number of functions

Interface function	Description
register_process (Process* p, **bool** start)	Inform the operating system model of the process workload model p. If start == **true**, the process is started immediately. Otherwise it will wait in the stopped state until explicitly started with start_process ()
start_process (pid_t pid)	Start the process with the given process id
stop_process (pid_t pid)	Stop the process with the given process id
kill_process (pid_t pid)	Kill (unregister) the process with the given process id

Table 5.7 The service registration interface contains only one function

Interface function	Description
register_service (provider, name, type)	Register a service name of type provided by provider

register_process (). For the OS model to actually schedule the process for execution the process must be started. Processes can also be stopped to temporarily disable them (e.g. while they are waiting for an external event) and killed once they are not needed anymore. Killing a process just removes its mapping to the operating system model and does not delete the process workload object.

Service Registration Interface

High-level services must be registered to OS models so that application models can utilise them through the generic service interface (Table 5.4). Service registration interface (Table 5.7) consists of a single function register_service (), which takes a pointer to the service provider, service name, and service type as parameters. The service provider can be any model, which realises the generic service interface, e.g. a platform component or a process workload model.

The service provided must provide the generic service interface of Table 5.4.

5.2.6 Model Parameters

ABSOLUT provides a number of macros for adding parameters to both application and platform models. Parameters are useful for setting up model latencies, for example. The values of the parameters are read from a configuration file during the system model initialisation and thus changing the values does not require recompilation.

PARAM_DECL(type, name) macro is used in the header files of models to declare parameters, which are constant. Normal C++ rules apply to the type and name of the parameter. PARAM_DECL_VAR(type, name) macro declares parameter, whose value can be changed during the simulation. If the declaration is in the public part of the class declaration, all workload and execution platform models are able to modify its value through method name(value).

PARAM_SET(type, name) macro is used in the implementation of model constructors. It will automatically obtain the value of the parameter from a configuration file when the model is constructed. By default, this configuration file is config in the directory where the simulation is started. Type and name must be exactly the same as in the corresponding PARAM_DECL() or PARAM_DECL_VAR(). PARAM_SET3 (type, name, defval) behaves the same as PARAM_SET() with one exception. The third parameter, defval, defines a default value which will be used if the configuration file does not specify a different value.

PARAM_VALUE(name) macro contains the value of a parameter declared with PARAM_DECL(type, name) or PARAM_DECL_VAR(type, name) macro.

5.3 ABSINTH Workload Model Generator

ABSINTH (ABStract INstruction exTraction Helper) presented [20] is a tool for generating workload models from application source code. ABSINTH has been implemented by extending GNU Compiler Collection (GCC) version 4.3.1 with two additional passes. It can be triggered with a single switch during the compilation of any source code supported by GCC.

The first ABSINTH pass is responsible for constructing the function layer of the workload model, i.e. the control flow between basic blocks in each source code function. The second pass will traverse RTL (GCC's low-level intermediate language) to extract load primitives read, write, and execute for each basic block.

There are three phases in the model generation process with ABSINTH:

1. First, the source code must be compiled with profiling,
2. Then, the compiled binary must be executed with a data set corresponding to the use case, and
3. Finally, the source code must be compiled again with both profile-guided optimization and ABSINTH enabled

5.3.1 Interface Between IMEC MPA and ABSOLUT

The IMEC's MPSoC Parallelization Assist (MPA) is a tool for efficiently mapping applications onto multicore platforms ([26]). It takes the sequential C source code

and a parspec file describing its computational partitioning as input. The tool performs the partitioning, inserting communication and synchronisation as required to respect the dependencies in the sequential application.

The interface between MPA and ABSOLUT is based on modelling primitives of the MPA-API (RTlib) as services in ABSOLUT. This facilitates generation of workload models of the MPA parallelised code using ABSINTH workload generation, and mapping them onto a multi-core execution platform model for performance simulation.

5.4 ABSOLUT Performance Simulation

The executable simulation model of the combined workload and execution platform models is based on the OSCI SystemC library, extended with configurable instrumentation. During the simulation of the system model the workloads send load primitives and service calls to the platform model. The platform model processes the primitives and service calls, advancing the simulation time while doing so. The simulation run will continue until the top-level workload model stops it when the use case has been completed.

The platform model is instrumented with counters, timers and probes, which record the status of the components during the simulation. These performance probes are manually inserted in the component models where appropriate and are flexible so that they can be used to gather information about platform performance as needed. They can also be used inside application models as well. After simulation the performance probes output the collected performance data to the standard output. A C++-based tool, VODKA, is used for viewing e.g. processor utilization, bus and memory traffic and execution time, graphically.

5.4.1 Performance Probes

Performance probes can be used to extract performance data of e.g. component utilisation, request / response traffic, or latencies during the simulation run. The probes are inserted manually in the models in appropriate places. It is up to the user to decide, what information to extract and where to put the probes to extract it.

5.4.1.1 Status Probe

Status probe implements a two-state (e.g. on/off, busy/idle) probe, which measures the time spent in both states and calculates the proportion of time spent in each state. Status probes are intended to collect information about e.g. the utilization of components and scheduling of processes performed by the operating system models.

They can record the new status of a component and the time that was spent in the previous state each time it changes. They will also periodically record the time spent in each state in an interval. For example, the existing processor, accelerator and memory models use this to record, how the utilisation of those components changes as a function of time. The VODKA tool is able to show the resulting curves graphically.

Sample output from an ARM model:

```
p.ss.arm: Idle:    57.9338% [2.87158e+09 ns]
p.ss.arm: Busy:    5.82825% [2.88886e+08 ns]
p.ss.arm: Waiting: 36.238% [1.79619e+09 ns]
```

where the ARM has spent 58% of simulation time idling, 6% processing data, and 36% waiting for memory accesses to complete. The numbers in square brackets give the corresponding time in nanoseconds.

5.4.1.2 Timer

Timers are used to measure the elapsed time in system events during the simulation, including task switch times of the OS models and processing times of services.

A timer will display the average, minimum, and maximum time measured by the probe during the simulation. For example,

```
p.ss.dif.disp_upd: 8.6e+6 (8.2e+6-9.2e+6) [248]
```

where the display interface has updated the screen 248 times and it has taken $8.6 * 10^6$ ns on average. The fastest update took $8.2 * 10^6$ ns and the slowest $9.2 * 10^6$ ns.

5.4.1.3 Counter

Counters are used to calculate the number of load primitives, service calls, requests and responses performed by the components.

In the following sample

```
p.ss.os.counter_task_switches: 744
```

the operating system model has performed 744 task switches during the simulation run.

5.5 Case Example

The ABSOLUT approach has been applied to a JPEG encoding case example. The application in this case is a JPEG encoder written in the C language. The original source code is purely sequential and utilises only a single thread of execution. For

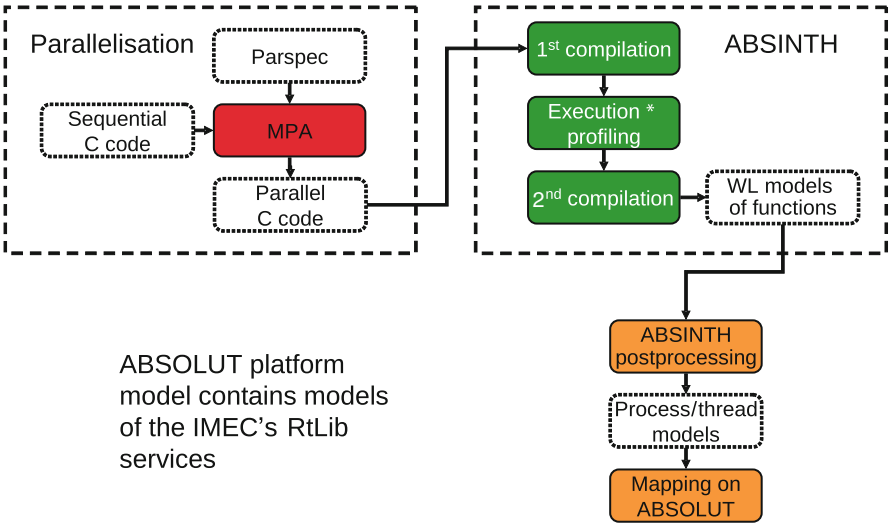

Fig. 5.10 Applying MPA and ABSINTH in JPEG encoder case study

the case example several parallel versions of the JPEG encoder were created with the IMEC MPA. The platform model contained a model of the MPA runtime library for thread management. The process is depicted in Fig. 5.10.

ABSINTH was used to generate four sets of workload models from the encoder source code. The first one was from the unmodified sequential application and the other three from parallelised versions of the application: Par-1 had two threads with the second thread executing Getblock and DCT algorithms. Par-2 consisted of three threads with the second and third one interleaving the execution of Getblock, DCT, and Quantization. Par-3 had also three threads with the second and third thread executing just Getblock and DCT in an interleaved manner.

The execution platform model is depicted in Fig. 5.11. It consists of 4 ARM nodes connected by routers, which form a ring-like network. Each node has an ARM9 CPU, local SRAM memory, a shared bus, and an interface to the other nodes. The essential parameters of the platform components are shown in Table 5.8.

The models were mapped to the ARM nodes of the platform according to Table 5.9. The mapping of the threads can be modified with a single line of code in ABSOLUT. Furthermore, altering the partitioning of the algorithms can be done quickly by modifying the parallelisation specification and then running both MPA and ABSINTH to generate another set of models.

The execution time of the sequential encoder was about 70 ms (Table 5.10). The Par-1 and Par-3 versions improved it to about 55 ms, i.e. the partitioning in Par-3 (Table III) did not bring real benefits compared to Par-1. However, Par-2 provided a speedup of 1.8 with an execution time of 39 ms. All speedups given

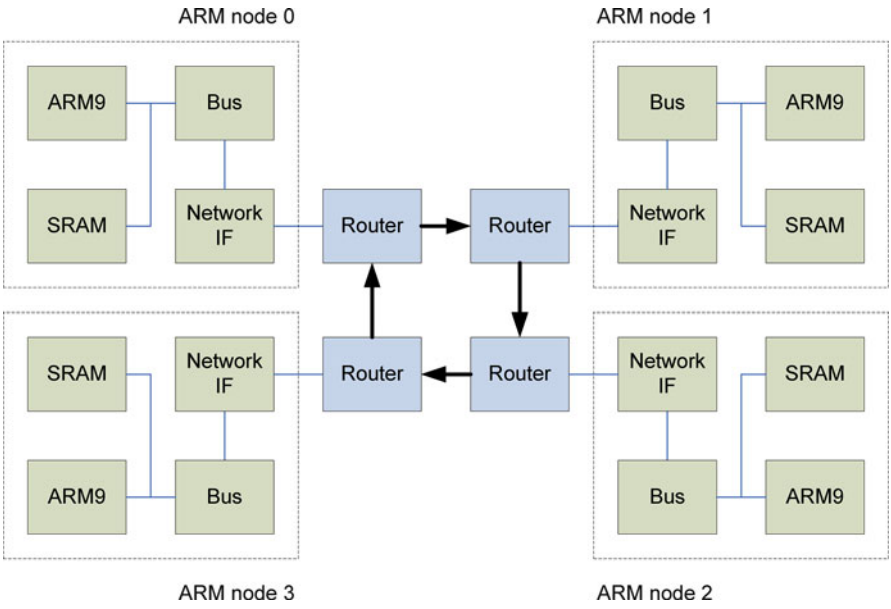

Fig. 5.11 Platform consisting of four ARM nodes

Table 5.8 Platform model configuration

Component	Parameters
ARM	200 MHz clock, writeback cache
SRAM memory	166 MHz clock, 32 Mb
Bus	100 MHz clock
Network IF	100 MHz clock
Router	100 MHz clock

Table 5.9 Mapping of JPEG encoder parallelisations to subsystems

	Seq	Par-1	Par-2	Par-3
Localthread Info	Node0	Node0	Node0	Node0
Getblock DCT		Node1	Node1, Node2	Node1, Node2
Quantization		Node0		Node0
Huffman			Node0	

by the ABSOLUT approach are lower than the theoretical maximum given by MPA high-level simulation, which assumes linear scaling and does not take the communication overhead into account.

Both Par-1 and Par-3 have 100% utilisation on the cpu of the ARM node 0. Par-1 has 44% cpu utilisation in the second ARM node, whereas Par-3 has 21% utilisation across nodes 1 and 2. Par-2 has 88% utilisation in the first node: it is idling at some

Table 5.10 JPEG encoder simulation results

		Seq	Par-1	Par-2	Par-3
Execution time [ms]		69.5	55.3	38.7	53.9
Speedup		1.00	1.26	1.80	1.29
MPA HLS speedup		1.00	1.66	2.50	1.66
Node0 utilisation	ARM	100%	100%	88%	100%
	SDRAM	59%	55%	43%	53%
	Bus	34%	36%	29%	35%
Node1 utilisation	ARM	0%	44%	53%	21%
	SDRAM	0%	23%	31%	11%
	Bus	0%	12%	18%	6%
Node2 utilisation	ARM	0%	0%	53%	21%
	SDRAM	0%	0%	31%	11%
	Bus	0%	0%	18%	6%

Table 5.11 Power consumption configuration

	Power consumption
ARM idle	6 mW
ARM active	0.133 mW / MHz + 5 mW $\rightarrow$ 31.6 mW
Cache idle	4 mW
Cache active	0.060 mW / MHz + 4 mW $\rightarrow$ 16 mW
Memory idle	$1 * 10^{-6}$ mW / bit-cell $\rightarrow$ 32 mW
Memory active	$1 * 10^{-6}$ mW / MB + $6 * 10^{-7}$ mW / MB $\rightarrow$ 51.2 mW
Bus	0.03 mW / MHz $\rightarrow$ 3 mW
Network IF	0.06 mW / MHz $\rightarrow$ 6 mW
Router	0.037 mW / MHz $\rightarrow$ 3.7 mW

point of simulation while waiting data from the other two threads. Since Par-2 has a shorter execution time and more work for nodes 1 and 2, the cpu utilisation in those nodes is considerably higher at 53%.

The same case study example as above was used for the experimentation of the power estimation. Parameter values for the power consumption of the components were obtained from literature and are shown in Table 5.11. The power data for ARM9 and 16 k instruction and 16 k data caches were adapted from [10] that describes an experimental SoC design with three power domains, one for the CPU, one for the caches and one for the rest of the SoC. The ITRS2009 data was used for the embedded SRAM. The data predicted for 2009 shows static power dissipation of $1 * 10^{-6}$ mW/bit-cell and dynamic power consumption per cell of $6 * 10^{-7}$ mW/MHz. The power data for the shared bus was adapted from [21]. For the Network interfaces, routers and links, power data was adapted from [23].

The power consumption of the system was simulated at the same time as the performance as the power probes are integral parts of performance probes. Simulation results for the average power consumption and energy consumption with the different JPEG parallelisations are listed in Table 5.12. All parallel versions have

Table 5.12 Power and energy consumption caused by the JPEG encoders

	Sequential	Par-1	Par-2	Par-3
Average power	206 mW	228 mW	255 mW	227 mW
Energy	14.3 mJ	12.6 mJ	9.9 mJ	12.2 mJ

a higher average power than the sequential version. However, since the execution time is shorter, the total energy consumption is lower. The best parallelisation (Par-2) provides about 30% reduction in total energy consumed.

5.6 Conclusions

This chapter described the ABSOLUT modelling and simulation approach. Firstly, it gave an outline view of the approach and its evolution. Secondly, it described how to create different models. Thirdly, it described the means for simulation. Finally, it walked through an example in order to show how things work in practice.

References

1. L. Benini, A. Bogliolo, and G. DeMicheli. A survey of design techniques for system-level dynamic power management. *IEEE TVSLI*, 8(3):299–316, 2000.
2. L. Benini, R. Hodgson, and P. Siegel. System-level power estimation and optimization. In *International Symposium on Low Power Electronics and Design*, 1998.
3. M. Caldari, M. Conti, M. Coppola, P. Crippa, S. Orcioni, L. Pieralisi, and C. Turchetti. System-level power analysis methodology applied to the amba ahb bus. In *The Design Automation and Test in Europe (DATE)*, 2003.
4. E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference (DAC)*, pages 402–405, 2000.
5. N. Dhanwada, I.C. Lin, and V. Narayanan. A power estimation methodology for SystemC transaction level models. In *ACM Proceedings of CODES+ISSS05*, pages 142–147, September 2005.
6. K. Flautner, D. Flynn, D. Roberts, and D.I. Patel. IEM926: An energy efficient SoC with dynamic voltage scaling. In *The Design Automation and Test in Europe (DATE)*, 2004.
7. D. Gajski, J. Zhu, R. Dörner, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000. 313 p.
8. F. Ghenassia, editor. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2005. 271 p.
9. M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.
10. S. Idgunji. Case study of a low power MTCMOS based ARM926 SoC: Design, analysis and test challenges. In *IEEE International Test Conference (ITC)*, pages 1–10, October 2007.
11. R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley & Sons, Inc., 1991. 685 p.

12. T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. UML-based multi-processor SoC design framework. *Transactions on Embedded Computing Systems*, 5(2):281–320, 2006.
13. J. Kreku, M. Eteläperä, and J-P. Soininen. Exploitation of UML 2.0-based platform service model and SystemC workload simulation in MPEG-4 partitioning. In *International Symposium on System-on-Chip Proceedings*, pages 167–170, 2005.
14. J. Kreku, M. Hoppari, T. Kestilä, Y. Qu, J.-P. Soininen, and K. Tiensyrjä. *Languages for Embedded Systems and their Applications*, volume 36 of *Lecture Notes in Electrical Engineering*, chapter Application Workload and SystemC Platform Modeling for Performance Evaluation, pages 131–148. Springer, 2009.
15. J. Kreku, M. Hoppari, T. Kestilä, Yang Qu, J.-P. Soininen, P. Andersson, and K. Tiensyrjä. Combining uml2 application and systemc platform modelling for performance evaluation of real-time embedded systems. *EURASIP Journal on Embedded Systems*, 2008.
16. J. Kreku, M. Hoppari, K. Tiensyrjä, and P. Andersson. Systemc workload model generation from uml for performance simulation. In *Forum on Specification and Design Languages*, 2007.
17. J. Kreku, T. Kauppi, and J-P. Soininen. Evaluation of platform architecture performance using abstract instruction-level workload models. In *International Symposium on System-on-Chip Proceedings*, pages 43–48, 2004.
18. J. Kreku, J. Penttilä, J. Kangas, and J-P. Soininen. Workload simulation method for evaluation of application feasibility in a mobile multiprocessor platform. In *Proceedings of the Euromicro Symposium on Digital System Design*, pages 532–539, 2004.
19. J. Kreku, Y. Qu, J.-P. Soininen, and K. Tiensyrjä. Layered uml workload and systemc platform models for performance simulation. In *International Forum on Specification and Design Languages (FDL)*, pages 223–228, 2006.
20. J. Kreku, K. Tiensyrjä, and G. Vanmeerbeeck. Automatic workload generation for system-level exploration based on modified gcc compiler. In *Design, Automation and Test in Europe conference and exhibition*, March 2010.
21. K. Lahiri and A. Raghunathan. Power analysis of system-level on-chip communication architectures. In *ACM Proceedings of CODES+ISSS04*, pages 236–241, September 2004.
22. P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Kluwer Journal of VLSI Signal Processing*, 29(3):197–207, 2001.
23. P. Liu, B. Xia, C. Xiang, X. Wang, W. Wang, and Q. Yao. A networks-on-chip architecture design space exploration – the LIB. *Computers and Electrical Engineering*, (35):817–836, 2009.
24. Micron Technology, Inc. *System Power Calculators*, 2007. Available at http://micron.com/support/dram/power_calc.html.
25. Micron Technology, Inc. *Mobile DRAM Power-Saving Features and Power Calculations*, 2009. Available at http://download.micron.com/pdf/technotes/tn4612.pdf.
26. J.-Y. Mignolet, R. Baert, T.J. Ashby, P. Avasare, Hye-On Jang, and Jae Cheol Son. Mpa: Parallelizing an application onto a multicore platform made easy. *IEEE Micro*, 29(3):31–39, May–June 2009.
27. J. M. Paul, D. E. Thomas, and A. S. Cassidy. High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors. *ACM Transactions on Design Automation of Electronic Systems*, 10(3):431–461, July 2005.
28. A. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.
29. H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco. System-level performance analysis in SystemC. In *Proceedings of the Design Automation and Test in Europe Conference*, Paris, France, February 2004.

30. S. Thoziyoor, N. Muralimanohar, and N.P. Jouppi. *CACTI 5.0*. HP Laboratories, 2007. Technical report HPL-2007-167.
31. T. Wild, A. Herkersdorf, and G.-Y. Lee. TAPES — trace-based architecture performance evaluation with SystemC. *Design Automation for Embedded Systems*, 10(2–3):157–179, 2006. Special Issue on SystemC-based System Modeling, Verification and Synthesis.

Chapter 6
MPA: Parallelization Made Easy

Geert Vanmeerbeeck and Thomas J. Ashby

Abstract This chapter of the book covers the work performed on IMEC's parallelization tool called MPA. The work done on this tool in the context of the MOSART Project involved several extensions to an already existing baseline version of the MPA tool. In this section we will give an introduction to the entire MPA parallelization tool, and briefly compare our approach to some of the existing alternatives for doing application parallelization. All developed extension in the context of this project will also be explained in more detail. Finally we will conclude this book chapter with a simple example to illustrate the most interesting features of IMEC's parallelization tool.

6.1 Parallelization with the MPA Tool

The general idea of the MPA tool 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. These pieces of code that will be parallelized are called *parsections*.

For each parsection, the designer specifies how many threads must execute it and what part each of these threads must execute. The designer can divide the work over threads in terms of functionality, in terms of loop iterations, or a combination of both depending on what is the most appropriate for a given parsection.

These parallelization directives have to be written in a file provided to the MPA 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 retaining) of multiple parallelizations for the same sequential code.

G. Vanmeerbeeck (✉) • T.J. Ashby
IMEC Belgium, Kapeldreef 75, B-3001 Leuven, Belgium
e-mail: vanmeerb@imec.be; ashby@imec.be

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_6,

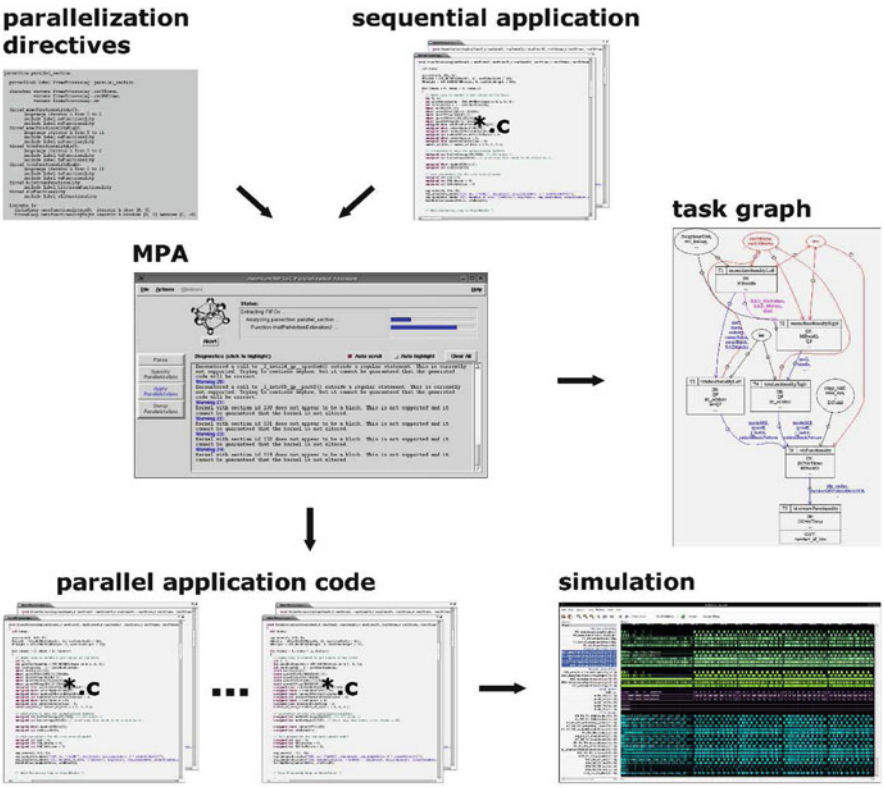

Fig. 6.1 Overview of the MPA tool-flow

Given the input code and the parallelization directives, the tool will generate a parallel version of the code and insert FIFOs and synchronization mechanisms where needed (see Fig. 6.1).

Hence, the designer doesn't have 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 MPA tool provides the optional mechanisms of shared and private variables. By specifying a variable as being shared or private, the designer explicitly tells the tool that it does not have to add synchronization and communication mechanisms for that variable, because he (=the designer) will take care of that himself (or they are unneeded).

The current version of the MPA tool is the scalar version, meaning that it can handle scalars but also arrays (and structs). However, arrays are treated as opaque objects or as "big scalars". If a thread writes to one (or more) element of an array, and another thread (potentially) reads one (or more) element of the array, the whole array will be communicated between the two threads even if the element being read is different from the element being written.

Future work could be to make the tool more intelligent and also analyze which elements or part(s) of the array are being read and/or written. This could potentially save large amount of (unnecessary) data traffic. In our experience however we found this limitation not a big problem. We let the MPA tool handle all (smaller) temporary arrays, while we mark the bigger arrays as *shared variables*. This way we can manage that part of the data traffic by ourselves.

6.1.1 MPA and RTlib: The API

The build-in code generator inside the MPA tool is using a documented yet generic API to separate the parallelized application code from the implementation of platform services related to configuration, communication, synchronization and parallel computing (threads and/or processes).

The implementation of this API for a specific platform or operating system is further called the MPSoC Run Time Library (or RTLib).

This generic set of functions was chosen over a specific implementation to avoid any kind of restrictions with respect to the implementation of any of the required functionality. For example, if the code generator would use a common multi-thread implementation like POSIX threads, the MPA tool could not be used with platforms using Quick-Threads or platforms with other specific micro-kernels.

Hence we have chosen to use a non-standard API for the link between parallel code and platform services. However, as a reference implementation we are also providing an implementation of our generic API on top of the well known POSIX library.

Without going into too much details of this API, there are four main groups of API functions:

- Configuration: this group includes all the `mps_config_XXX()` functions,where `XXX` = `component`, `memory`, `thread`, `source_fifo`, `target_fifo`
- Communication: this group includes the `mps_fifo_put_XXX()` functions, the `mps_fifo_get_XXX()` functions, and the `mps_open_connection()`
- Synchronization: this group includes all the `mps_loop_sync_XXX()` functions, where `XXX` = `init`, `post`, `wait` or `delete`
- Parallel computing: this group includes all the `mps_master_XXX()` functions and all the `mps_slave_XXX()` functions. These functions are used to spawn and join parallel threads (or processes) onto your system, and pass a number of configuration parameters onto them.

The resulting parallelized code, generated by the MPA tool, must be compiled and linked with any version of the RTLib. Here there are two options: either the user wants to do a high level functional simulation of the parallelized code on his development workstation, or he wants to compile the code for a real MPSoC platform (or MPSoC platform simulator). In the first case, the user can link the code

with the functional version of the MPSoC RTLib which comes together with the MPA tool itself. In the other case, the user has to link the code with a platform specific version of the MPSoC RTLib. As mentioned above, the MPA tool comes with an example implementation of the RTLib API build on top of POSIX threads and semaphores. This allows the user to compile and run his parallel application on any Linux OS based system.

6.1.2 MPA and First Order Performance Estimation

The MPA tool also comes with a high-level performance gain estimation approach for your applications. This performance gain estimation is based upon timing annotation of *kernels*. A *kernel* is supposed to be a compute intensive part of the application that does not include any communication and/or synchronization (i.e. an FFT calculation kernel). The approach taken here is to assign execution times to these compute intensive parts of the application (these can be measured from the sequential execution), and annotate these timings during the parallel execution of the application. Upon completion a report is generated with performance details about all the parallel treads involved, the usage of the FIFO communication channels and other synchronization mechanism. Also a global comparison is done to the sequential execution of the application.

For this high-level performance estimation MPA also comes with a tool that allows to automatically insert ARP calls. ARP stands for "ATOMIUM[1] Record an Playback". All function calls for the ARP API start with an `arp_` prefix. ARP calls are inserted into the application code to indicate both the start (`arp_enter_kernel()`) and end (`arp_leave_kernel()`) of kernels. The arguments to these functions are thread and kernel identifiers to keep track of them in a timing database.

This timing database can be obtained by running the application while using the ARP API in *Record* mode. In this mode timing information is gathered throughout the execution of the application and is written in a specific format to a file. This file (database) can then be used in *Playback* mode to annotate the kernels present inside the application.

Note here that the usage of the ARP API is orthogonal to the use of the MPA tool. It is handy for getting early estimations about possible parallel solutions for your application, but it is not required at all. In fact MPA can be used without ARP altogether. Performance analysis of your parallelizations will then have to be evaluated in a different way, however (i.e. platform simulators or emulators).

[1] ATOMIUM is a framework developed at IMEC that consists out of multiple tools including ATOMIUM/MPA. The framework offers C-based source code parsing and analysis capabilities. Other tools within the ATOMIUM framework include an analysis, instrumentation, profiling and memory array usage tool (ATOMIUM Analysis), a tool for doing scratch-pad management or large arrays (ATOMIUM/MH) and a memory footprint reduction tool (ATOMIUM/MC),

6.2 Related Work

6.2.1 Parallelization

There has also been much work on parallelization, whether automatic, aided or largely manual. Our work differs from the standard approaches ([5, 6, 12, 15]) in that the tool is responsible for handling dependencies rather than the programmer, and the emphasis is on being able to rapidly create multiple machine specific parallelizations. Whilst our approach is more ambitious than those widely used tools, MPA is less ambitious than other work in the academic literature in that it isn't fully automatic; however, it is very usable and fills in an important part of the puzzle for semi-automatic parallelization that can target both functional and data parallelism.

Some recent work that is similar to ours in that they target functional parallelism and uses techniques similar to the geometric model includes [1, 10, 13]; however, they seem to only target array dependencies and attempt to do the whole process automatically. Work that seems closer to our approach in terms of the type of dependencies targeted and the embedded focus is [3, 8]; however, they don't mention how they handle array dependencies, and they aim for full atomisation. An alternative to static analysis to recover dependencies is dynamic analysis; examples of applying this to functional parallelism include [9, 16]. A key problem with those approaches though is the reliability of the dependence information and how to map it to communication channels in source code when the analysis is done at the binary level.

The geometric model we use is similar to the polyhedral model used for parallelization in work such as [17] (and the other work they reference), but our end goal is different (mostly checking) and the failure to extract a geometric model for a piece of code doesn't prevent the tool from being useful. Although we do not handle symbolic expressions in the way the polyhedral model can, we also do not suffer the complexity penalty on generated code. Also, that work implicitly implies a shared memory target due to the lack of explicit introduction of communication.

6.2.2 Dataflow Models

The result of a parallelization using MPA could be viewed as a form of dataflow model. There are many dataflow models of computation reported in the literature (for one overview, see [11]), and previous work has reported interesting results on sizing communication channels for instances of such models (see [14]) as well as safety checks such as deadlock detection. However, there are several important differences between using MPA and the approach assumed in the dataflow literature that prevents these techniques being directly reused.

6.2.3 Deadlock

Firstly, MPA-parallelized programs are derived from sequential programs rather than directly constructed. Consequently, parallelisations with unsized FIFOs that do not use loop syncs cannot deadlock during parallel execution[2] as MPA only inserts synchronisation to enforce dependencies present in the original sequential program, which itself is deadlock free by definition. Secondly, using dataflow models to check loop syncs is not easy as the parallelised programs cannot easily be translated into the models due to conditionals, data dependent execution times, loop nesting etc., even for more expressive versions such as cyclostatic dataflow [2].

6.3 MPA Extensions for MOSART

In this section all the extensions developed for MPA in the context of the MOSART Project are explained.

6.3.1 MPA Baseline

At the start of the MOSART project, IMEC already had a (research) tool called SPRINT for doing application parallelizations [4]. This tool however was limited to functional parallelism.

Due to restrictions in the underlying implementation of the tool, it would have been very hard to implement all the extension promised within the MOSART design flow at the start of the project. Mainly here the promise of offering some kind of data-level parallelism would prove to be a very tough case.

For this reason IMEC has chosen to re-implement the functionality already offered by the SPRINT tool onto an existing framework for doing analysis and transformations related to memory accesses. These are also known as Data Transfer and Storage Exploration (DTSE) transformations [7].

This proven framework, known as the ATOMIUM framework, already contained the required basic functionality (like abstract syntax tree generation and variable lifetime analysis) for re-implementing the baseline functionality offered by the SPRINT tool. Additionally, the availability of proven technology for doing memory access analysis, would enable the implementation of all the promised parallelization extensions.

As a result, the re-implementation of SPRINT within the ATOMIUM framework resulted in a new tool name: MPA, which stands for “MPSoC Parallelization Assist”.

[2] Assuming the tool processes the code correctly.

6.3.2 Simulation, Visualization and Kernel Annotation

A first extension developed for the MPA tool is visualization. This visualization depicts the parallelized application's internal structure. It does this by showing all the loops, functions and threads present in the output code and how they are nested. It also shows where the FIFO accesses are performed, relative to loops, functions and threads.

If more then one parallel section is indicated to the MPA tool, it will generate separate views for every parallel section as well. This is done because for larger applications the overall application view rapidly becomes too large and too complex. On these separate visualizations every thread can be identified, all the communicating FIFO queues, and also all the configuration variables (read only) and shared variables.

The output of the visualization is two graph descriptions per parallel subsection, in dot file format that can be displayed by a suitable dot viewer, with the graphs representing either a detailed or high level view of the inserted communication channels (example in Fig. 6.4). This summary of the parallel thread structure and relationships cannot, by definition, incorporate runtime information, and as such does not present the designer with a complete view on what the parallel performance of the application will be like.

To supplement the static view of a mapping, we have implemented a form of dynamic trace generation based on a profiling framework. The ATOMIUM framework, on which MPA is based, has been extended to generate source code level instrumentation of files. In the initial phase, the designer identifies source code constructs (i.e. lines or blocks) of interest with a specific style of label. These labeled elements are termed kernels. A kernel is usually a relatively small, time intensive piece of code. Kernels cannot be nested. After labeling, a tool processes the code to produce the instrumented version. An abridged example of the input and output are given in Fig. 6.2.

As well as the individual kernels themselves, it can be seen that the tool is inserting instrumentation to keep track of the invocation context of a particular execution of a kernel, thus allowing the individual samples to be faithfully replayed. The entry and exit of a kernel also use callbacks supplied by the designer to collect timestamps from the platform on which the instrumented code is being executed, and it is these timestamps in combination with the context information that constitute the profile.

The MPA release package includes a pthread based implementation of the RTLib that will run on a standard workstation, thus providing a functional parallel platform simulator that can be used to check the correctness of parallelized code. Feedback on the potential parallel performance of a given mapping is generated as an addition to this functional simulator. Once a sequential source code has been instrumented, the instrumented version can be parallelized, and the instrumentation is changed to read an existing profile (generated by running the instrumented sequential version) and record events such as kernel execution in a parallel trace, in wave format. This

```
foo() {

  ato_kernel_1:{
    for (i = 0; i<DIM; ++i){
      r += a[i][i];
    }

  }
}

main() {

  foo();

}
```

```
foo() {
  arp_enter_kernel(1, 1, 0);
  ato_kernel_1:{
    for (i = 0; i<DIM; ++i){
      r += a[i][i];
    }
    arp_leave_kernel(1, 1);
  }
}

main() {
  arp_enter(1, 2, 0);
  foo();
  arp_leave(1, 2);
}
```

Fig. 6.2 Example input code and (most of) the result of kernel profiling instrumentation, showing kernel boundaries around the labeled block and subsection boundaries around the function call containing the kernel

trace is then displayed using a waveform file viewer (i.e. Modelsim or GTK-Wave). An example of the output is given in Fig. 6.3.

As the trace is based on a full dynamic execution of the parallelized code, it incorporates all the dynamic behavior of the application for that input set. As such it provides a valuable complement to the existing static information. Also, as the parallel trace is created using timestamps that are pre-collected, the time taken to create the parallel trace is only slightly longer than that required to execute a functional simulation, which is typically fast when executed on a powerful workstation. This can be contrasted with doing a full simulation of the execution of the parallel code on a model of the platform, which may take far longer.

6.3.3 Data Level Parallelism

The idea of data level parallelism is to distribute the computing based upon certain data structures over multiple (parallel) threads (or processors). However, the fact that multiple threads are accessing the same data structures can potentially lead to serious errors like data overwriting.

In the MPA tool we have implemented a way to assign contiguous iteration subranges to different threads. This distribution of loop iterations however will not in any way change the order in which the iterations will be executed. Moreover, if arrays are used in the inner-loop part, MPA will generate a communication and synchronization mechanism to ensure the correctness of the execution. Remember that the current tool version of MPA is scalar based. This means that the *entire* array will be passed through the FIFO every time.

Fig. 6.3 Visualization of timing trace output from high-level simulator. The trace shows thread activity, kernel execution and FIFO occupancy over time

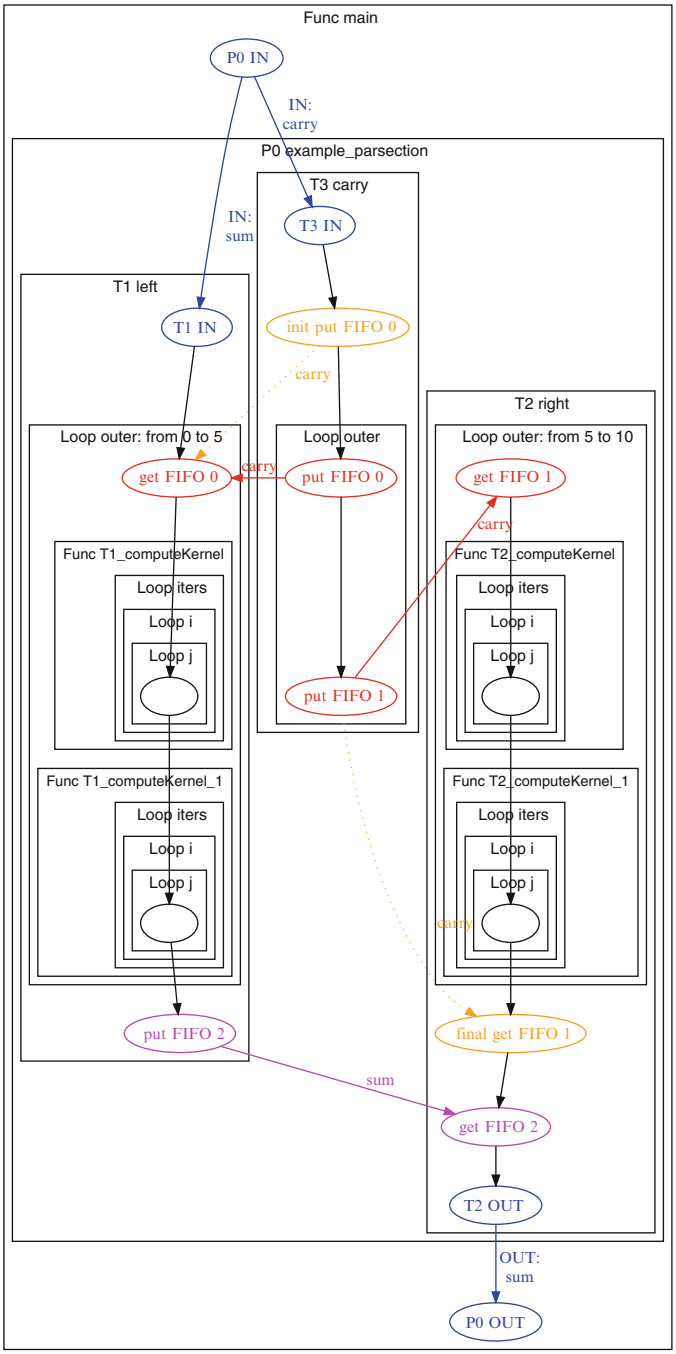

Fig. 6.4 Graphical of internal application structures

```
PAR:{
  for (j = 0; j < 10; ++j) {
    L1: {
      B[j] = A[j] + B[j-1];
    }
  }
}
```

```
thread T1a:
  looprange iterator j from 0 to 5
  include label L1

thread T1b:
  looprange iterator j from 5 to 10
  include label L1
```

Fig. 6.5 Example input code and accompanying parallelization directives for a data-level split of a for-loop using the loopsync extension

However, in a lot of cases arrays are used only as input data. In a lot of these cases the designer can improve the overall result of the data level parallelism by explicitly marking these arrays as shared. For shared variables MPA will *not* introduce any communication or synchronization primitives.

Another way of improving the overall performance of data level parallelism with the MPA tool is to introduce so-called *loopsyncs*. This is an advanced loop synchronization mechanisms to relax the synchronization of distributed loop iterations. The *loopsync* mechanism is also one of the extension implemented within the context of the MOSART Project and will be discussed later in this chapter.

The example in Fig. 6.5 is a simple split of the original loop range into two halves, with each half being assigned to one thread. In the case of no dependencies across iterations, the loop structure is copied to all threads, with the copies being adapted to have suitably modified iteration ranges. In the case of cross iteration dependencies, there must be a data-split FIFO to satisfy the dependency from the last iteration of a given sub-range to the first iteration of the next. In the generated code (see Fig. 6.6), this is placed after the loop for the preceding sub-range, and before the loop corresponding to the following sub-range. Note that the FIFOs are outside the loop bodies.

The analysis required to insert data-split FIFOs is based on looking for loop carried dependencies. These are represented in the Factored Use-Def model (FUD)[3] as dependencies from definitions within the loop back to the mu-def associated with the loop header. Whenever such a dependency exists, there must be a corresponding data-split FIFO, and if there is no such dependency then a data-split FIFO is not required. This analysis is enough to ensure safety for non-shared variables.

6.3.4 Advanced Scalar Dependency Analysis

The implementation of this extension in MPA does a relatively direct analysis of the dependencies between the partitioned sections of code. Such a direct interpretation

[3]Factored Use and Definition chains, or FUD chains, is a data-flow analysis technique often used in compilers for detecting and representing variable dependencies.

```
/* Thread 1 (T1a), remote thread of parsection 1:  */
void T1_T1a(mps_tid_t master, mps_tid_t slave)
{
  ...
  mps_slave_start(0, 1);

  for (j = 0; j < 5; ++j) {
    L1: {
      B[j] = A[j] + B[j - 1];
    }
  }

  mps_fifo_put_data(0, B); /* DS fifo 0: put T1->T2 (B) */

  mps_slave_end(0, 1);
}

/* Thread 2 (T1b), remote thread of parsection 1:  */
void T2_T1b(mps_tid_t master, mps_tid_t slave)
{
  ...
  mps_slave_start(0, 2);

  mps_fifo_get_data(0, B); /* DS fifo 0: get T1->T2 (B) */

  for (j = 5; j < 10; ++j) {
    L1: {
      B[j] = A[j] + B[j - 1];
    }
  }

  mps_slave_end(0, 2);
}
```

Fig. 6.6 MPA Generated code for a simple data-split

can lead to unnecessary sequentialisation of threads that can be avoided by doing a more detailed analysis of the relationship between the different uses and definitions of the variable. In particular, we use an *advanced scalar dependency analysis* to locate and exploit *reduction chains*.

A reduction chain is a chain of variable updates with an associative and commutative operator. For example, the calculation of a sum variable by means of **sum** = **sum** + **something**. If only the end value of the reduction chain is used, and the intermediate values are only used in the update operation to calculate the next value, then the order of computations can be changed. This can be used to relax the otherwise sequential dependencies between the operations. For instance, if the sum variable is computed over several threads, we can compute a partial sum in each thread and send them to a central point where they will be summed together to form the total sum. This allows the threads to run in parallel instead of sequentially. The main use of this is to break what would otherwise be circular dependencies in reductions (such as summations) calculated in loops, where the value of the

```
x = 0;

for (j = 0; j < 10; ++j) {

   x += B[j] - A[j];
}
... = x;
```

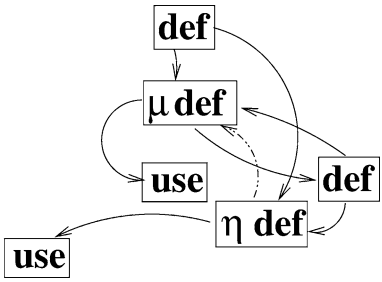

Fig. 6.7 An example of a reduction chain in a piece of pseudo code, alongside the part of the FUD model dealing with the x variable

computed reduction is used after the loop. An example of such a construct and the relevant fraction of its associated Factored Use-Def (FUD) model is given in Fig. 6.7.

The analysis for reduction chains is done on-demand during the creation of the abstract FIFO model that captures the information about which dependencies will result in communication in the mapped version of the code. If the analysis finds a reduction chain then the FIFO model is not created, but rather a reduction chain object is made and attached to the parts of the FUD model that it relates to. This is used later in the flow of the tool to create the actual FIFO communication channels that will be implemented.

The analysis of the reduction chains is base on pattern matching and analysis of the FUD model. The FUD analysis consists of looking for a reduction chain section, where the following uses are only used in defining the same variable (which is the next def in the chain). The pattern matching part is in the analysis of the abstract syntax tree the accepted patterns are certain forms of update statement that link the use nodes to the def node. If a reduction chain section is found, then the tool checks for possible extension of the chain in both directions. This can lead to reduction chains that span several threads and therefore offer an even better improvement of parallelism.

Reduction chains have been implemented for mixed addition and subtraction chains, multiplication chains and min/max chains, giving a wide coverage of different operators. The chains are limited to directly manipulated single variables; variables accessed through pointers and reduction chains that span multiple variables will not be analyzed.

6.3.5 *Loop Sync*

The *loopsync* is a means for a designer to relax the synchronization constraints for a given application to the MPA tool.

The basic idea behind the loopsync is to specify synchronization in terms of the executed iterations of the loop bodies that are being synchronized. A loopsync

```
PAR:{
  for (j = 1; j < 9; ++j) {
    L1: {
      B[j] = 138/2;
      s = 3;
    }
    L2: {
      A[j+1] = B[j] + s;
    }
    L3: {
      C[j] += A[j-1] * 42;
    }
  }
}
```

Original sequential code

```
sharedvar varname A, B
thread T1:
  include label L1
  include label L3

thread T2a:
  looprange iterator j from 1 to 5
  include label L2

thread T2b:
  looprange iterator j from 5 to 9
  include label L2

loopsync LS1:
  threadloop T1  iterator j skew 0
  threadloop T2a iterator j skew 1
  threadloop T2b iterator j skew 1
```

Parallelization Specification

Fig. 6.8 A section of code and a parallelization specification for it, including loop syncs

specifies the allowable relative positions of two (or more) threads whilst executing the synchronized loops. Consider the example split in Fig. 6.8. Given that array **B** is *shared*, the user has specified a loopsync to protect the shared accesses, where threads **T2a** and **T2b** must trail thread **T1** by exactly one loop iteration. Note that this loop sync is not semantically correct as it will lead to incorrect use of array A. the example is constructed this was so that it can also be used later to discuss loop sync checking.

The specification of loop syncs is formulated to make them compact when writing synchronization for multiple threads. Each loop sync in effect specifies pairwise two-sided dependencies between all of the synchronized threads. A two-sided dependency means that both the maximum progress of the first thread with respect to the second and vice-versa is specified. For loop sync **LS1** in the example, this equates to a flow dependency (thread **T1** must have finished executing iteration i before **T2** can begin iteration **i**) and an anti-dependency (thread **T2** must have finished executing iteration **i** before **T1** can begin executing iteration $\mathbf{i} + \mathbf{2}$), although in general both dependencies may be anti-dependencies depending on how the allowable overlap is specified in the loop sync.

In the original sequential code, each execution of a given loop body can be numbered from **0** to $\mathbf{N - 1}$ where **N** is the number of executed iterations, also known as the *trip count*. In order to distinguish different dynamic executions of the loop itself when it is contained in other loops, nested loop iterations can be identified by a vector of values with the number of dimensions equaling the loop nesting depth. These vectors can be uniquely mapped onto machine integers provided the range at any nesting depth is within some reasonable limit. This labeling of iterations is the basis for implementing loop syncs. Each thread has a loop sync counter that it updates each time just before it starts executing the body of a loop that is synchronized. The value used is the one calculated from

```
unsigned int mpa_lv1_curr_L1;
mpa_lv1_curr_L1 =
   MPA_PS1_T1_START_L1_LOOP_ITERATOR_j;

for (j = 1; j < 9; ++j) {
  mpa_lv1_curr_L1 += MPA_PS1_L1_BASE;
  mps_loop_sync_post(0, 0, mpa_lv1_curr_L1,
    MPA_PS1_LS0_T1_MAXSKEW_L1_LOOP_ITERATOR_j);
  mps_loop_sync_wait(0, 0, mpa_lv1_curr_L1,
    MPA_PS1_T1_MINSKEW_L1_LOOP_ITERATOR_j);
  L1: {
    B[j] = 138 / 2;
    s = 3;
    if (j < 5) {
      mps_fifo_put_int(0, s);
    }
    if (j >= 5) {
      mps_fifo_put_int(1, s);
    }
  }
L3:{
       C[j] += A[j - 1] * 42;
  }
}
mpa_lv1_curr_L1 =
  MPA_PS1_T1_END_L1_LOOP_ITERATOR_j;
mps_loop_sync_post(0, 0, mpa_lv1_curr_L1,
  MPA_PS1_LS0_T1_MAXSKEW_L1_LOOP_ITERATOR_j_1);
mps_loop_sync_post(0, 0, 0xFFFFFFF, 0);
```

Generated code for thread T1

```
unsigned int mpa_lv1_curr_L1;
mpa_lv1_curr_L1 =
  MPA_PS1_T2_START_L1_LOOP_ITERATOR_j + 0 * 1;

for (j = 1; j < 5; ++j) {
  mpa_lv1_curr_L1 += MPA_PS1_L1_BASE_1;
  mps_loop_sync_post(0, 1, mpa_lv1_curr_L1,
    MPA_PS1_LS0_T2_MAXSKEW_L1_LOOP_ITERATOR_j);
  mps_loop_sync_wait(0, 1, mpa_lv1_curr_L1,
    MPA_PS1_T2_MINSKEW_L1_LOOP_ITERATOR_j);
  L1: {
    s = mps_fifo_get_int(0);
  }
  L2:{
    A[j + 1] = B[j] + s;
  }
}
```

```
mpa_lv1_curr_L1 =
  MPA_PS1_T2_END_L1_LOOP_ITERATOR_j;
mps_loop_sync_post(0, 1, mpa_lv1_curr_L1,
  MPA_PS1_LS0_T2_MAXSKEW_L1_LOOP_ITERATOR_j_1);
mps_loop_sync_post(0, 1, 0xFFFFFFF, 0);
```

Generated code for thread T2

Fig. 6.9 Generated code for two of the threads from Fig. 6.8

the sequential code by mapping all the iterations to integers. In this way each thread being synchronized can broadcast its progress to the other threads by posting its current loop sync counter value (after updating it but *before* beginning the execution of the loop body). A thread meets its synchronization constraints by executing a **wait** immediately after the **post**. The **wait** command blocks until the other threads have progressed sufficiently for it to release, as determined by the minimum skew of the waiting thread and the maximum skew of the other threads. As loop synchronization constraints act on loop trip counts, they are expressed in terms of trip count distances. The resulting code for the example in Fig. 6.8 is given in Fig. 6.9.

In order to calculate the mapping of iterations in the original code to integers, the total iteration ranges of each individual loop in a nest are extracted from the sequential code by analyzing loop control expressions. The number of bits required to represent that range is then calculated, and the maximum bit range for all loops at a given loop nesting level is recorded. The size of the loop sync counter variable is then the concatenation of these maximum bit ranges. In the case that such a range size cannot be calculated from the control expressions, the tool signals an error to be corrected by the user.

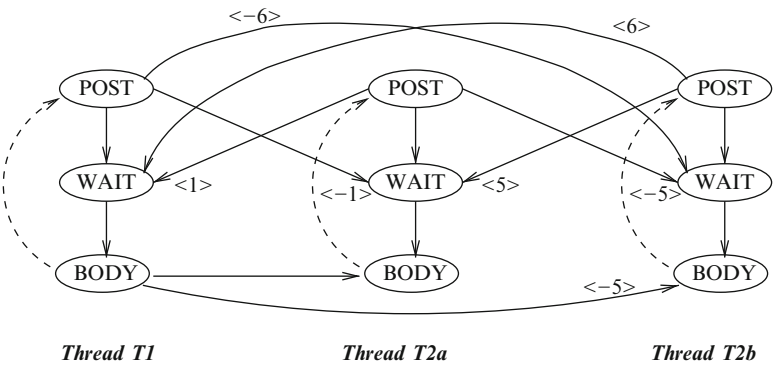

Fig. 6.10 Constraint model for the code and mapping in Fig. 6.8. Nodes are events, edges are dependencies. Dependence distances (in *iterations*) are annotated to edges as vectors except when (**a**) the distance is zero (in which case the annotation is omitted) or (**b**) the edge is a loop iteration back edge (in *dashes*) for which the distance is always 1

6.3.6 Loop Sync Deadlock Detection

Deadlock detection was investigated as one of the possible extensions of MPA as part of the MOSART project. Although there was considerable progress on how to model the problem, the extraction of answers from the model proved to be difficult. The modeling of FIFOs and how they relate to deadlock was also investigated.

The constraints imposed by loop syncs are modeled internally in MPA by representing each synchronized loop in the program as an iterated sequence of three events: ***post***, ***wait*** and ***body***. The ordering of these events is represented in a graph with directed edges to denote precedence. Each edge is labeled with a vector to denote the minimum distance (i.e. number of loop iterations) that the dependence is valid for, being the tightest synchronization constraint that it can represent. For example, the constraint model for Fig. 6.8 is shown in Fig. 6.10. The edge from the ***body*** node to the ***post*** node represents iteration in the loop. The edge from the first ***body*** node to the second indicates the dependence due to the FIFO from thread T1 to T2 to communicate variable s.

Let us consider what will happen if the user now adds a second loop sync to protect accesses to shared array A, with the following form:

```
loopsync LS2:
  threadloop T1  iterator j skew 1
  threadloop T2a iterator j skew 0
  threadloop T2b iterator j skew 0
```

The constraints become mutually inconsistent as ***T2*** cannot also be ahead of ***T1*** by exactly one iteration. The model can be checked for this by searching paths from a node back to itself where the values on the path sum to a lexically negative or

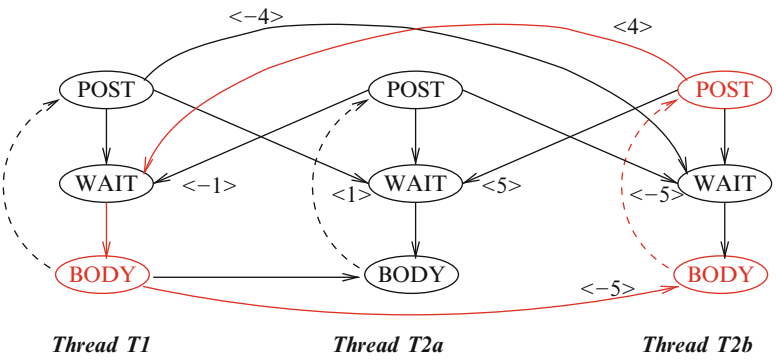

Fig. 6.11 Constraint graph resulting from the second loopsync, with the first loopsync omitted for clarity. One of the paths showing that the loopsync is illegal is highlighted in *red*

zero vector. This implies that a particular instance of an event must occur before or simultaneously with a previous instance, which indicates an illegal graph and therefore inconsistent constraints. As well as covering dependencies between loop syncs, the technique will also show when a loopsync violates the flow dependencies that arise from dependencies across threads due to the partitioning (that will result in FIFOs). For instance, even without loop sync LS1 in Fig. 6.8, LS2 is illegal as it contradicts the FIFO inserted for variable s. An example of such a path can be seen in the constraint model in Fig. 6.11.

Nested loops are modeled by associating a body node with the graphs of any loops nested in it. These nested models are then used to calculate a dependency between body nodes in the parent graph by summarizing the dependencies in all nested loops. Data splits at a given loop level require a further refinement of how the model is constructed. FIFO dependencies between functional splits should only be inserted between loops whose sub ranges actually overlap (and hence actually need a FIFO), but the distances of these and loop sync dependencies remain unchanged. Loop carried FIFOs in a loop split that result in a FIFO from after the loop for the first sub range to before the loop for the second sub range (loop split FIFOs) are modeled in the parent loop as a FIFO with distance 0.

The specification, modeling and checking of loop syncs for loop split sub ranges that do not overlap is done on the basis of treating the loop sync as if the iteration sub ranges were extended such that they would overlap. This has several consequences. The first is that loop syncs for non-overlapping sub ranges are implemented slightly differently from the non loop-split case. As loop syncs are interpreted as if for extended overlapping iteration ranges, a loop sync constraint with skew 0 for two threads with abutting loop sub ranges results in sequential execution. To specify via a loop sync that two non-overlapping sub ranges must be executed simultaneously, the latter sub range must be shifted forwards (with negative skew) relative to the former. The second is that the checking can give a warning for a set of loop syncs that are not in practice directly conflicting. However, we consider it more important to maintain semantic consistency than to allow such corner cases.

The constraint graph is simplified before searching to minimize the number of edges, then search is performed using standard graph algorithms. Search is performed per loop nesting level, starting from the bottom upwards. Due to time limits, we only had the chance to explore the use of relatively unsophisticated graph search algorithms. On the sizes of graphs that we encountered, these proved to be unacceptably inefficient. The rationalization of the model and/or the use of more sophisticated algorithms to make full checking feasible is left as future work.

6.3.7 FIFO Sizing

Unsized FIFOs give rise to uni-directional synchronization in the parallel code, by design, to enforce dependencies. However, relying on unlimited FIFO storage for an actual platform has the disadvantage of potentially unpredictable behavior or even failure of the program due to resource overuse. As such, MPA should give sizes to the FIFO objects that it inserts where possible. Sized FIFOs also introduce synchronization in the opposite direction though (put will block when the FIFO is full), so we should ensure that the sizes we assign are at least safe, in that they don't introduce deadlock in the parallel program, and if possible efficient, in that they represent an appropriate trade-off of space for parallelism.

Individual FIFO sizing could be based on the same constraint model as loop sync (and FIFO constraint) checking, by looking for minimum legal values rather than checking for inconsistent existing values. However, this brings with it the same problems as encountered in loop sync checking, as the same information about implied constraints needs to be extracted from the model. Hence we chose to implement a global assignment of FIFO size through a parameter set by the designer. This has the advantage of not requiring any expensive analysis and having a conceptual uniformity. Further investigation into individual FIFO sizing is future work.

6.4 MPA by Example

In this section we will take a small example through IMEC's MPA tool to illustrate the tool flow and some of the capabilities of our approach.

6.4.1 The Example

In Fig. 6.12 you can find the (simplified) input source code that we will be using throughout this example section.

Note that the input sources have been manually annotated (once) with so-called *block-labels* (see lines 7, 11, 15 and 19 in Fig. 6.12). Annotating sources with block-

```
4  int main(int argc, char** argv)
5  {
6    int outer;
7    float sum = 0, tmp;
8    float carry = 1.2;
9
10   parsection1 : {
11
12     for (outer = 0; outer < 10; outer++) {
13
14       store_in_temp : {
15         tmp = computeKernel(carry);
16       }
17
18       compute_carry : {
19         carry = carry + 0.001;
20       }
21
22       calc_sum : {
23         sum += computeKernel(tmp);
24       }
25     }
26   }
27
28   printf ("%f\n", sum);
29
30   return 0;
31 }
```

Fig. 6.12 The example source code

labels do not introduce performance penalties. It is required by the tool that all code within a *parsection* is within the boundaries of (at least one) a block-label. The parallelization will be specified by stating which blocks should be executed where (inside which thread), however there is no default assignment and hence code not within the boundaries of a block-label cannot be assigned to a thread (resulting in a tool error).

6.4.2 Trivial: Everything in One Thread

To start we will use to tool to generate a trivial example: put all the code inside the *parsection* in one single thread. This will (almost) result in the original application. However this way we can analyze the resulting code, and for the other upcoming examples, we can compare the results to this one.

Here is the parallelization specification:

```
processor P1

parsection trivial_parsection:

  parsecblock label main::parsection1

  thread all:
    include label store_in_temp
    include label compute_carry
    include label calc_sum
```

This specification is located in a separate file and will be passed as an argument to the tool when we perform the actual parallel code generation.

From this simple parallelization specification, it is clear that we declare one single thread (named *all*) that spans all the (block-) labels present in the application. As a result we expect to see one Master thread and one Slave thread. The Master thread is always present, it is this thread that starts up the application and runs all the code before and after the parsection. It is also the Master thread that spawns and activates all the slave threads at the start of the parsection, and that waits for all slave threads to have finished at the end of the parsection. Slave threads are spawned and activated by the Master thread and execute (parts of) code that is located within a parsection.

When running the MPA tool, the internal structure and dependencies of the generated parallelization can be analyzed graphically. The plot in Fig. 6.13 is generated by MPA inside the output directory specified as a dotty graph (see www.graphviz.org for details). From this graph you can see the functions at hand, the nesting of all the loops and all the configuration and communication present in the resulting code. Also the boundaries for the parsection and for all the generated threads are made visible in this graph.

6.4.3 First Functional Split

Next, we will create our first real parallel version for this example. We will split of calc_sum into a separate thread, and thus creating a kind of Functional split or functional pipeline for this example application.

For this functional pipeline we will require two threads. We will put the `calc_sum` block into one thread and the rest in the second thread.

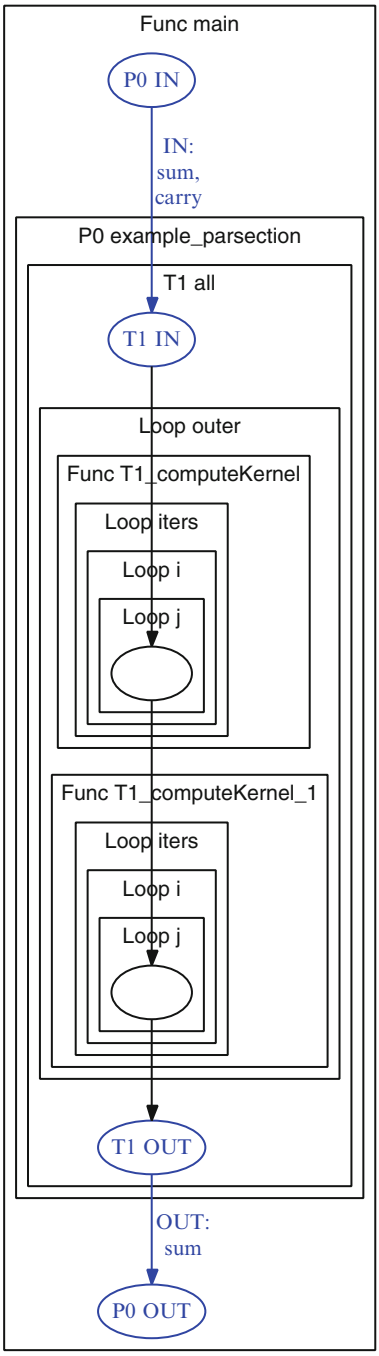

Fig. 6.13 Graphical representation of parallelization structure for trivial case

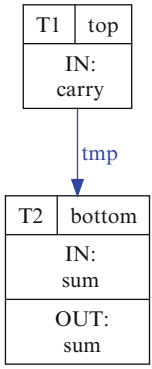

Fig. 6.14 Simplified graphical representation of functional split

Here is the parallelization specification:

```
processor P1

parsection example_parsection:

  parsecblock label main::parsection1

  thread top:
    include label store_in_temp
    include label compute_carry

  thread bottom:
    include label calc_sum
```

In Fig. 6.17a you can see the graphical representation of the resulting functional split. In this graph we can clearly see that a communication channel (FIFO) was required for variable `tmp`.

There is also a more simplified version of this graph available, not indicating any of the nested loops and/or internal structures, but simply illustrating all threads with their names, all queues (+ for which variable) and all thread configuration parameters. see Fig. 6.14 for the representation of this functional split.

In the resulting waveform in Fig. 6.15 we can see that this communication channel, although blocking in nature (blocking read when empty, blocking write when full) does not affect the overall performance of the parallelization. In fact from the thread activity waves we can see that in steady state both threads are almost continuously active.

The same conclusions can be drawn from the simulation report. This report is generated upon the end of simulation by the HLsim library. It reports the accumulated execution times for every thread, waiting times for queues, number of read and writes for every FIFO, etc. All these numbers are also compared against the same behaviour but then in one single (sequential) thread. This to give the designer an idea of the overall performance gain of his parallelization effort.

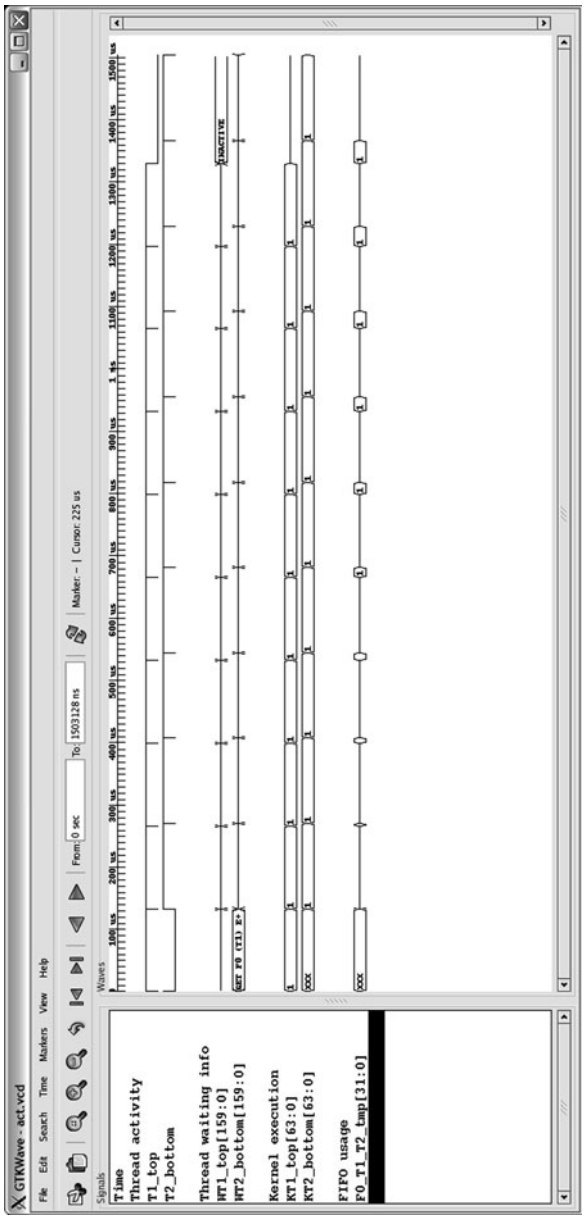

Fig. 6.15 Waveform for functional split

Here is the overall performance reporting that can be found in the report file (report filename is `mps_hlsim_report.txt`:

```
Parsection 1:

  #executions:             1
  tot #cycles:       1530128
  avg #cycles:       1530128
  min #cycles:       1530128
  max #cycles:       1530128
  active    :         100.0%

  tot processing cycles:      2700000
  speedup factor:                1.76x
  #threads:                         2
  speedup per thread:            0.88x
```

From this we can conclude that there are 2 parallel threads present in this parsection, and that the overall execution time compared to the single threaded (or sequential) execution of the application is 1.76 times faster for this parallel execution. Note that due to the way communication is simulated within HLsim, that this is more of a theoretical speedup factor. In a real implementation this number will reduce due to bus arbitration and communication overhead. Obviously, the closer the speedup factor is to the number of threads present, the better the result. Or the better the available parallelism in the application was exploited.

In addition to the overall performance report listed here above also a detailed report can be found per thread and per communication channel. Here is from the same report file the details for thread 2 and also for FIFO 0:

```
Thread 2 (bottom):

  tot #cycles active:          1530128
  avg #cycles active:          1530128 (100.00%)
  avg #cycles idle:             130008 (  8.50%)
  avg #cycles busy:            1400120 ( 91.50%)
  avg #cycles processing:      1400000 ( 91.50%)

  parallelization overhead:

   avg #cycles waiting for loop sync:            0 (  0.00%)
   avg #cycles waiting for FIFO:            130128 (  8.50%)

     FIFO put:                0 (  0.00%)

       pre put:               0 (  0.00%)
       FIFO full:             0 (  0.00%)
       put copy:              0 (  0.00%)
       post put:              0 (  0.00%)

     FIFO get:           130128 (  8.50%)

       pre get:              40 (  0.00%)
       FIFO empty:       130004 (  8.50%)
       transfer:              4 (  0.00%)
       get copy:             40 (  0.00%)
       post get:             40 (  0.00%)

...

FIFO 0: T1 -> T2 (tmp):

   elem size:                          4
   fifo depth:                         0
```

```
tot #elems transfered:                10
avg #elems transfered:                10
avg #bytes transfered:                40
max #elems in FIFO:                    1
avg #elems in FIFO:                    1
avg #times FIFO empty                 10 (100.00%)
avg #times FIFO max usage:            10 (100.00%)

avg #cycles waiting for FIFO:       130248 (  8.51%)

FIFO put:               120 (  0.01%)

  pre put:               40 (  0.00%)
  FIFO full:              0 (  0.00%)
  put copy:              40 (  0.00%)
  post put:              40 (  0.00%)

FIFO get:            130128 (  8.50%)

  pre get:               40 (  0.00%)
  FIFO empty:        130004 (  8.50%)
  transfer:              4 (  0.00%)
  get copy:              40 (  0.00%)
  post get:              40 (  0.00%)
```

From this report we see that FIFO 0 is a communication channel between threads `T1` and `T2` implemented to transfer the `tmp` variable which has a size of 4 (bytes). The depth of this FIFO is reported as 0 (NIL). For the HLsim simulator a zero depth FIFO means unbound or with infinite depth. The advantage of having an infinite sized FIFO is that often you can extract FIFO depths from running the simulation. Here for example the maximum number of elements in the FIFO is reported as 1. This means that a FIFO with a depth of 1 would be sufficient for this queue. Although making the FIFO with a depth of 2 (also known as ping-pong buffer) would probably be better to avoid read/write collisions.

From both the details of thread 2 and FIFO 0 it is clear that the HLsim simulator is implementing a polling based FIFO communication scheme: there are way more FIFO get requests than there are FIFO put requests.

Functionally however it is correct since the actual number of FIFO writes (listed in *put copy*) and FIFO reads (listed in *get copy*) are identical.

6.4.4 First Data Split

Now we will illustrate the *data split* capabilities of MPA. Note however that in this context data-split needs to be interpreted as being a kind of loop-iteration parallelization.

In practice this will mean that from all the required iterations a subset will be executed inside one slave thread, while the rest will be executed inside another slave thread. This approach however is not limited to two slave threads. In principle as many slave threads as there are loop iterations could be inserted by the tool.

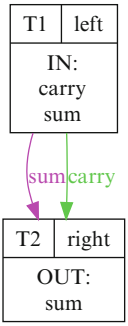

Fig. 6.16 Simplified graphical representation of data split

Here is the parallelization specification to use:

```
processor P1

parsection example_parsection:

  parsecblock label main::parsection1

  thread left:
    looprange iterator outer from 0 to 5
      include label store_in_temp
      include label calc_sum
      include label compute_carry

  thread right:
    looprange iterator outer from 5 to 10
      include label store_in_temp
      include label calc_sum
      include label compute_carry
```

This specification states that all kernels will be executed by both slave threads, however iterations 0 up to (but not including) 5 will be calculated by thread *left* whereas iterations 5 up to 10 will be calculated by slave thread *right*.

In Fig. 6.16 you can see the simplified graphical representation of this data level split.

Note from Fig. 6.17b that for the `sum` variable a reduction chain was detected by the MPA tool. There is no explicit communication channel generated for this variable, while upon first glance an inter-thread dependency is to be expected similar to the `carry` variable. However thanks to the build in reduction chain analysis the loop iteration dependency was removed by the MPA tool. This enhances the application speedup since now both slave threads can run in parallel without having to synchronize upon every iteration. For the final correct behaviour a special kind of communication channel, called reduction chain FIFO, is inserted for variable `sum`: at the end of thread T1 the thread *left* is sending its partial result for `sum` towards thread *right*. Here the final result for `sum` is calculated and sent back to the Master thread.

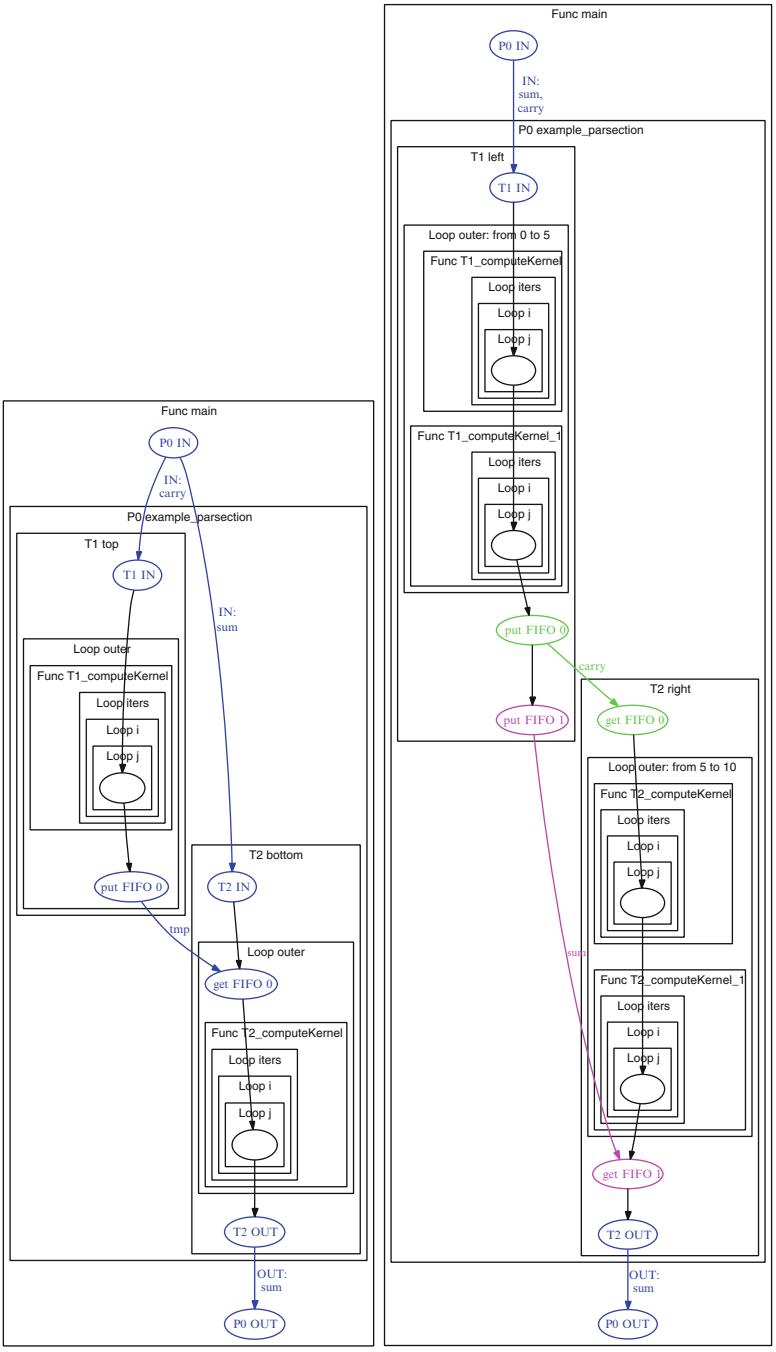

Fig. 6.17 Graphical representation of (**a**) functional and (**b**) data split

6.4.5 Combined Functional/Data Split

In this section we will illustrate a combination of the two previous techniques: a combined functional and loop-parallel application split.

Here is the parallelization specification that we will use:

```
processor P1

parsection example_parsection:

  parsecblock label main::parsection1

  thread left:
    looprange iterator outer from 0 to 5
      include label store_in_temp
      include label calc_sum

  thread right:
    looprange iterator outer from 5 to 10
      include label store_in_temp
      include label calc_sum

  thread carry:
    include label compute_carry\\s
```

Upon completion of the MPA tool, the parallelized application code is being compiled and run with a timing-annotated software simulator (called High-level simulator or HLsim). With this tool we can evaluate how much parallelism we have really brought to the application, compared to a sequential execution. It does this by simulating the generated software, taking timing annotation values for the kernels into account and while respecting all communication and synchronization between threads.

In Fig. 6.18 you can see the simplified graphical representation of this combined functional/data level split.

In addition to this textual report about the activity and waiting times for every thread, FIFO and synchronization primitive, it is also possible to have the simulator generate a waveform trace of the execution of the parallelized application.

If we do this for the latter version of the example at hand, we get a waveform similar to Fig. 6.19. Here we can see the activity for every thread, the depth counters for every FIFO and if a thread is blocked or waiting, it shows who or what it is waiting for.

The resulting graphical representation can be found in Fig. 6.20. From this graph we can indeed see that three threads are present in the parallelized code: T1(left), T2(right) and T3(carry). FIFO communication (and hence synchronization) is required for variable `carry` between threads T1 and T3 and between threads T2 and T3. In addition a special reduction chain FIFO is present at the end of thread T2 sending the (partial) value of variable `sum` to thread T3. To illustrate also the correct-by-construct approach of the MPA tool,you can see that for FIFO0 an initial

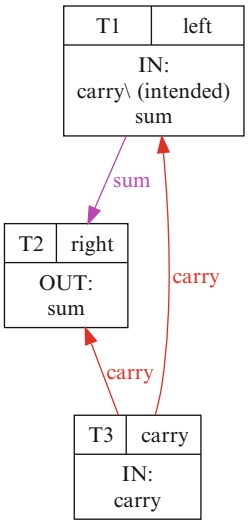

Fig. 6.18 Simplified graphical representation of the combined split

FIFO-put action was generated, while for FIFO1 a finial FIFO-get action was put in place. This is because the tool has detected that these were both required for a complete synchronization of the application, and of all its slave threads.

6.5 Summary and Conclusions

In this chapter a source-to-source transformation tool called MPA was presented that is able to introduce parallelism into the application. The parallelism is defined by the designer in a separate text-file, and uses labels that were introduced into the original application source code. This approach allows for an easy evaluation of different parallelization options without any changes to the application itself.

The MPA tool uses the underlying framework to do variable lifetime and dependency analysis and will based upon this information introduce the required synchronization and communication mechanisms between the generated parallel threads in the transformed application code.

MPA uses a generic API to separate the platform specific part of the implementation (threads, communication, synchronization) from the generated application code itself. This way MPA can be used to target multiple platforms and Operating Systems.

Within the context of the MOSART Project a number of extensions were developed to improve the capabilities of the MPA tool. These extensions include annotation and visualization of the resulting parallelization and the introduction of data-level parallelism to the output.

The bottom half of this chapter took a simple example in tutorial style through the MPA tool flow, illustrating the capabilities and features of the tool itself.

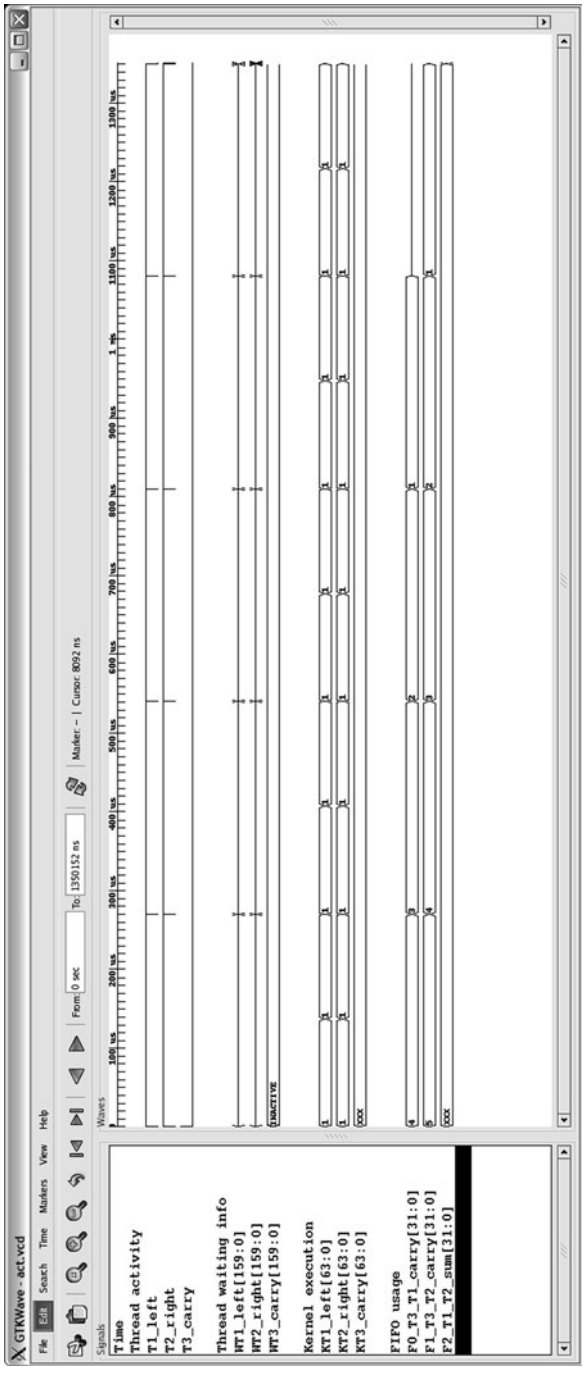

Fig. 6.19 Waveform for combined functional/data split

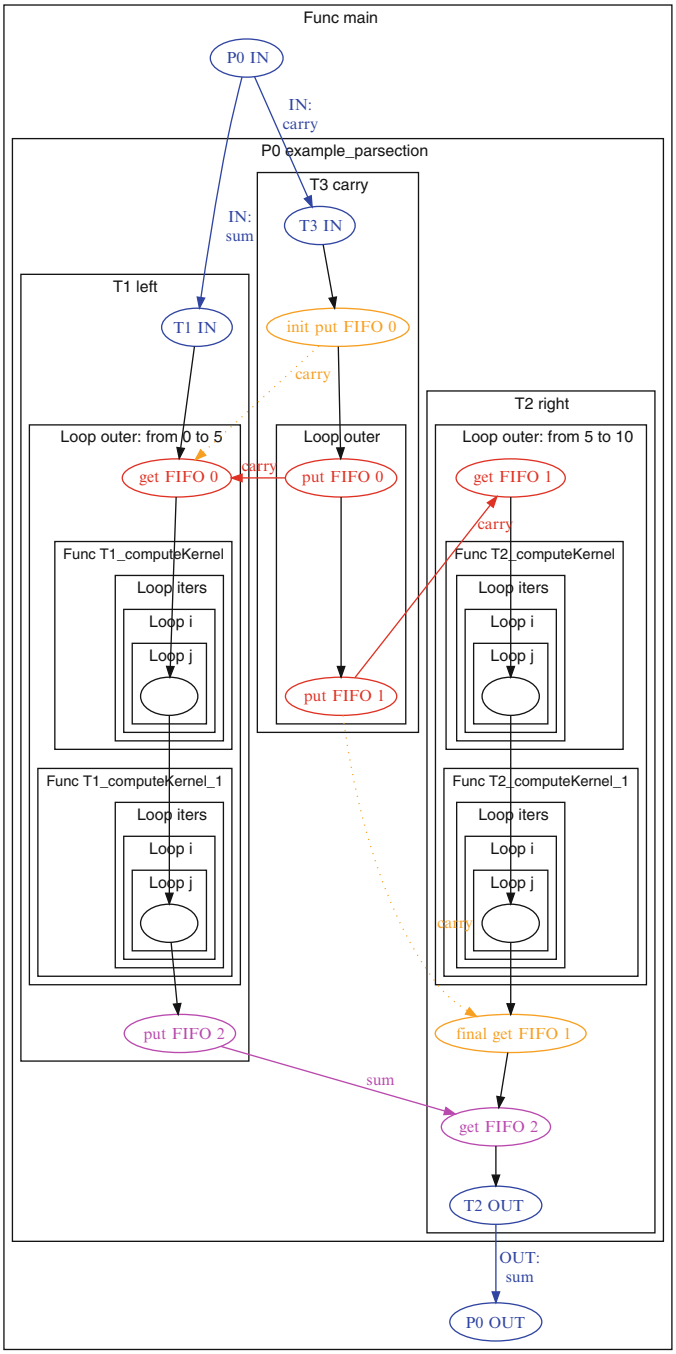

Fig. 6.20 Graphical Representation of combined functional/data split

References

1. Associated Compiler Experts BV. Parallelization using polyhedral analysis. White paper cosy-8153-polyhedral, Amsterdam, March 2008.
2. G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255 –3258 vol.5, 9-12 1995.
3. J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. Maps: An integrated framework for mpsoc application parallelization. *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 754–759, June 2008.
4. Johan Cockx, Kristof Denolf, Bart Vanhoof, and Richard Stahl. Sprint: a tool to generate concurrent transaction-level models from sequential code. *EURASIP J. Appl. Signal Process.*, 2007:213–213, January 2007.
5. Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. 5(1):46–55, January/March 1998.
6. ISO/IEC/JTC 1/SC 22. Iso/iec 9945-1:1996, portable operating system interface (posix) – part 1: System application program interface (API), November 1996.
7. Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Drdu: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst.*, 12(2):15, 2007.
8. T. Isshiki, M. Z. Urfianto, A. U. Khan, D. Li, and H. Kunieda. Tightly-coupled-thread model: A new design framework for multiprocessor system-on-chips. In *Design Automation Symposium (Japan)*, number 7, pages 115–120. Tokyo Institute of Technology, July 2006.
9. Ireneusz Karkowski and Henk Corporaal. Fp-map - an approach to the functional pipelining of embedded programs. In *HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing*, page 415, Washington, DC, USA, 1997. IEEE Computer Society.
10. Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 13–17, New York, NY, USA, 2000. ACM.
11. Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
12. Message Passing Interface Forum. MPI: A message-passing interface standard, June 1995.
13. Antoniu Pop, Sebastian Pop, and Jan Sjodin. Automatic streamization in gcc. In *GCC Developer's Summit*, 2009.
14. Sander Stuijk, Marc Geilen, and Twan Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Computers*, 57(10):1331–1345, 2008.
15. The Multicore Association. Multicore communications API specification v1.063 (MCAPI), March 2008.
16. William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.
17. Konrad Trifunovic, Albert Cohen, David Edelsohn, Li Feng, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. Graphite two years after. In *GCC Research Opportunities Workshop (GROW)*, 2010.

Part III
Industrial Applications

Chapter 7
MPSoC Architecture Performance Analysis for Agile SDR Radio Applications

Sylvain Aguirre and Bernard Candaele

7.1 Trends on SDR/Agile Radio and Supporting Multi-Core

7.1.1 Multi-Core Trends in Mobile Terminals

With the endless transformation of telecommunications in the last decades by the market (higher connectivity, deregulation, globalization, more mobility and new services) and related technology innovations, ubiquitous access to all types of media, data, audio or video is becoming a reality.

In the convergence trends of, communication access everywhere and associated IT plus multimedia services, the system definition is becoming extremely complex as well as the product development. The progress in microelectronics with multi-cores allows further integration but as such requires new development methodologies.

For embedded systems such in wireless mobile terminals, the more demanding algorithms in signal processing, image processing and high bandwidth connectivity have been enabled thanks to multi-cores platforms, while providing multi-modes operations, preserving low power and autonomy (Fig. 7.1):

- First generation was addressed with ASICs then programmable base band SoC such for the codec, the modem and radio protocol stack. The transition to multi-standard i.e. GSM, Edge, GPRS, UMTS … were enabled with programmed radios.
- New generation devices integrates general-purpose multi-core processors, and a number of domain specific many-core subsystems, such modems (dealing with GSM, HSPA, UMTS, CDMA, WIMAX, LTE/LTE-A), codecs and application

S. Aguirre (✉) • B. Candaele
Thales Communications and Security, Paris, France
e-mail: Sylvain.AGUIRRE@fr.thalesgroup.com; bernard.CANDAELE@fr.thalesgroup.com

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_7,

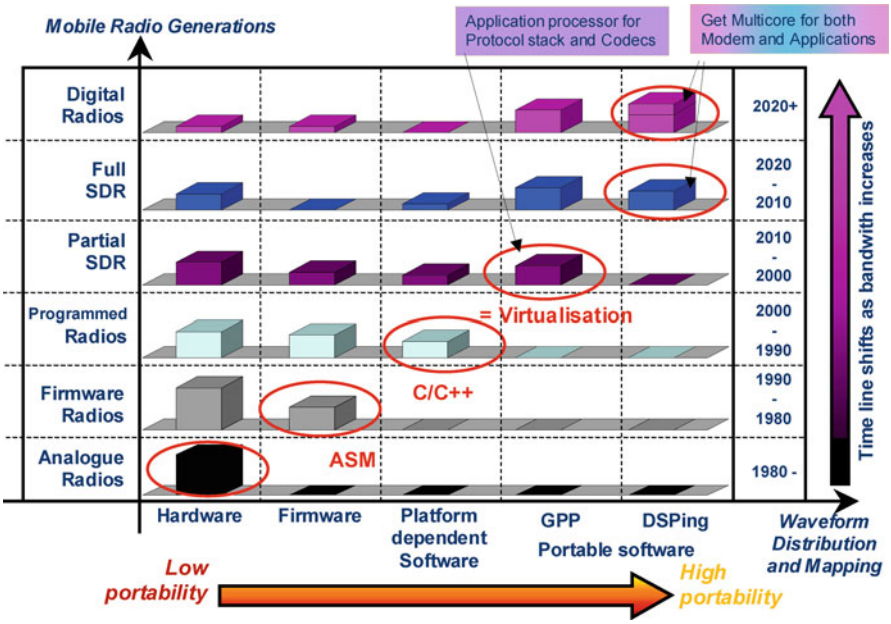

Fig. 7.1 Wireless waveform and application portability per generation

processors. At this stage, different programming models exist according to the processing domains. The application processor allows software portability and has enabled SOA (Service Oriented software Architecture), while the reuse of legacy code from previous generation devices, incremental software designs and the general paradigm of parallel programming onto multi-core are main challenges. The transceiver terminals have to cope high demanding processing, while being flexible enough to run multiple standards and while achieving design constraints: real time processing, efficiency and flexibility, small footprint and low power consumption, and cost.

- The architecture of new high end mobile phones is now resuming in two chips: 1 baseband and 1 radio standardized on their interfaces (such DigRF), then coupled to the application processor.

7.1.2 Considered Challenges

New methodology and design approaches are mandatory to deal with the high demanding embedded applications as described above.

The existing and further growing parallel architectures have raised the inherent difficulties in parallel programming to the front stage, and is demanding research into novel parallel processing architecture and software programming. In the frame

of MOSART program, several enabling technologies have been researched and exercised:

- Parallel multi-core architecture
- Distributed but shared memory: enabling architecture and real time performance to face the memory wall
- Multi-core architectural execution models: what parallel abstractions should the hardware provide to the system engineer
- Runtime when making the application parallel: how to characterize this in the system engineering phase?
- Also will abstraction executive models allow to identify and fix such interconnect congestion issues and energy dissipation

The test case will also exercise directions in programming parallel embedded systems onto heterogeneous platforms.

How to get a flexible solution? Flexible in the sense that a modification of the specification of the application can be quickly implemented in the system.

7.1.3 Use Case: New Processing to Support Future Flexible Radios

The considered use case by Thales is the future agile spectrum, also called cognitive radios. The Cognitive Radio (CR) concept was posed in 1999 by J. Mitola III, the Software Define Radio's (SDR) father. It defines self-aware radios that continuously adapt to their surrounding environment. Face to the increasing number of wireless communication protocols and to the limited spectrum of frequencies, the flexible spectrum SDR, named Cognitive Radio, is expected to be in charge of analyzing the spectrum resource in order to identify free channels for transmission purpose.

J. Mitola defines 9 levels of cognition and the corresponding capabilities starting form level 0 – the pre-programmed radio – up to the level 8 which corresponds to the ultimate version that autonomously proposes and negotiates new protocols.

The Cognitive Radio challenges are several. Amongst them, building networks of CRs will necessitate to introduce intelligence in the mobile radios and in the network along with the mechanisms that enable the self-awareness such as spectrum sensing. Solutions need to be developed to allow distributing this intelligence across all the different functional layers of the system. This constrains to build the radios and the system as a whole and make sure that the distributed intelligence cooperates toward a common target goal. This will necessitate to develop new waveforms in a cross-layer approach that will be able to integrate the spectrum sensing feature. The platform that will be hosting the CR is an important parameter since it conditions its capabilities to serve the CR features. The target platform is a Software-defined Radio (SDR) one (Fig. 7.2) with wide bandwidth transceivers covering several bands from 100 Mhz up to 6 Ghz, multiple RF chains, on the fly reconfiguration, ...

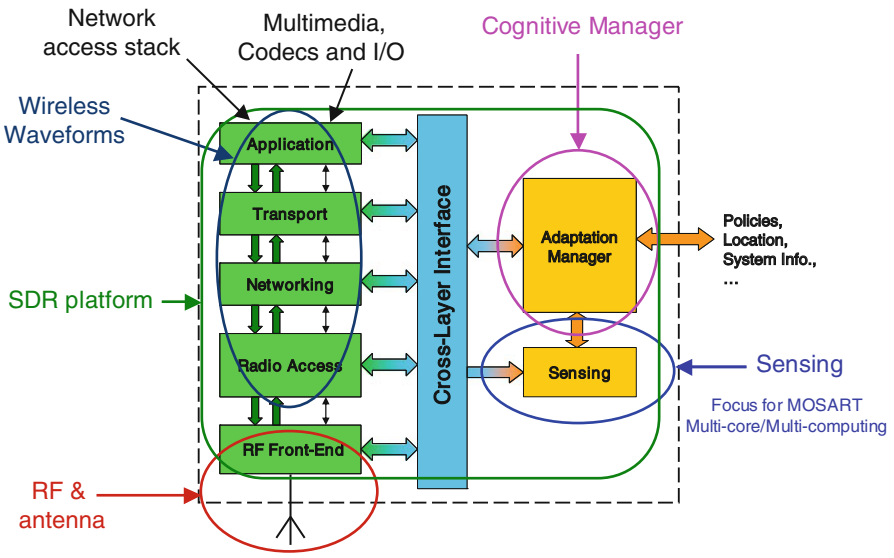

Fig. 7.2 The building blocks for a cognitive radio terminal

Related researches are developed in the frame of the EC FP7 Work Program "The Network of the Future" such to bring solutions and to impact the worldwide telecommunications. Flexible SDR radios are also expected to give new industrial opportunities, facilitating the introduction of mechanisms for dynamically sharing the processing resources among different communications standards as reaction to actual load shifts.

Such in June 2010, the Federal Communications Commission (FCC) released access to vacant TV band UHF channels and allowed limited sensing services on devices using these new released bands, like channel characterisation and energy detection. Consequently, regulation authorities and operators will ask for advanced monitoring to control the spectrum usage.

Thales Communications and Security, which is a major player in the radio communication system market and spectrum monitoring system for operators, addressed cognitive radio focusing on the sensing algorithm as part of MOSART study. The sensing processing requires a huge computational power with a low level of power consumption within the mobile terminal, but we imagine thanks to the More Moore technology, this processing will be affordable in the next years (Fig. 7.1).

Heterogeneous many-core SoC is the targeted technology for ensuring reconfigurability of terminal to support both the multiple waveforms and the applications under very high performance requirements. Many new issues have to be overcome to create this type of circuits. In that context, the MOSART project proposes an innovative and efficient design development methodology to build a relevant system that fits with this type of applications.

Due to the overall complexity of Cognitive Radio application in terms of computation power and memory size, the use case has been focused on the *sensing* algorithm. The perimeter of Thales experiment is built on two main steps:

– Transposition of a GSM bandwidth signal into base-band (referred as step 1 in the rest of this document).
– Channel extraction of the incoming wide band radio signal (referred as step 2).

The objective and approach to address heterogeneous many-core embedded system issues are described in terms of design methodology and tool flow. A coarse-grain partitioning of the application is detailed as a preliminary step towards a more accurate platform architecture design space exploration and mapping.

Finally, results related to each MOSART technology and tools that have been experimented all along the ICT-FP7 project are reported to value the MOSART approach in the MPSoC design domain:

– Code profiling for accelerators and code marking for parallelisation
– Generation of a parallelised code version under coarse grain estimation of the performance
– Computing Performance Profiling per MPSoC configuration and selection of the best multi-core configuration

7.2 MPSoC Design Methodology and Tool Flow

7.2.1 Design Methodology Overview

This paragraph describes the design methodology exercised by Thales during the MOSART project and the associated tools and technologies. Figure 7.3 shows a high level view of the MOSART methodology for MPSoC design. It begins with a sequential description in C code of an embedded application from which two explorations are achieved at both processor architecture and code partitioning levels.

The left side of the methodology (left hand box) consists in identifying the algorithms to be accelerated (while preserving flexibility) with the best processor architecture through Design Space Exploration (DSE): the resulting structure leads to a so called Application Specific Integrated Processor (ASIP) that it is supposed to be the most suited processor to meet the application requirements in terms of computational power, power consumption and local memory accesses. The resulting platform is made up of a set of the previously defined ASIP and it is claimed as homogeneous platform.

On the other hand, a more application specific, and then efficient, platform can be defined. In that scope, several different processor architectures are needed to be part of the targeted platform. The partitioning of the software – and the associated number of split codes – is obtained once exploration on concurrency extraction of the application is done.

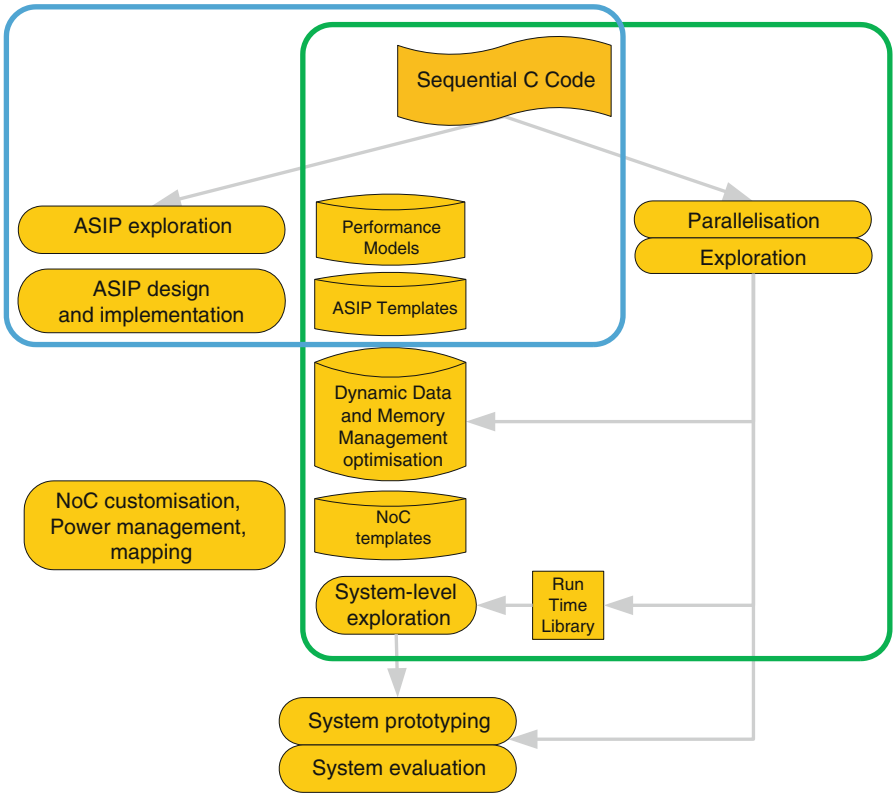

Fig. 7.3 Overview of the MOSART MPSoC design methodology

From the software programming part, another exploration is expected on software parallelisation, as shown in Fig. 7.3 (right hand box). The first objective is to identify part of the code that can be executed in parallel. To do so, the system engineer has to virtually divide the whole application into three categories:

– Sequential code: control, strong data dependencies.
– Data parallelism: chunks of data that can be processed in parallel.
– Functional parallelism: tasks that can be executed in parallel.

Functional parallelism can be seen as pipelining whereas data parallelism corresponds to the Single Instruction Multiple Data (SIMD) approach.

The second objective is to generate a parallelized software by splitting the whole C application into pieces of C code.

Finally, a third and last DSE is realized at the platform level, called "System Level Exploration", that takes into account the outputs of the two previous explorations as well as other IP models like interconnects, memories and dynamic data memory management.

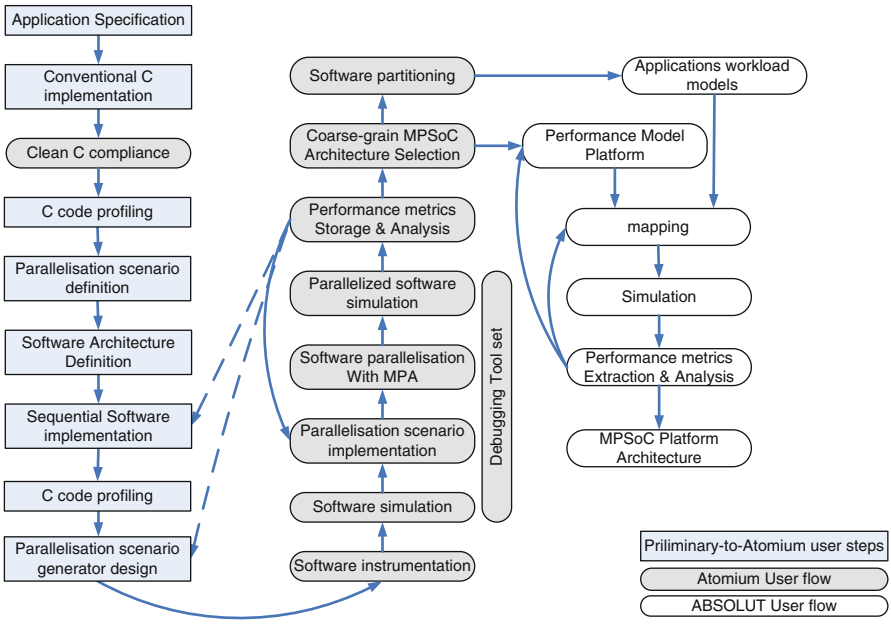

Fig. 7.4 User view of the MOSART MPSoC design flow

7.2.2 Design Tool Flow Overview

Customers of multi-core SoC platforms cannot afford to manually program batteries of processors. Consequently, providers of high performance embedded solution work on smart software development framework allowing to easily spread applications on their products. The programming environment is planned to ease the *end user* in designing the software and balancing the working load over all the computing units.

The MOSART Electronic System Level (ESL) design approach – especially the steps entitled *Parallelisation* and *Exploration* in Fig. 7.3 – is expected to prepare a part of the valuable programming solution to the *end user*.

Are considered as *end users*, designers that need to run an application on an MPSoC device: a system engineer for DSE or a software engineer for programming. Likewise at this stage, the targeted device can still be virtual or a prototype hardware board.

Figure 7.4 describes in details the way of deploying and executing an application on MPSoCs as Thales has experimented it with the MOSART design methodology and tools. The design flow has been built around Atomium and ABSOLUT tools provided respectively by IMEC and VTT.

This chart is divided into three main parts:

- The part of the chart made up of rectangular boxes collects all the mandatory steps which are required and recommended before achieving code partitioning with the Atomium tool suite.
- The dark round boxes are actions to be performed according to the Atomium tools usage. The objective is to explore the design space of parallelized scenarios of the applications in order to select the most relevant parallelized code in terms of performance.
- Finally, ABSOLUT (round white boxes) is used for identifying the best platform architecture based on the previously selected parallelized C code.

The next two sections describe, in details, the MOSART design flow from a user viewpoint at code partitioning and platform architecture exploration levels, respectively. Likewise, benefits and future enhancement techniques are also reported based on Thales experiments.

7.3 Coarse Grain Parallelism Extraction

This chapter focuses on user activities required to express and extract parallelism from the Sensing algorithm (rectangular and dark round boxes). The objective is to check how deep the application can be parallelized. Two results come out of the parallelism extraction: first, it provides to the user an idea of the (homogeneous) platform complexity in term of number of uniform processors required. Then, the MPSoC Parallelisation Assistant (MPA) tool provides the C code of the application in a parallelized C format.

7.3.1 Towards Parallelism

As highlighted by Fig. 7.4, preliminary steps (rectangular boxes) are numerous and required time. However, these steps are fundamental in the sense that it allows the user to reduce drastically time spent using the Atomium tools.

The starting points of the design flow consist in the specification of the application – the sensing algorithm in our case – and its implementation in a standard C and sequential code.

At this step, the most efficient approach is to take into account Atomium restriction and guideline rules, called *Clean C*.

The code partitioning work effectively starts at the fourth step, entitled *C code profiling*. The idea is to use a standalone tool able to report timing for all the functions called by the program. Thales used the well-known *gprof* utility that helps to identify time consuming functions used in the *sensing* application. At this step, the user knows where to focus his effort and what functions have to be considered as potential candidates for being executing in parallel over several processor units.

Table 7.1 Sensing execution time ratio per function

	Macro functions	Global execution time (%)	Micro functions	Local execution time (%)
Sensing use case	Glue	0.7	–	–
	Step 1	5.6	–	–
	Step 2	93.7	Transposition	47.2
			Quantification	7.1
			NB Filtering 1	34
			Signal Centering	0
			NB Filtering 2	5.4

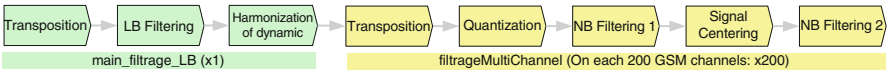

Fig. 7.5 MPA-based software architecture of the sensing algorithm

As shown in Table 7.1, it came out that most of the execution time (93.7%) of the sensing algorithm was concentrated in a part of the C code that will be renamed *step 2* as explained below.

Furthermore, software architecture guidelines have been determined – through usage of the IMEC's tools – in order to have a full benefit of Atomium utilities in terms of parallelisation efficiency.

As a consequence, the *sensing* algorithm software architecture has been designed according to Fig. 7.5.

Step 1 is performed via 3 functions labelled as main_filtrage_LB in the picture whereas step 2 is realized with the 5 remaining functions labelled as filtrageMultiChannel (more detail on the application in Sect. 7.1.3). The key idea is to divide the algorithm into functions that may be grouped for parallel execution purpose.

The software architecture must be designed by taking into account MPA features and the way parallel computing can be defined within Atomium. Indeed, the software architecture must allow the user to define any parallel code scenario that seems relevant to him.

From that paradigm, many potential parallel software architectures of the *sensing* were derived from the preliminary architecture shown in Fig. 7.5. Combined with analysis of the initial C code profiling reported in Table 7.1, effort for extracting parallelism was concentrated on *step 2* which roughly consists in extracting 200 GSM channels (200 kHz wide each) from a 40 MHz wide band radio signal.

Resulting parallelisation scenarios are reported in Fig. 7.6 in which step 1 is kept unchanged because it is not time consuming (only 5.6% of the whole execution time) whereas step 2 is duplicated up to 200 times. The adopted principle is to perform channel extraction in parallel – i.e. 200 *channelisations* at the same time – due to the fact that the same functionality is achieved 200 times on unchanged input data. In other words, this approach is similar to loop unrolling techniques applied by compilers and where no data dependency exists between iterations: it is referred as *data parallelism*. However, not all combinations are possible because the algorithm

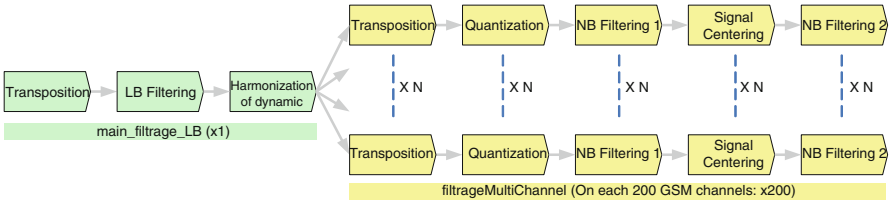

Fig. 7.6 Sensing parallelisation scenarios

has to deal with 200 channel extractions only. So, ten configurations have been considered: the five sequentially executed base functions are referred as a macro function that is duplicated N times on separate processors and executed $200/N$ times (where N is equal to 1, 2, 4, 8, 10, 20, 25, 50, 100 or 200).

Thus, the sequential *sensing* algorithm is split in N parts where N represents also the number of threads that will run simultaneously on the under construct platform.

The MPA philosophy is quite simple in the sense that each piece of C code identified to be a thread/kernel is annotated through a non-intrusive method by the means of C labels. Hence, once the parallelism scenarios are defined, associated C labels must be thought to be integrated in the implementation of the sequential C code. Then, each label is mapped to a thread within a MPA configuration file. In our case, the mapping is an off-putting task because of the large number of potential threads and scenarios.

As a consequence, the regular architecture of the parallelised software allows to develop a C program utility able to automatically generate the MPA configuration file for a chosen parallelisation scenario, as mention in the box entitled *Parallelisation scenario generator design* of Fig. 7.4.

All the tasks detailed in this paragraph prepare the *user* to the next step that is the use of Atomium tools for effectively achieving the parallelisation of the sensing C code.

7.3.2 *Sensing Application Parallelisation*

At this step, the user is ready to get inside the Atomium design tool flow[1] in order to parallelise its application.

Due to basic assumptions used by the Atomium tool suite like the fact that a unique, shared and single clock-cycle memory is considered, the performance gain reported for any parallelisation scenario produces a coarse grain parallelisation. Furthermore, memory congestion issues are not taken into account and threads are performed on identical CPU cores.

[1]The tools involved are provided by IMEC and are part of version 4.0.5 of the Atomium tool suite.

Table 7.2 Atomium clean C rules applied

Recommendation type	Rule number	Original Status	Current Status
Restriction	2	☑	☑
Restriction	4	☑	☑
Restriction	13	☑	☑
Restriction	14	☑	☑
Restriction	17	✖	☑
Restriction	21	✖	☑
Restriction	23	☑	☑
Restriction	24	☑	☑
Not reported rule	28	✖	☑

As a preamble, few words must be dedicated to C code conformance. Indeed the Atomium set of rules, named *Clean C for MPSoC* for C coding style, are divided into two main categories: *Restriction* rules that must be fulfilled and *Guidelines* that are strongly recommended.

Table 7.2 reports rules that Thales has dealt with in order to be compliant with Atomium[2]. *Goto* and *return* C procedure caused dysfunction during simulation of the instrumented C code (Restriction 17). Likewise, uses of pointers were removed to fit with *restriction 21*. Finally, function calls inside condition statement induced a failure during the parallelisation process (added rule 28).

The Atomium flow is made up of two main concepts that are, on one hand, sequential C code instrumentation for thread synchronisation and performance feature reporting, and on the other hand, concurrent task extraction for parallelisation of the C code achieved by MPA.

Indeed, instrumentation and simulation of the sequential code allow to get scheduling and timing information that will be used during the simulation of the parallelised C code to verify potential gain.

The developed C utility is applied to layout a chosen parallelisation scenario. Then, MPA is used to parallelise the application and inserting appropriate mechanism like FIFO for data dependency handling between threads.

Obviously, the next step consists in executing the partitioned software for functional checking and performance gain analysis as depicted in Table 7.3.

Furthermore, the *Performance Storage & Analysis* step allows validating whether the applied parallelisation scenarios were relevant. Even if extraction of concurrency was limited to 50 threads instead of 200 items as expected due to memory leak issues during simulation of the sequential C code, it came out that the execution time gain, in terms of clock cycles, was proportional to the number of threads in charge of extracting channels from a wide band signal, as illustrated in Table 7.3.

The 1:1 ratio between the number of cores and the performance gain is due to the fact that no data are exchanged between threads during the channelisation step and

[2]Thales used version 0.2 of the *Clean C for MPSoC* document for C code conformance checking.

Table 7.3 Estimated execution time gain vs. number of cores

Number of cores	Execution time (Million of cc)	Gain
1	17,000	–
2	8,500	2
4	4,250	4
8	2,125	8
10	1,700	10
20	850	20
25	680	25
50	340	50
100	–	–
200	–	–

then, the lack of data dependency nullifies the communication and synchronisation timing overhead. Indeed, the Atomium *Performance Analysis* utilities show that no FIFO is inserted between threads.

Obviously, a platform made up of 50 processors will not reach a gain equal to 50 because Atomium does not take into account memory congestion. So, the gain cannot be linear but asymptotic.

Atomium helps the *user* to generate a parallelised version of its application under coarse grain estimation of the performance. As a consequence, the design flow requires an additional tool able to capitalise the given C code partitioning under more realistic and accurate assumption like the memory organisation, interconnect Quality-of-Service (QoS) and CPU architectures.

In that purpose, the ABSOLUT tool has been experimented to provide more accurate performance results as well as power consumption and hardware resource use estimation (more details in Sect. 7.5).

7.3.3 Parallelised Software Debugging

Due to the number of cores in MPSoC systems, debugging is even more important for those architectures and requires additional debugging features. Indeed, data communications and synchronisations between processors rely on communication protocols (e.g., message passing) and the way they are implemented (e.g., FIFO).

Likewise, observing memory usage and behavior at runtime is critical and mandatory for MPSoC debugging.

Atomium debugging capability underlies on the openGL GNU GDB debugger. Furthermore, well-known tools like *Valgrind* can also be used to identify memory management issues. Those tools can be used at every compiling or simulation steps reported in Fig. 7.4 and have been experimented to fix bugs.

Table 7.4 Parallelized C code complexity

Sensing test case	MPSoC reference	MPA
Sequential C code	2146 LoC	
Design approach	Customised platform	Function call tool generation
Parallelised C code[a]	+37%	+61%

[a]Figures obtained for a 10 processor-based platform

7.3.4 Parallelised Software Complexity

Parallelisation of the initial C code leads to an increase of its complexity. As shown by Table 7.4, the C code parallelized with MPA is complex and contains much more lines of code than a tranditional approach (hand-based). However, an MPA advantage is to generate automatically and faster the parallelized C code. Furthermore, the partitioning is very flexible in the sense that the definition of parallelized scenarios is defined through labels. As a consequence, another benefit of MPA is to keep the integrity of the initial and sequential source code.

7.3.5 Productivity

An identified added value of MPA compared to traditional manual approach for code parallelisation resides in the productivity factor gain brought by the tool, which is estimated, to be equal to 8 in that experiment.

Another important metric deals with the size of the generated parallelised C code. In order to limit the memory instruction size and the associated debugging time, the code must be as small as possible. For a 50-core platform, the parallelised code overhead in terms of lines of C code is about 4 times longer than its sequential initial version counterpart.

7.4 Fine Grain Platform Architecture Definition

This section describes the methodology that is applied for platform modelling and architecture exploration.

Once a satisfying parallelisation of the application is found with the Atomium flow, the parallelised code is translated into a set workload models (one per thread) that represent the load of the application in terms of computation and communication activities. Then, these statistical workloads are mapped on to a virtual simulation platform model developed in SystemC and called ABSOLUT (see Fig. 7.4). ABSOLUT represents the computation and communications capacities of the platform.

The virtual simulation platform is developed upon the following information:

- Number of concurrent processors identified by MPA.
- Type of each processor
- Memory organisation
- Type of memories
- Type and definition of the communication channels

7.4.1 Underlying MPSoC Architecture

The baseline platform that has been considered in this experiment is a Shared Memory multiProcessor (SMP) many-core organisation where each node is made up of a Very Large Instruction World (VLIW) ASIP, a private local memory, a public and shared local memory and a Data Management Engine[3] (DME) in charge of data coherency and transfer through the interconnect.

Consequently, the memory is distributed across all the nodes of the MPSoC. On the other hand, the considered ASIP is a 4-issue in-order processor with a single load/store unit.

On the interconnect side, a low power 2D-mesh Network-on-Chip (NoC) has been taken into account with synchronous communication mechanisms as detailed in Chap. 3.1.1 McNoC Overview.

Finally, all nodes were connected to the NoC via Open Core Protocol (OCP) interfaces.

7.4.2 Performance Model Metrics

In addition to code partitioning achieved by MPA, Atomium is also in charge of checking the functionality of the parallelized code. On the other hand, ABSOLUT is a performance model platform dedicated to refine the MPA coarse-grain speedup and to report power consumption and energy figures.

To do so, metrics of Table 7.5 have been introduced in ABSOLUT to deals with power consumption. Those metrics have been extracted from a TSMC 90nm technology (HP library).

Likewise, memory requirements in terms of storage capacity for the sensing application have been extracted by studying the sensing algorithm and are reported in Table 7.6, as well as their respective power consumption based on Table 7.5. In terms of latency, data are located in private SRAM memories that can be accessed in a single clock cycle per its associated processing unit core.

[3]For more information on DME see Chap. 1.3 Data Management Engine (DME).

Table 7.5 Elementary power consumption of the MPSoC units

Components	Power consumption
ARM idle	6 mW
ARM active	0.133 mW/MHz *400 MHz + 5 mW = 58.2 mW
Cache idle	4 mW
Cache active	0.060 mW/MHz * *400 MHz + 4 mW = 28 mW
Memory idle	(1*10e–6 mW/bit − cell)*64K = 64 uW
Memory active	(1*10e–6 mW/Mt + 6*10e–7 mW/Mt)*64 = 102 uW
Network IF	0.06 mW/MHz*400 MHz = 24 mW
Router	0.037 mW/MHz*400 MHz = 14.8 mW

Table 7.6 Data memory size per node of the MPSoC

MPSoC configuration (# cores)	Data memory per core (KB)	Data memory consumption (uW)	
		Idle mode	Active mode
1	250	250	400
2	135	135	216
4	84	84	134.4
8	56	56	89.6
10	51	51	81.6
20	40	40	64
25	38	38	60.8
50	33	33	52.8
100	31	31	49.6
200	30	30	48

Table 7.7 MPSoC structure organisation

Number of master threads	Number of slave threads	Number of cores	2D mesh organisation	Number of unused cores
1	2	3	1×3	–
1	4	6	2×3	1
1	8	9	3×3	–
1	10	12	4×3	1
1	20	21	7×3	–
1	25	28	7×4	2
1	50	54	9×6	3
1	100	104	13×8	3
1	200	204	12×17	3

Finally, Table 7.7 describes the chosen MPSoC backbone that is a 2D mesh structure made up of a matrix of processors and a control processor in charge of executing a part of the algorithm (step1) and organizing communication between threads.

All the cores are VLIW ASIPs with an Instruction Per Cycle (IPC) equal to 0.5. The processors, DME, memories and NoC are running at a clock frequency of 400MHz[4].

7.4.3 Application Programming Interface

Once the hardware MPSoC architecture has been described, the platform is implemented, as depicted in Fig. 7.4 ("Performance Model Platform" bubble). Consequently, an abstraction layer of services – so called Run-time Library – provided by the platform and required by the sensing application must be defined and implemented. In our case, the default Application Programming Interface (API) available in ABSOLUT was sufficient to serve the application needs.

7.5 Performance Results for MPSoC Solutions

ABSOLUT provides several type of results of an application deployed on a MPSoC platform like the occupancy rate of the hardware resources, the number of memory accesses, the computing power, the execution time, the power consumption and energy.

For instance, ABSOLUT reports the activity per each processor of the platform. Activity is divided into three categories: idle, memory access and data processing mode. This experiment confirms our expectations in the sense that all processing units of the MPSoC have the same behavior in terms of activity, except the control dispatcher processor.

Indeed, Figs. 7.7 and 7.8 show an opposite activity rate between processors in charge of processing the step2 of the algorithm and the one dedicated to treat the step1 part. This is due to the fact that the control/master processor has to manage an increasing number of threads and to transfer data to more (slave) cores.

The execution load is equally spread over all the step2 processors. This behavior validates the source code partitioning achieved by MPA.

The global activity of each node decreases when the MPSoC complexity increases because they have less individual tasks to perform: all the cores equally share the load. Nevertheless, the total computing power increases as illustrated in Fig. 7.9 to reach a maximum of 4.2 GIPS for 28 cores and proves the benefit of the parallelisation.

[4]Both IPC and frequency have been obtained from the generation of a customized ASIP for the sensing application with the Processor Designer tool of Synopsys.

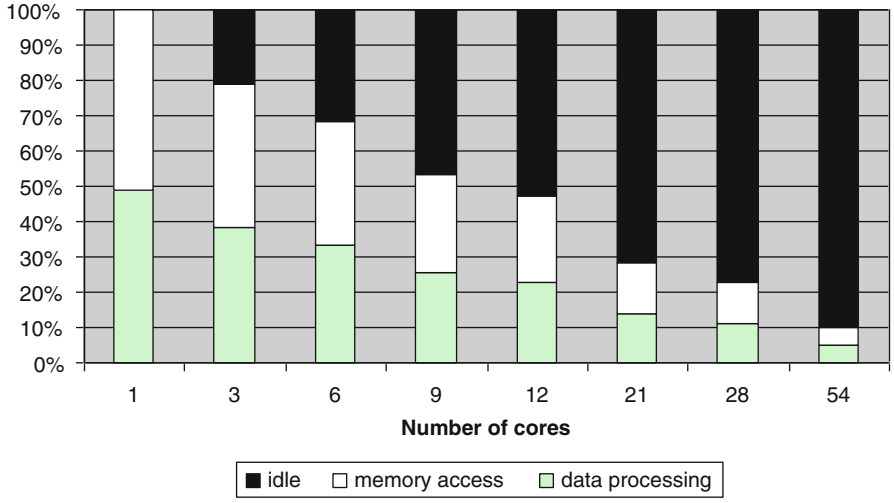

Fig. 7.7 Processing cores activity profiling per platform configuration

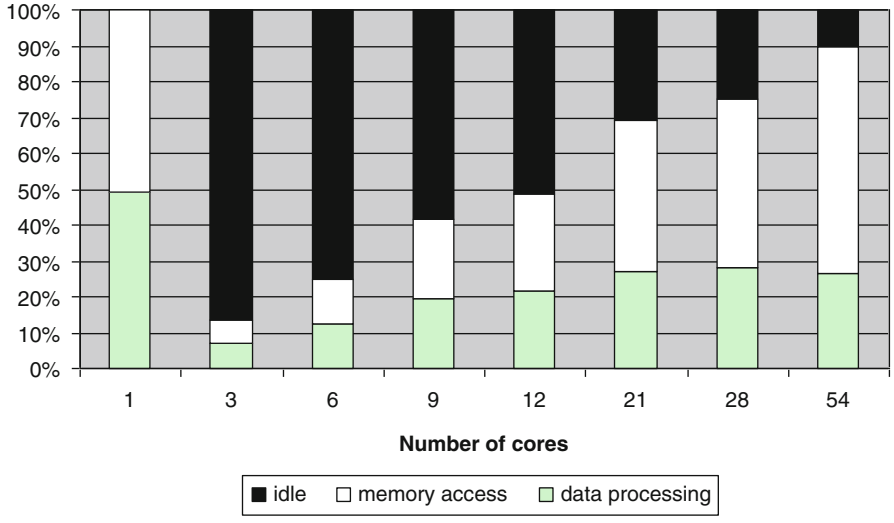

Fig. 7.8 Control core activity profiling per platform configuration

However, the computing power saturates as of 21 cores. This is a side effect of the parallelisation in the sense that a new critical execution time path appeared: indeed, this experiment focused on step2 of the sensing algorithm and step1 has not been parallelised. Consequently, time to execute the step1 part of the code limits improvement brought by the parallelisation of step2.

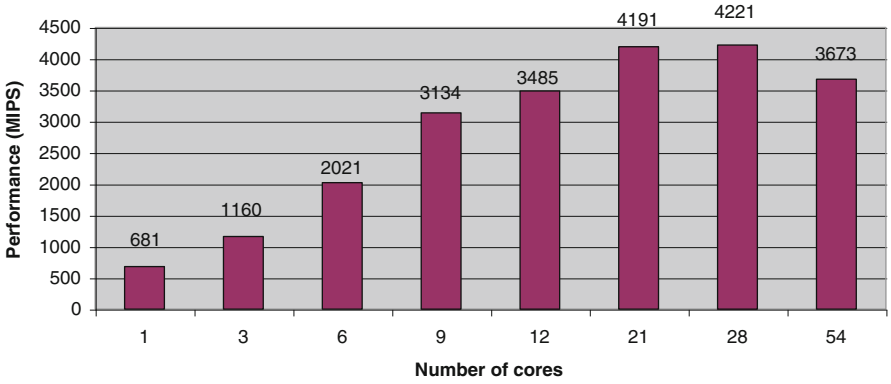

Fig. 7.9 Computing power profiling per MPSoC configuration

Furthermore, the computing power decreases for a 51 processors configuration: this is due to the data communication overhead between the master processor in charge of managing and distributing a common set of data to all other cores.

This phenomenon is highlighted by the Amdahl's low that claims that the maximum parallelising potential speedup of an application is modeled by (7.1).

$$speedup \leq \frac{1}{(1-P)+\frac{P}{N}} \tag{7.1}$$

where P represents the percentage of execution time the application can be parallelised and N is the number of time P part can be processed in parallel.

This speedup will not be reached since interconnect congestion, memory access time and latencies are not taken into account in this formula. Moreover, each additional processing unit will bring less computing power than the previous one and the speedup limit is equal to 1/(1-P).

This behavior is illustrated in Fig. 7.10 where the theoretical speedup saturates at 92.2% of execution time improvement compared to the sequential program: this limitation is reached as of 21 cores.

Indeed, in the sequential form of the application, step1 represented 5.6% of the total execution time, only. But as of 21 cores, the execution time between step1 and step2 are balanced and step1 corresponds to 45% of the execution time. Furthermore, the control processor in charge of step 1 has to deal with more and more threads and data to be duplicated to serve all others cores.

As a consequence, to go beyond 21 cores and increase the performance, step1 must also be taken into account in the parallelisation process.

Another aspect of the methodology and tools experiment was to determine the efficiency of the MPSoC solutions other than the execution time and complexity

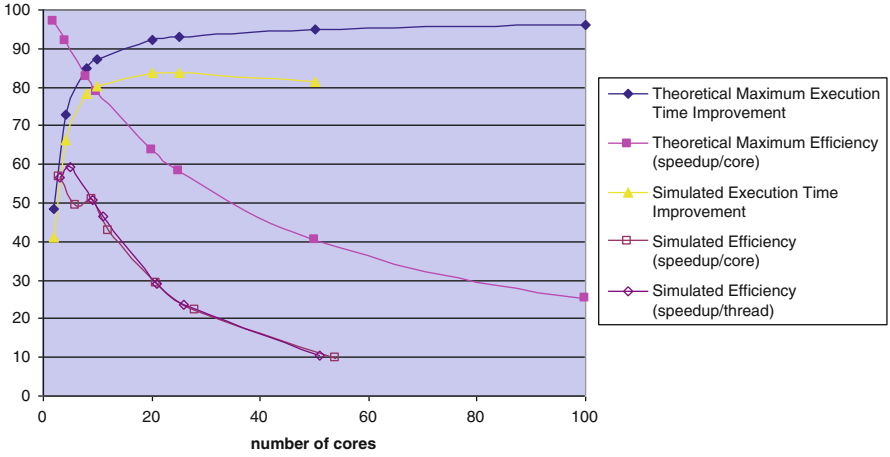

Fig. 7.10 Theoretical and simulated execution time speedup profiling

ratio shown in Fig. 7.10. Indeed, power consumption is often the strongest constraint of embedded systems. So, the idea is to use a metric for efficiency purpose that takes into account energy of CMOS integrated circuits.

Power consumption will not be considered for estimating efficiency since this parameter can be "improved" by reducing the clock frequency as shown in formula (7.2), what is not necessarily expected.

$$P = C^* V^{2*} f \tag{7.2}$$

where C represents the total input capacitance of the cells connected to a CMOS gate

V is the power supply
f is the operating frequency

Energy is not a valid alternative neither even if the previous clock frequency trick disappears in that case since the application will run longer. Indeed, (7.3) shows that the energy metric can be "enhanced" by reducing the power supply. However, doing so increases the period, τ, of the circuit as illustrated in formula (7.4) that will slow down the execution of the application.

$$E = C^* V^2 \tag{7.3}$$

$$\tau = K^* C^* \frac{V}{(V - V_t)^\alpha} \tag{7.4}$$

where τ represents the critical CMOS timing path

V is the power supply
V_t is the CMOS transistor voltage threshold

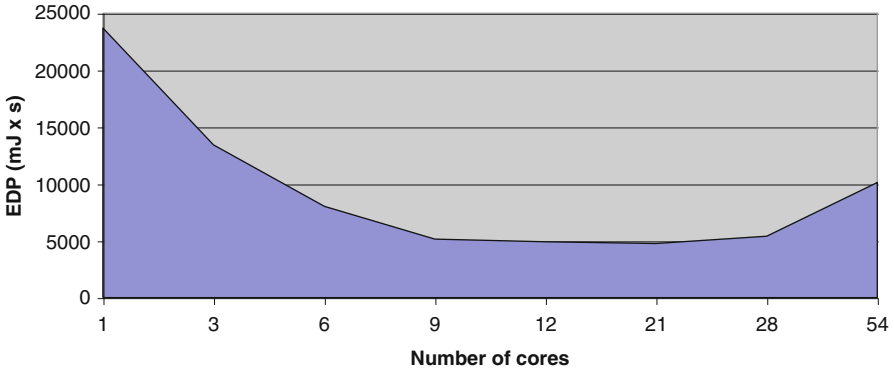

Fig. 7.11 Energy-delay product profiling per MPSoC configuration

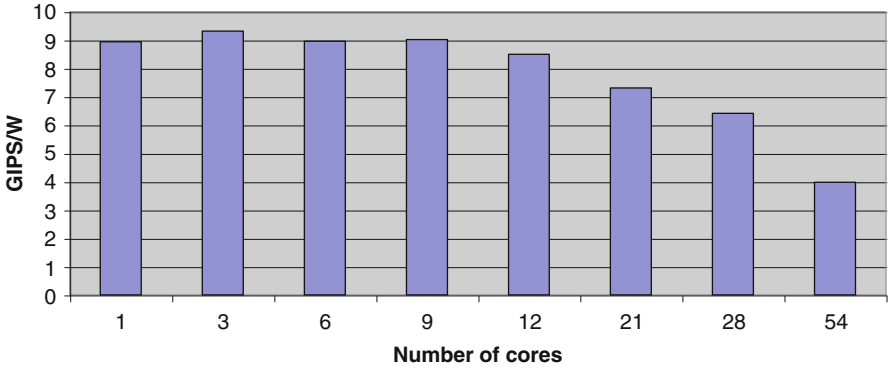

Fig. 7.12 MPSoC platforms efficiency based on computing power and power consumption

So, a relevant metric to compare energy for a same level of performance is the Energy-Delay Product (EDP). Figure 7.11 reports the EDP metric for all the considered MPSoC architectures and highlights that the configuration made up of 9 cores is the best among the 8 solutions since the smallest EDP value involved a minimum number of cores compared to the 12, 21 and 28 matrixes.

Even if the computer power increases with the platform complexity as illustrated in Fig. 7.9, the power consumption increases as well in such a way that the optimum EDP figure is obtained for a platform made up of 9 cores only.

Based on computing power and power consumption figures generated by simulating platform performance models, it came out that the maximum number of GIPS per Watt is equal to 9 and do not really scale with the MPSoC size as shown in Fig. 7.12. Indeed, the power consumption increases with the number of cores from 76 mW for a single core to 923 mW for 54 ASIPs, as reported in Fig. 7.13, whereas the computing power saturates around 4.2 GIPS and even decreases as previously explained.

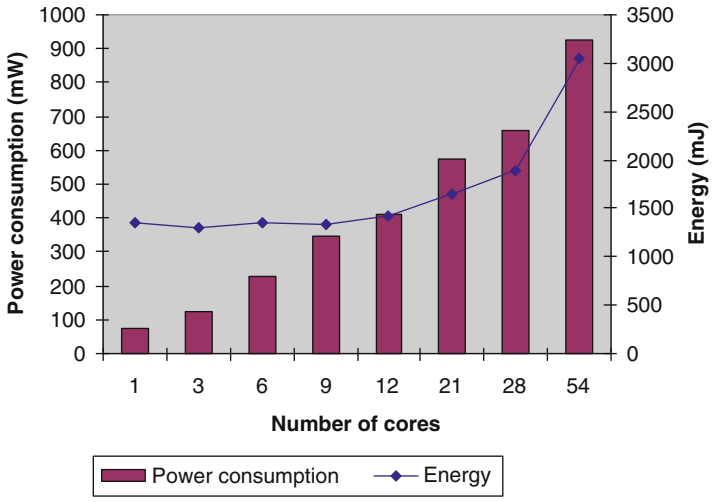

Fig. 7.13 Power consumption and energy profiling per MPSoC configuration

The GIPS/W figure is not higher than 9 for the following reasons:

- A 90nm technology was considered; GIPS/W will be enhanced by the use of other 65, 45, 32 or 22nm technologies.
- Clock and power supply domain management mechanisms were not considered and would reduce power consumption and hence increase the GIPS/W number. The total power consumption of the platforms has been overestimated because a conservative power consumption model of each processor was used.
- The many-core architecture does not take advantage of the NoC capabilities; the Distributed Shared Memory (DSM) organisation, associated with the sensing algorithm feature, does not solicit the high data throughput facilities of the NoC since most of data to be processed are local to each node and processing time is greater than time required to update local memories.
- The application does not allow having benefit of dynamic data management techniques dedicated to limit power consumption and reducing global memory access time.
- A part of the application that was voluntary not parallelised became a new critical path in the sense the parallelised code waits for data provided by the part of the application that is still sequential.
- A single and simple master processor handles synchronisation and data distribution with the others processing units. Time requires by this task increases with the MPSoC complexity.

The previously results relevancy depend on the ABSOLUT tool credibility. To be credible, the ABSOLUT outputs have to be realistic and accurate. That is the reason why results of a single-core ABSOLUT platform have been compared to the performance of a similar industrial VLIW synthesized processor that ran the

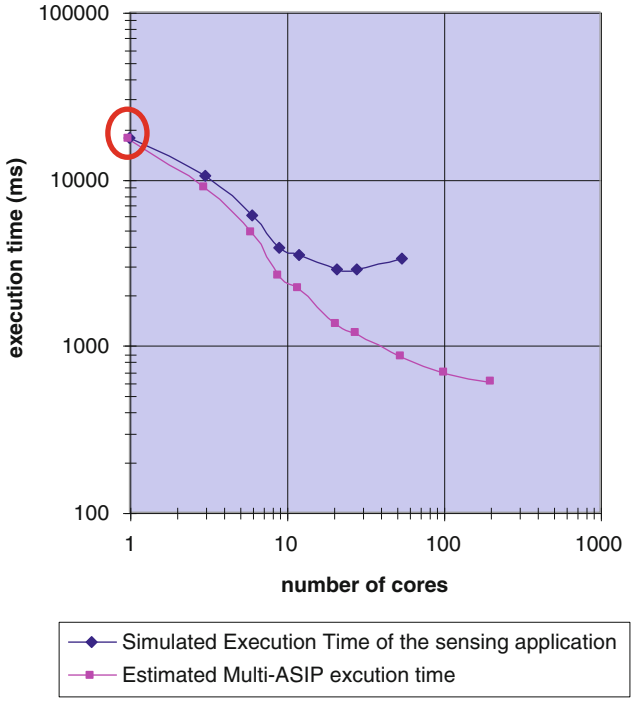

Fig. 7.14 Executed vs. Simulated sensing execution time

same application: a gap of 3.2% have been identified between those two approaches as shown in Fig. 7.14 (cercled points). The squared curve has been extrapolated from core 1 by using the Amdahl's law equation (7.1). The two curves mainly diverge because Amdahl's law does not take into account interconnect and memory latencies.

7.6 Conclusion

Power consumption and energy dissipation are ones of the main issues that embedded computing systems will have to tackle in a close future. In that context, many-core SoCs have appeared to be valuable solutions. A trend for such multi-application platforms is to consider heterogeneous systems made up of many, specific and simple processing units in order to limit the power consumption. The idea is to spread and balance the application operations over the platform resources in a way that hot spots and performance, best energy operating or quality of service will be leverage.

Through this experiment, a methodology and design tool flow for MPSoC implementation and code application partitioning have been evaluated by exercising future cognitive radio application processing. It has been possible to run a whole flow from the application specification to its deployment on a relevant many-core platform model whose architecture has been identified and challenged during the design flow.

The MOSART approach is based on tools that have been developed, validated and integrated in a tool suite in a way that they allow to express, extract and exploit parallelism from a sequential application coded in C language. The design flow allows executing a virtual performance platform model which the parallelised C code is mapped on it.

The methodology is made up of two main steps. The first one consists in parallelising a sequential application described in a high level language according to various scenarios defined by the user. The resulting threads communicate each other via FIFOs that are automatically generated and integrated in the code. The output is a set of parallelised codes that are then validated by simulation.

The second step of the methodology transforms the previously parallelised application into a set of workload models (one per processing unit). Then, these workload models are mapped to an accurate performance platform model that is created by the user through libraries and customised IPs.

The test case has been supported by two tools that are Atomium for code parallelisation purpose and ABSOLUT for mapping, fine grain performance modeling and profiling. These tools are distributed by IMEC and VTT, respectively.

Atomium and ABSOLUT are consistent from a processing unit occupancy rate point of view. Furthermore, the ABSOLUT performance platform model has been validated with a synthesized VLIW processor. The execution time of the sensing application running on a single ASIP core platform modelled with ABSOLUT has been measured to be more than 95% accurate compared to its execution on a similar synthesized VLIW processor: indeed, in this configuration, time required to perform the sensing algorithm on top of an ABSOLUT platform is 3.2% accurate.

Both tools were integrated in a tool flow in order to scan a large set of MPSoC architectures that can be compared regarding computing power, power consumption, energy, resource occupancy rate and memory access figures.

The main part of the sensing application has been split in 50 concurrent subfunctions but it appeared that the 25-core MPSoC platform provides the higher computing power (4.2 GIPS) and the shorter execution time. This due to the fact that a part of the application was not taken into account in the parallelisation process and became the new critical path after that.

Based on TSMC 90nm power consumption figures, the platform made up of 9 cores is the most efficient in terms of computing power regarding the lower power consumption profile.

Of course, the achieved performance will be improved when moving to deep submicron, and with the use of new power management of the cores and power supply techniques.

In conclusion and in the scope of many-core SoCs, the following challenges have to be mastered in order to be able to exploit the benefits of such architectures:

- Flexible application partitioning from sequential codes.
- Predictability and determinism of programs running on top of many-core structures.
- Application implementation at a high level of abstraction.
- Early evaluation of performance and reliability of programs running on a many-core architecture.
- Middleware and hardware mechanisms to serve, at run-time, several constraints like computing power, low energy operation, thermal dissipation and reliability or quality of service.
- Optimized compilers and operating environments that take into account the MPSoC micro-architecture to tailor the performance, thermal dissipation or low energy operations.
- Adequate programming model and debugging support.

Chapter 8
Application of the MOSART Flow on the WiMAX (802.16e) PHY Layer

Frank Ieromnimon, Dimitrios Kritharidis, and Nikolaos S. Voros

Abstract Current and emerging telecommunication standards require the fast and efficient design of large hardware/software systems that need to bring together diverse engineering skills. MOSART offers a solution to the problem of fast design of large systems by converting the current design cycle into the transformation of an executable reference specification to software optimized for execution on tailored multiprocessor platforms. ICOM has tested this approach using as test-bed a part of the PHY layer for the 802.16e (WiMAX) standard; a strategic roadmap system for INTRACOM. Use method and experimental results are reported for this exercise.

8.1 Introduction

This chapter outlines the implementation of a complex case study borrowed from telecommunication domain using the MOSART approach. Scalable multiprocessors such as MOSART are envisioned as the answer to the ever-increasing computation requirements of modern applications. These same applications also pose severe constraints regarding power-consumption on one hand and speed of development on the other. MOSART aims for both high power efficiency and ease/speed of development, through the use of an innovative Multi-core Multi-ASIP architecture and advanced development tools.

The application described in the next sections is an implementation on the MOSART platform of selected parts of the PHY layer of an experimental prototype

F. Ieromnimon (✉) • D. Kritharidis
INTRACOM S.A.Telecom Solutions, Peania, Greece
e-mail: fier@intracom.gr

N.S. Voros
Technological Educational Institute of Mesolonghi, Department of Telecommunication Systems & Networks (consultant to Intracom Telecom Solutions S.A), Greece

D. Soudris and A. Jantsch (eds.), *Scalable Multi-core Architectures: Design Methodologies and Tools*, DOI 10.1007/978-1-4419-6778-7_8,

of an IEEE 802.16e based broadband wireless system. The 802.16e standard has been defined to support broadband mobile connectivity in urban environments, through the use of scalable OFDMA and adaptive modulation ranging from BPSK to QAM64, spatial multiplexing techniques (MIMO antennae), power-saving modes, etc. The standard places heavier processing requirements than the earlier fixed WIMAX standard of 802.16d, coupled with the ever-present need for low-power mobile terminals. The use of NOSTRUM – a flexible multi-core platform – for this application will go well with the need to adapt to changing standard specifications. The chosen application subset has been coded in 'C', and gone through the steps of parallelisation by means of the IMEC Atomium/MPA tool and simulation/evaluation by means of the ABSOLUT environment by VTT.

8.2 Current State of the Art in Complex System Design

The current method for designing large systems, encompassing significant amounts of SW and HW, can be roughly depicted with the diagram of Fig. 8.1. Usually, one starts from textual specifications, either formalised in international standards or informal. There may also exist partly or wholly executable specifications, in the form of SDL or more recently UML code. Legacy 'C' code can also exist and may have to be incorporated in the application. Typically, no single such specification system will be capable of defining the entire application.

Translation into a cohesive form of executable specification is a largely manual process. It is common to describe a large part of the specification in MATLAB

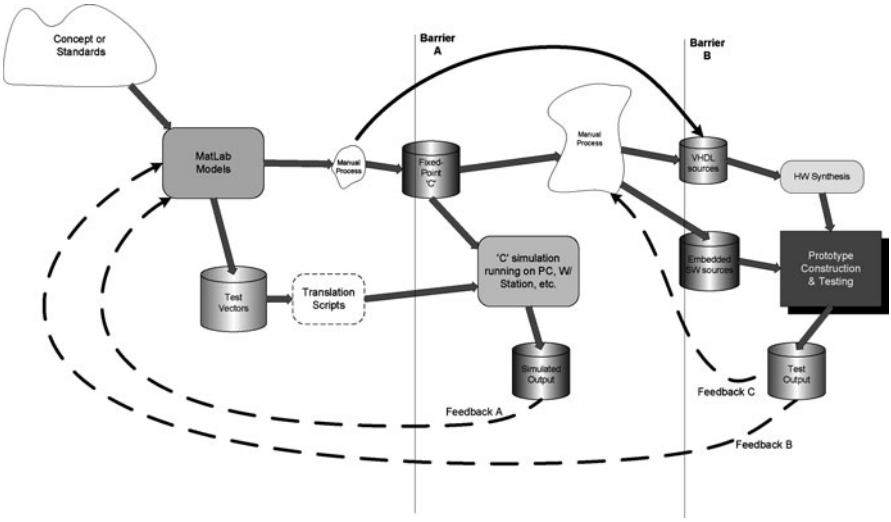

Fig. 8.1 Pictorial representation of a typical design flow for a large telecom & multimedia system, including embedded SW and custom HW

code, to allow early stage validation and refinement of the application. This stage is often employed for the generation of reference test-vectors, to be used in subsequent simulation and validation steps.

Following the specification (and possible tool-assisted validation) stage, the original specification can be rendered in 'C' code, which can be executed on a generic computing platform (PC or workstation) for validation purposes. Test vectors generated earlier by MATLAB or other tools may also be used as input stimuli or output comparison data. If the simulation step indicates discrepancies with the expected behaviour, the designer(s) go back to the 'C' rendering stage, to modify the code in order to bring the simulated behaviour in line with informal specifications or with the equivalent simulation output from, e.g. MATLAB. This regression stage is illustrated Fig. 8.1 as "feedback A". Since the code modification process is done pretty much in the same manner as the first rendering stage, the number of necessary iterations varies with design team experience.

Once the executable specification has stabilised, the design team moves on to the design of custom hardware that embodies the specification's functionality either wholly or partly, with the rest of the intended functionality being supplied by software running on embedded processors. Hardware specification languages such as Verilog and VHDL are in widespread use and the custom hardware design is again a manual process, because tools that can automatically generate high-quality synthesizable RTL code starting from 'C' are still in their infancy.

Writing the SW to run on embedded processors is again a manual process of identifying parts of the base 'C' source that should execute on the embedded processor(s) and modifying them in order to be incorporated in the RTOS that typically provides an abstract interface between the processor's user space and the custom hardware it must interface to. As such, this procedure is even more difficult to automate in any way than custom HW design.

During this phase, there exist again feedback paths leading to earlier development phase. One of them ("feedback C") is driven by the simulation of the RTL models of the developed HW and drives manual modification of the RTL source description. The other path ("feedback B") only becomes effective after building of a physical prototype, which includes both the synthesized HW and the SW developed for the embedded processor(s). It is then that broader, system-level functionality issues can be identified and may lead all the way back to the original models.

8.3 The MOSART Proposal

MOSART proposes the use of a multi-core NoC platform as the vehicle for implementation of complex systems. The platform is programmable, without the need to develop custom hardware alongside the processing cores. Thus, while an executable specification for the application is highly desirable from the developer's point of view in any case, a complete specification written in 'C' is all that

is required as the reference point for system development within the MOSART framework. System development thus becomes more SW-oriented and the vehicle for this SW approach is a multiple-processor platform.

Multicore systems are getting increasing attention recently, because people are realizing that the two-decades old warning coming from computer science and engineering departments in universities around the world is now coming true: single processors are running out of steam and large ASIC design is not getting any easier. One must note that multi-processor systems of various architectures are not novelties. Various memory and interconnect topologies have been tried in various experimental machines in the past, such as shared/distributed memory, bus-based, star or grid interconnects, etc. Of these topologies, the idea of distributed memory and a rectangular grid interconnect lends itself well to the planar lithography techniques universally employed for chip manufacture. The implementation platform for MOSART is employing just this architecture with some added features, that enhance its capabilities: Point-to-point packet-based inter-node communication, a distributed DMA capability to offload processors from simple data-movement between nodes and per-node power-control features for power optimization. And of course, the main advantage of a grid-based multi-core platform is always there besides these enhancements: intrinsic expandability that is not limited by bus capacity or memory devices throughput and communication latency that grows at worst linearly with array size. A grid-based architecture is ideally set to exploit applications rich in data-parallelism or streaming-based.

This changes a bit the design paradigm. It is now more important to produce an executable specification of the entire application (e.g. in 'C') before moving on with application mapping on the NoC platform. Therefore, the "Barrier A" seen in Fig. 8.2 becomes costlier to cross, as the designer needs to stay within the 'C'-source generation phase for as long as it takes to produce a complete description of the target system, because that will essentially be the SW running on the multicore platform. That description should be the golden reference for the MOSART environment (taking into account concurrency-modelling limitations and real-time issues).

The NoC takes care of inter-processor communication in an efficient and user-transparent manner. The processing nodes may be standard cores of fixed architecture but, more importantly, they can also be user-defined ASIPS. This second choice open the possibility of (and creates the need for) an optimization stage, that attempts to optimally match ASIP capabilities to the target application. Effectively, the development effort becomes the interplay between porting of the golden reference 'C' source and producing one or more ASIPs optimized to run the (modified for concurrency) 'C' source. The feedback paths implied by this process are indicated as "opt A" and "opt B", alongside the already discussed "feedback B", in Fig. 8.2 that illustrates a user's view of the MOSART flow. Note that the indicated feedbacks are within the MOSART framework. This implies that data format and other interface issues are taken care-of. Therefore, dwell-time within these internal development cycles is very short, compared with the conventional approach, where ad-hoc translation scripts and protocols are created as needed, with the associated expense of time, possibilities for misinterpretation, or even outright bugs.

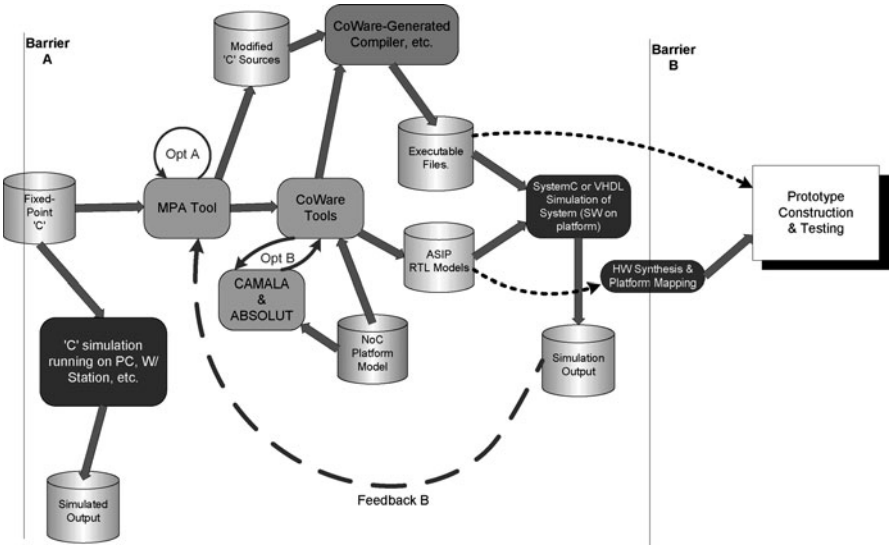

Fig. 8.2 Illustration of the development flow within the MOSART framework. "Barrier A" separates the MOSART domain from the ad-hoc executable-specification generation phase of the previous phase

Ultimately, refinement of the interactions between components of the MOSART toolset should result in a production system with the characteristics of current FPGA systems, where the user is (usually!) not concerned with issues of correctness of results or tool robustness, and (s)he is fairly confident that the physical platform loaded with the targeted application bit-streams will behave just like the simulated system. Of course, the pitfalls associated with passing from a simulation of typically limited scope to a physical system with a thousand-fold or million-fold increase in speed, still remain to be addressed. However, the taxing problem of mapping a conventional sequential program to a multi-core platform will be solved, relieving the designers from the onus of coming up with a silicon and power-efficient platform, on top of the main effort associated with building their application.

One important observation can be made on the working model presented so far: The original ("golden") executable specification is coded in sequential 'C', but the target of the MOSART flow is parallel programs that must run concurrently (and correctly!) on multiple, possibly differing processor cores. Thus, the original sequential program must be correctly turned into a number of concurrent programs. This is a very hard task. Indeed, it is quite hard to produce correct concurrent programs starting from scratch and having the underlying multi-core platform stable and fully documented. Taking a sequential program, identifying available parallelism and exploiting it through manual program transformations is harder still.

That is the reason that parallel programming is making such slow inroads into the realm of system programmers, despite the acknowledgment of the advantages of

multi-core platforms. A large part of the problem stems from the fact that there can be no automatic method for identifying parallelism available in a program, thus human intervention is usually required, while at the same time is not enough: Humans are very good at identifying patterns in programs, however they are not so good at elaborating the consequences of the usual flow-control mechanisms found inside programs, such as conditional statements, data-dependencies, exceptions, etc.

The MOSART approach for this problem is machine-assisted parallelization that is user-driven. The user inserts in the original program labels, using standard 'C' notation, that leaves the functionality of the program intact. In addition, the user supplies a small script, suggesting how the labelled program segments might be parallelized. From then on, it is left to the MOSART tool flow to employ the user input, for the correct transformation of the original program into multithreaded version that exploits available parallelism.

8.4 Intracom Telecom Demonstrator

Selected parts of the PHY layer for an experimental IEEE 802.16e prototype have been chosen as the demonstration vehicle for MOSART. The chosen parts, although not accurately reflecting the standard's PHY layer, they include computationally heavy components, which exhibit substantial amounts of parallelism.

The drawing of Fig. 8.3 illustrates the component train for the application, linked with shared memories which are organized as ping-pong buffers, to hold processed data in transit between the chain's processing stages. The blocks identified as "Tx chip sequencer" and 'Rx byte assembler" are simple FSMs that act as interfaces to the byte-serial data stream leaving the first Scrambler block and entering the second Scrambler (doing the unscrambling). The next section describes the designed and implemented components.

8.4.1 Architecture of the Demonstrator

8.4.1.1 The TX Path

The example application's Tx consists of a single-antenna "transmitter path", made-up from a bit-pattern generator, a QAM encoder capable of modulation schemes up to QAM64 and an inverse-FFT of 512 points plus cyclic-prefix insertion. A 16-bit scrambling polynomial of the form $x^{14} + x^{13} + 1$ is also employed as a scrambler of the incoming bit-stream, to reduce the power content of any individual spectrum component that will be present on the iFFT's output. The unit operates in bit-serial format, but it handshakes with the following component on a byte-by-byte basis.

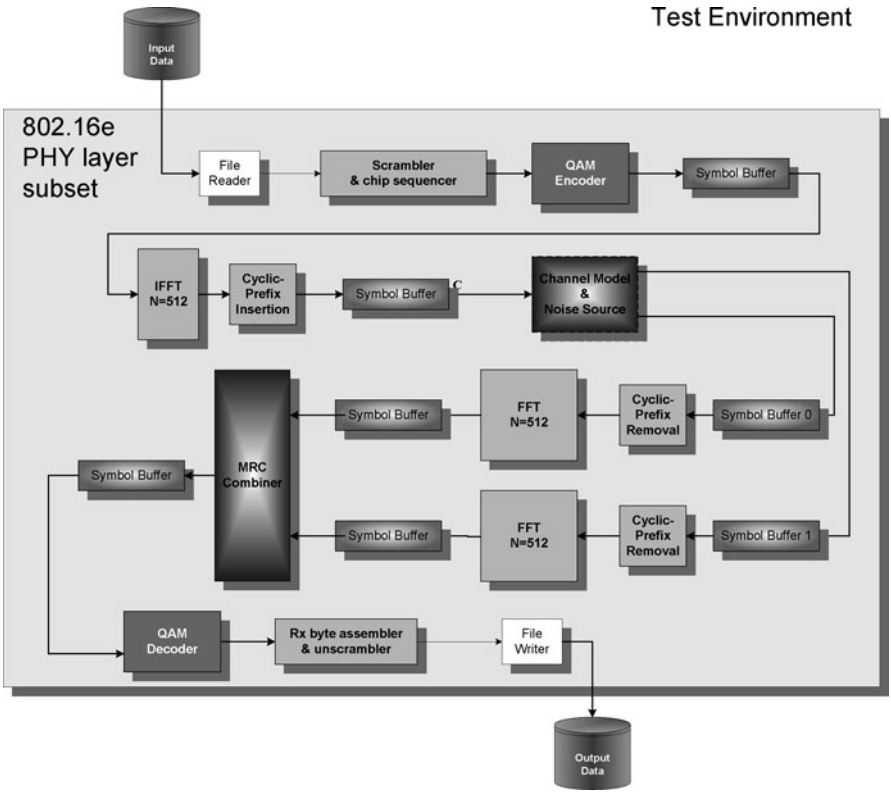

Fig. 8.3 Application for evaluation of the MOSART platform

8.4.1.2 The Multipath Propagation Channel Model

The code for this block models a channel with two dispersive, AGWN propagation paths. The coefficients characterizing path loss over frequency bands are developed offline and jointly with the corresponding data for the MRC combiner (see next section).

Figure 8.4 illustrates the mechanism for generation of the path-loss/phase-shift coefficients, while also generating (under that same operational scenario) the compensation vectors and the SNR-derived weight-factors for the MRC/equalizer block discussed in the next section. Figure 8.5 illustrates the outline of the channel model. As can be seen, the individual channel delay/phase shift is generated by superposition of two individual phase-shifted components. AWGN noise is also added to each distorted symbol stream, with a predetermined SNR.

Compensation vectors CV1,2 and weight-factors W1,2 are calculated as illustrated in Fig. 8.6 by the program of Fig. 8.4.

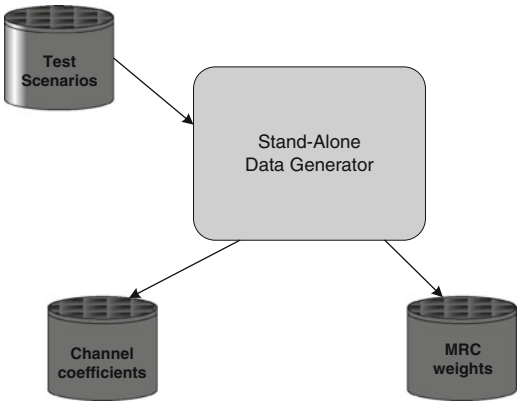

Fig. 8.4 Generation of path-loss coefficients and their compensating coefficients for the MRC/equalizer block

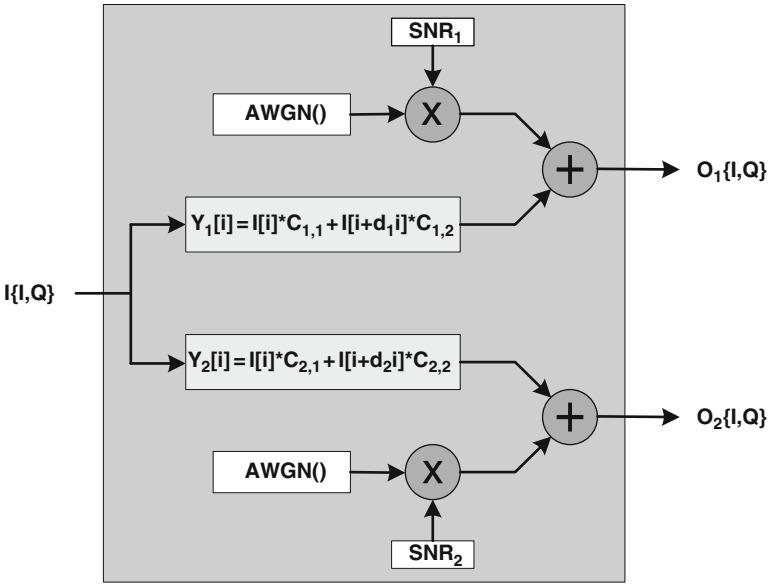

Fig. 8.5 Structure of the channel model, featuring two propagation paths with distortion and AWGN addition

$CV_j[i] = Q[i]*Q_j[i]'/|Q_j[i]'|^2$, for all i in {0,…,511}, j in {1,2}

$W_1 = 1 - 1/(1 + P_2/P_1)$
$W_2 = 1/(1 + P_2/P_1)$

P_j = Sum of $|Q_j[i]' - Q[i]|2/N$ for all i in {0, …, 511}, j in {1,2}

Fig. 8.6 Method for generation of the MRC/equalizer block's compensation vectors and weight factors

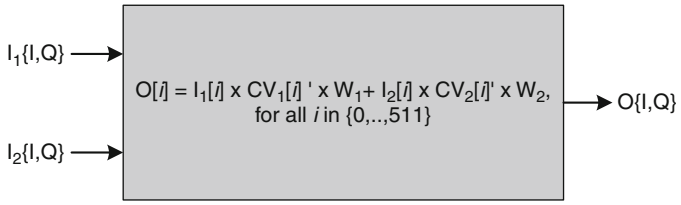

Fig. 8.7 Conceptual view of the MRC/equalizer block

Q stands for the symbol vector produced by the i_FFT of the Tx-path, while Qj' stands for the distorted & noise-loaded output symbol produced by the channel model's path j. The compensation vectors and weight factors are generated by the off-line program illustrated in Fig. 8.4, since we will not be incorporating any dynamic correction mechanism for updating the weight tables. The program re-uses the channel model in order to produce distorted outputs of pre-selected input symbols. The vector generator can also produce time-varying channel coefficients and compensation factors, thus being able to simulate a variety of operating conditions for the targeted physical layer chain.

8.4.1.3 The Rx-Path

The receiver path subset is formed around two FFTs of 512 points, (we assume a twin receiving antenna architecture), preceded by their respective Cyclic-Prefix Removal units, a Maximum Ratio Combiner plus equalizer, that merges the outputs of the two FFTs based on pre-calculated SNR figures for the channel model's propagation paths, followed by a QAM Decoder, also capable of detecting modulation schemes up to QAM64 and finally a bit descrambler. The decoder's output drives a bit-pattern checker, which examines the received bit-stream against the one processed by the TX path. Figure 8.7 illustrates in some detail the algorithmic design of the MRC block.

8.4.2 Performance Requirements for the Demonstrator

We assume a maximum transmit/receive rate of 100Mbit/s, achievable by employing 64QAM modulation throughout. Under these assumptions, a rough measure of the computational requirements for the chosen blocks is as follows:

- *QAM Encoder*: assuming QAM64 modulation, six integer multiply/accumulate ops per sextet of data bits, plus two array-access ops per symbol, coming to 133 MIOPs/s.

- *IFFT/FFT*: Assuming a Radix-4 FFT of 512 points yields a requirement for 4.8GIOPS/s per FFT. Since we have one IFFT and two FFTs in our example, the above figure must be tripled. We assume here that the unity complex roots (twiddle-factors) for the FFTs have been calculated offline and reside in memory.
- *Maximum Ratio Combiner*: For the 2x1 MRC/equalizer, assuming 8 multiplications and 3 additions per sample-point and channel, plus four array-access ops per sample, we come at approximately 110MIOPs/s.
- *QAM Decoder*: Assuming all incoming traffic was modulated in QAM64, a total of 7 fixed-point operations plus 22 integer operations per symbol requires a maximum capacity of 470 MIOPs/s.
- *Cyclic-Prefix Insertion/Removal*: The operation of this block consists almost exclusively of in-memory reorganization of the working data buffer. The computational requirements are thus approximately 2,8million memory read-write ops/s, plus 2,8MOPs/s for index arithmetic, per CP block.
- *Scrambler/de-scrambler block*: These blocks operate bit-serially, thus making inefficient use of the processor's datapath. Taking this point into consideration, and assuming no compiler optimisation of the required operations, 1,4GOPs/s are required for blocks capable of delivering a 100Mbit/s data stream.
- *1-by-2 Channel Model*: Based on the computational load as defined by the channel's description, 130MOPs/s plus 10 Million math-library function-calls/sec are required by this block. Assuming that each function call is of the order of 10 Ops, we arrive at a rough estimate of 250MOPs/s for the channel model block.

Neglecting for now the computational overhead for the implementation of the multipath channel model shown in Fig. 8.3, we can make the following observations: First, that the computational load of the IFFT/FFT blocks exceeds the needs of all other blocks combined by approximately a factor of 10. If this point is taken into account, the computational requirements for a single Tx-antenna/twin Rx-antenna PHY-layer implementation for 802.16e can be expected to stay just below 20GIOPS/s.

8.5 Evaluation Results

8.5.1 Code Outline of the Application

The system outlined in Fig. 8.3 was originally coded in fixed-point 'C' and validated by running on a Linux-running PC. Output files generated by running the code are stored for reference against which similar output of the parallelized versions of the code were compared for consistency. The code outline is shown below.

```
#include <various_libs.*>
#include <user-defined-functions-prototypes.h>
#include <user-defined-constants.h>
```

```
unsigned char     buff_a[SYMBOL_SIZE],....,
buff_j[SYMBOL_SIZE][2];
int       buff_a[SYMBOL_SIZE], ....,
buff_j[SYMBOL_SIZE][2];
.....
main()
{
unsigned char local_c, local_1, local_2...., local_n;
int       l_buff_a[SYMBOL_SIZE][2],....,
 l_buff_i[SYMBOL_SIZE][2];


<read-only table and constants initialization>
<file I/O initialization>
  for (iter = 0; iter < LOOP_SIZE; iter++ {
    // Read a number of characters from file
       pointer rp.
  // Scramble and partition the byte stream into
     bits groups
// according to modulation scheme.
   // Generate FFT_SIZE number of QAM samples.
   // Perform iFFT and CP insertion on FFT_SIZE
      number of
// QAM-encoded samples.
    // Emulate a two path channel featuring
       phase/amplitude
    // distortion and AWGN noise.
    // Perform FFT on symbols coming from path A of
       channel model.
    // Perform FFT on symbols coming from path B of
       channel model.
    // Run MRC on the outputs of FFT_A and FFT_B.
    // QAM decode the output of the MRC.
    // Assemble bit groups into characters and
       unscramble them.
    // Write the resulting character block into file
       pointer wp.
}
   close(rp):
   close(wp);
}
```

As can be seen, the application sequential code consists of an initialization section, with the blocks illustrated in Fig. 8.3 having been modelled as a succession of code fragments inside the "for(...)" loop which represents data traffic during simulation.

8.5.2 Parallelization Opportunities and Their Handling

The code outline of the previous section is a clear indicator of our methodology towards parallelization: A clean partition of processing steps, without a single feedback loop connecting any block, lends itself naturally to a pipeline of concurrent functional blocks, each one working on data supplied from the block upstream while supplying data to the block (or blocks in the case of the channel model) downstream. This view influenced code restructuring in the manner indicated below.

```
for (iter = 0; iter < LOOP_SIZE; iter++ {
     Read_Scramble: {
       // Read a number of characters from file
          pointer rp.
     // Scramble and partition the byte stream into
        bits groups as per modulation scheme.
     }
 QAMenc_iFFT: {
     // Generate FFT_SIZE number of QAM samples.
   // Perform iFFT and CP insertion on FFT_SIZE
       number of QAM-encoded samples.
    }
    Channel: {
     // Emulate a two path channel with
        phase/amplitude distortion and AWGN noise.
    }
    FFT_MRC_QAMdec: {
     // Perform FFT on symbols coming from path
        A of channel model.
     // Perform FFT on symbols coming from path
        B of channel model.
     // Run MRC on the outputs of FFT_A and FFT_B.
     // QAM decode the output of the MRC.
   }
   Unscrable_Write: {
     // Assemble bit groups into characters and
        unscramble them.
     // Write the resulting character block into
        file pointer wp.
   }
}
```

As shown in the shaded box above, code inside the functional block has been partitioned into a number of labelled statement blocks (called “kernels” in ATOMIUM terminology). Each block (or kernel) encloses a number of the functional blocks outlined in Fig. 8.3. There is no one-to-one correspondence of functionality with the

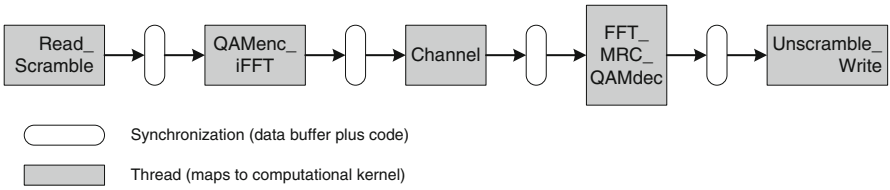

Fig. 8.8 Functional split of application code

labelled kernels (e.g. one kernel only includes the Channel model, while another kernel can include both FFTs, the MRC and the QAM decoder). The reasons for this partition will be discussed in more detail in the following sections.

8.5.3 Parallelization Schemes and Their Evolution

Before using the IMEC instrumentation and parallelization tools ('anl' and 'mpa'), the source code had to be adapted in order to conform to the Clean-C guidelines from IMEC. Instrumentation was then added in the form of user-modifiable function calls that the 'anl' utility is employing for the purpose of measuring execution time of the kernels (the "enterCB_record()' and "leaveCB_record()" functions supplied in template format in the IMEC design environment).

One must note that the "granularity' of time-measuring by the kernel-monitoring functions was rather poor, so that artificial time-counting overheads were introduced by modifying the profiling functions in order to extract kernel execution times other than just zero. This was judged necessary so that the execution profile screenshots, generated by post-mortem visualization of simulation data, would display a more realistic execution activity, instead of uniform chunks of activity irrespective of computational load per kernel. Admittedly though, the activity profile reported by 'hlsim' can only serve as an abstraction of actual concurrent activity, as it is not capable of modelling path- and traffic-dependent communication delays on a NoC platform such as those of interest to MOSART.

The first attempts at parallelization adopted a natural mapping of computational kernels, as indicated by the labelled blocks outlined in the previous section, into individual threads, as shown in Fig. 8.8. In ATOMIUM terminology, Fig. 8.8 illustrates what is called a functional split of the original sequential code, i.e. generation of threads that map to one or more functions that contain the definition of computational kernels. Remember that the parallelised code is only the segment enclosed by the "for(iter = 0; . . . ..)" loop.

Eventually, after analysis of the parallelism that can be extracted from the application's functions the parallelization scheme of Fig. 8.9 was adopted. This scheme is a combination of functional and data split, with read-scramble being mapped to a single thread, feeding into a multitude of QAM-encode/iFFT threads,

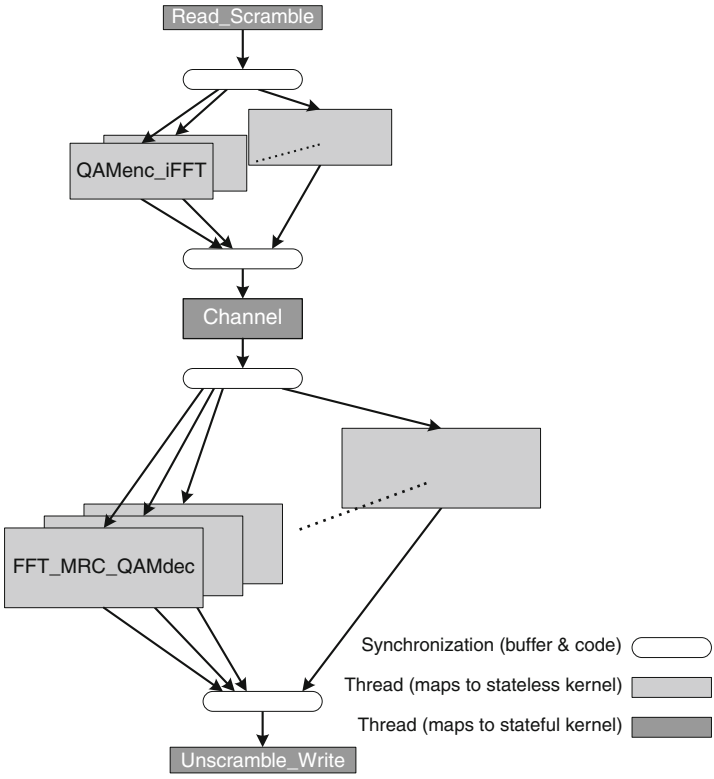

Fig. 8.9 Combined functional/data split of sequential code

followed by the Channel model mapped to a single thread, followed by another set of data-parallel threads that map onto the FFTa/b, MRC and QAM-decode combination kernel, followed by the unscramble/file-write kernel mapped onto a single final thread.

The mapping of Fig. 8.9 encodes several observations about the application: first, it addresses the state-preservation problem of stateful processes by having these kernels (identified in orange in the figure) mapped onto single threads. Second, it is a reflection of knowledge about the modelled system's properties, i.e. the fact that the QAM-encode/iFFT kernel and the FFTa,b/MRC/QAM-decode kernel are both heavy computationally and stateless, i.e. they are mere functions (albeit of significant complexity) applied onto the data stream. Thus, it makes sense to generate multiple copies of these threads in order to gain from their concurrent execution. And because the FFTa,b/MRC/QAM-decode threads are (roughly) twice as heavy as the QAM-encode/iFFT threads, it also makes sense to have more copies of the former than the latter type of thread. Unfortunately, the thread that is mapping the Channel model kernel, being stateful, cannot be replicated, despite being also computationally "heavy". Worse still, the thread sits right in the middle of the

pipeline of functional blocks, thus creating a bottleneck for parallelization. The effect of this topological feature of the application on speedup and performance is discussed later in this section.

In order to address spurious data-dependencies between members of the thread-sets handling QAM-Encode/iFFT and FFT/MRC/QAM-Decode, that severely degraded achievable speedup, the *loopsync* directive provided by MPA was used with the parallelization scenarios (the "par.spec" files). With this scheme, the split of the data-set between the threads of a group is controlled by having the upstream buffer feeding into the downstream threads replicated by as many times (approximately) as there are threads to share the computational load. Additional code in the application source controls update and access of the multi-buffer entries.

Finally, the "par.spec" file needs to declare the multi-buffer as shared, to prevent FIFO insertion by 'mpa'. By this measure, we finally achieved significant speedups for various thread counts, as discussed in the next section. Execution of the parallelized code gave results consistent with those of the sequential reference code and thus we were able to accept the resulting code as correct. Using loopsyncs resulted in significant reduction of the FIFOs needed for inter-thread communication and a consistent correlation between thread-count and speedup was obtained.

A downside of the use of loopsyncs was that source-code is now coupled to the "par.spec" file that generates the multi-threaded version of the reference source. This is less than ideal, although in our case we chose to declare multiple buffers only once, making them large enough to accommodate all the multi-threaded scenarios we cared to explore. In this manner, only constants in the source code needed editing; a step that can easily be replaced by scripting, in order to support automated regression runs.

8.5.4 Use of the VTT ABSOLUT Suite

Following a satisfactory outcome of the sequential software parallelization process, the resulting code needs to be translated for use by the ABSOLUT environment, where the parallel code is compiled and linked to a System-C model of a multi-core platform of the chosen architecture. The resulting object is simulated in order to extract performance metrics such as memory and communication overheads, impact of communication overhead on computation, multi-core efficiency vs. sequential execution, etc.

8.5.5 Use of the "g++absith" Compiler and of the Python Post-Processor

The makefile used to control generation and compilation of the instrumented and parallelized code was modified, in order to support a two-step process of

(a) extracting use-profiles of user functions calls and (b) use of these profiles for the generation of the workload models that are employed by the ABSOLUT environment for simulating code execution on a properly configured virtual multi-core platform. During the first part of this process, 'g++absinth' is used to compile the parallelized code generated by the 'mpa' tool. When executed, the binary thus created is producing, in addition to its nominal output, a set of files having the .gcda suffix. These files contain profiling information for all user-defined functions that make-up the targeted application. During the second part of the process, the 'g++absinth' compiler is invoked again, but this time with a switch specifying that the previously generated profile information be used for generation of the workload models making-up the user application. In addition to the executable binary, a set of C++ files and headers are produced, alongside workload model files (called "basic block" files, having the suffix.bb).

The .bb files contain instances of service models that correspond to mps_xxx() function calls from the ATOMIUM RtLib library. Some additional file-editing work is required for binding the application process with the target platform model, updating of the CMakefile that configures and generates makefiles, before finally compiling and executing the binary that simulates the application running on the virtual multi-core platform.

8.5.6 An Automated Platform Generation Utility

Exploitation of large multi-core platforms involves experimentation with node configurations of varying size and topology, in order to find the best fit between the type and amount of parallelism extracted by the ATOMIUM tool and the particular characteristics of the target platform. For demanding applications running on large multi-core arrays, the amount of work required for generation of the platform variants may turn out to be very large.

Therefore, in order to aid the process of platform generation and thus improve turnaround times during platform mapping exploration, ICOM developed a small utility called 'build_grid' and running in a Linux environment. The utility automatically generates user-selectable sizes and topologies of multi-core platform source-files, based on template files that have been augmented with a proprietary syntax for regular expressions. The utility is supplied at run-time with arguments that control the generated platform topology: The user can select among a ring, a flat rectangular grid open in both dimensions, a grid closed in one dimension (a "tube") and a grid closed in both dimensions (a torus). The size of the platform must be supplied as an argument to the utility; the number of nodes if a ring is specified, or the number of nodes along each of two dimensions, if a grid variant is chosen. If a grid is chosen, the generated variant (flat, tube or torus) depends on the template file base name, also supplied if different from the default at program invocation.

The use of 'build_grid' allowed quick generation of the various platform topologies examined in the course of platform exploration.

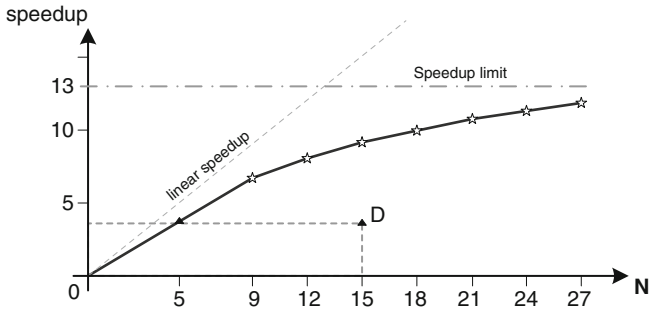

Fig. 8.10 Graph of speedup-vs-number of threads for the parallelized 802.16e PHY application

8.5.7 Speedup Obtained from Parallelization

The picture of Fig. 8.10 illustrates the parallelism that was extracted with varying numbers of threads in two different topologies. The horizontal axis refers to numbers of threads allocated to the application's computing tasks, whereas the vertical axis indicates the obtained speedup, as measured by the ATOMIUM 'hlsim' high-level simulator. As can be seen in the figure, the obtained speedup is less than the ideal ("linear") speedup, due to the non-zero (but fixed-size) communication delay modeled by 'hlsim'.

A significant (and expected) corollary from the adoption of the combined functional/data-split for parallelizing our application was the observation in practice of Amdhall's Law, which gives speedup limits of a system in the presence of sequential bottlenecks, such as the AWGN Channel model in our case. Indeed, the obtained speedup is clearly trailing-off as more threads are added to the multi-sets, reaching asymptotically an upper limit of approximately x13 times speedup. The reason for the reduction of speedup with increasing number of threads is that, effectively, the tasks which can only be mapped onto a single thread impose an upper limit on speedup, since they become bottlenecks. In the limiting case, 'hlsim' reports that the sequential part of the parallelized application workload corresponds to approximately 1/13th of total execution time, with all parallel activity converging to zero time, as the number of threads for the parallelizable section goes towards infinity.

The picture of Fig. 8.10 is the idealization reported by 'hlsim' (the IMEC-supplied high-level simulator), which models communication overheads in a very abstract manner. However, even this elementary modelling mechanism is sufficient to reveal the impact of the chosen parallelization scheme: the data-point "D" of the diagram corresponds to a 15-thread parallelization scenario where inter-thread synchronization is entirely left to the MPA analysis algorithm. As a result, excessive numbers of communication FIFOs were employed, due to a non-trivial data dependency that was not possible to locate and remove from the source code. The consequence of the large number of FIFOs was a speedup of approximately

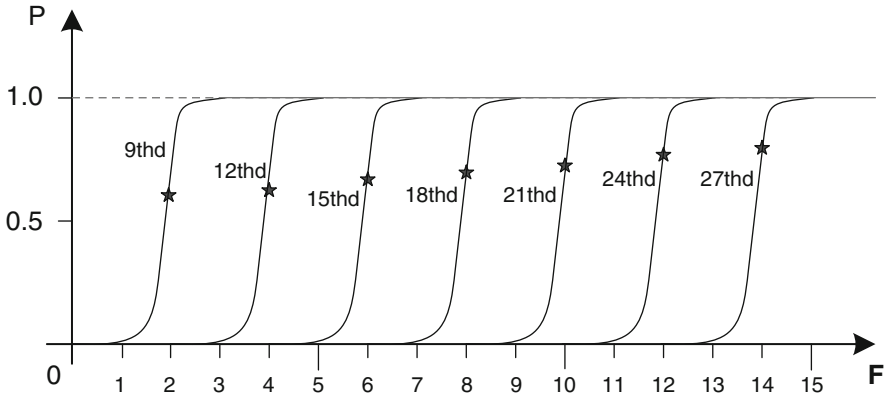

Fig. 8.11 Graph of performance degradation vs comm. FIFO size for various numbers of parallel threads. The loopsynch mechanism is the chosen parallelization scheme for these tests

3.6 for the 15 threads. In contrast, the line of the graph corresponds to data-points collected after the loopsynch communication mechanism was adopted for the parallelized code.

8.5.8 Performance Impact from Communication FIFO Sizes

The impact of the size of the communication FIFOs connecting threads was shown to be quite significant on the performance of the parallelized application. The graph of Fig. 8.11 illustrates the performance degradation **P** observed for various sizes **F** of communication FIFOs vs number of parallel threads. In this context, **P** is defined as the ratio of speedup as a function of FIFO size, over the speedup achieved with unconstrained FIFO size.

From the graph we observe an abrupt collapse in the simulator's performance as the FIFO size goes below a threshold value, which is closely correlated to the number of threads in each of the thread sets. This is due to the chosen parallelization scheme, whereby the data-buffers interconnecting the application's thread-sets are implemented as multiple buffers, in order for the loopsynch mechanism to control the run-time binding of particular buffers to each thread of a multi-thread set. In contrast, when the fully automatic synchronization scheme is chosen for the 15-thread scenario (data-point "D" in Fig. 8.10), the measured speedup is entirely insensitive to the size of the communication FIFOs, all the way down to single-entry FIFOs.

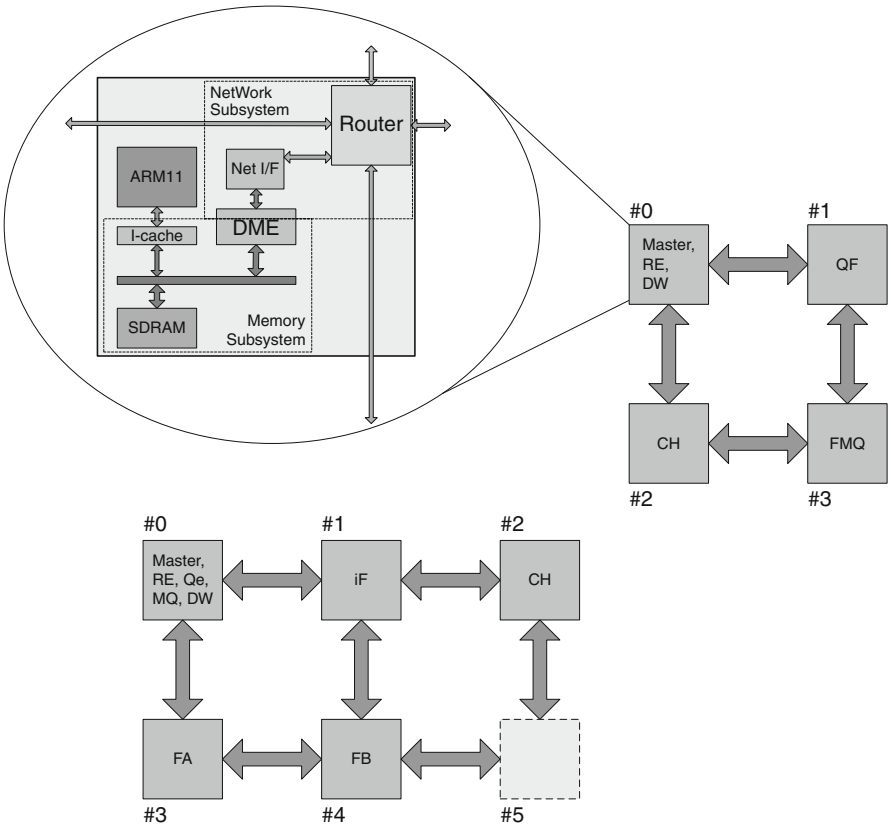

Fig. 8.12 Outline of platforms for ABSOLUT simulations

8.5.9 ABSOLUT Environment Simulation Results

A smaller number of simulations was performed, compared to the simulation runs of 'hlsim' that produced the graphs of the previous section, due to some loss of consistency between the MPA and ABSOLUT. However, there is sufficient amount of data for partially characterizing the MPA/ABSOLUT interface.

The platform employed for the ABSOLUT simulations was of three different configurations as seen in Fig. 8.12: single node, a 2×2 grid and a 2×3 grid of PEs (in the later case, one node was inactivated). Each node features an ARM9-based processor with instruction and data caches, an amount of DRAM (for our simulations; SRAM is also an option), the network interface that comprises with its peers the NoC fabric and a DME engine for setting-up DMA transfers across the PE fabric.

The first observation from the ABSOLUT simulator log files is that the obtained speedup is reduced compared to the one reported by the 'hlsim'. This is to

be expected, since the abstract simulator does not model the platform in detail. In addition, modeling accuracy of speedup by 'hlsim' may be influenced by the precision of kernel execution time-logging. Therefore that actual communications cost as modeled by ABSOLUT does not enter the statistics by 'hlsim', resulting in a more optimistic picture, compared with ABSOLUT data. Still, there is significant speedup obtained:

- $\times 2$ improvement of execution time for a 4-node platform, i.e. an efficiency of 50% per processor on average, for a 5-thread parallel version of the application mapped onto the 4-node processor.
- $\times 3$ to $\times 6.6$ speedup for a 5-node platform, i.e. an efficiency of 60% per processor for an eight-thread version mapped onto a 5-PE platform.

8.5.10 Effect on Granularity on Parallelism and Load-Balancing

A five-thread model was run on a four-node simulated platform. The five threads generated correspond to the base case of the data-split driven family of models, as illustrated in Fig. 8.8. As such, it is characterized by a significant imbalance in the allocation of workload: thread #3 maps the computational kernel that is doing two FFT transforms plus the MRC combiner and QAM-Decode tasks. Compared to threads #2 (single iFFT transform) and #3 (channel model), this thread is disproportionately sized, to say nothing of threads #1 and #5 (file-I/O and scramble/unscramble). Therefore, the speedup figure obtained for the four-node platform is an average figure between four processors that are not evenly loaded, as illustrated by the graph of Fig. 8.13. Looking at the graph, we see that processor #3 is kept busy practically all the time, while processor #0 is practically idling. It is obvious from the graph that load imbalance leads to the effective elimination of PE#0 from the work-pool. Furthermore, it is easy to deduce that coarse granularity can lead to such an imbalance, if a naïve static allocation scheme is employed for mapping tasks to processing elements.

8.5.11 An Improved Load-Balance Solution

The observations on the 5-thread model were actually used as a guide for exploring more a efficient solution involving eight threads. Some code tweaking was again required, alongside a new parallelization scenario for the MPA. The different mapping of source-level computation kernels to threads was driven by the observation that thread #4 of Fig. 8.13 actually corresponds to something like 50% of all computation for the application. Therefore, the two FFT operations and the MRC operation were rendered as individual threads. Lighter threads such as file-I/O and scramble/unscramble, QAM-Encode and Decode and also the MRC thread were bunched together onto a single PE of a 5-PE platform (actually six PEs,

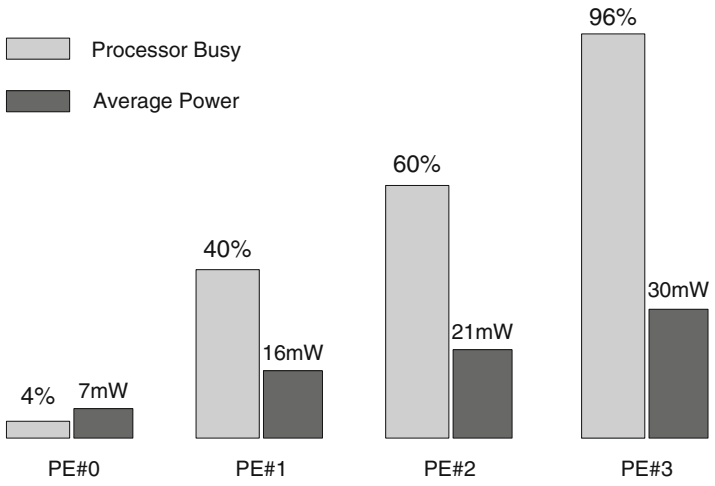

Fig. 8.13 Load balance and processor power figures for the 5-thread

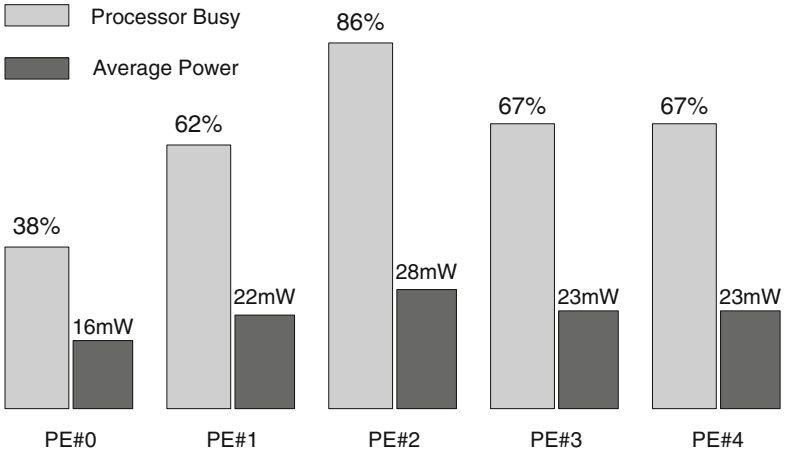

Fig. 8.14 Load balance and processor power figures for the 8-thread model running on a 5-PE platform

with the sixth processor inactive). The iFFT, Channel, FFT_a and FFT_b threads were mapped onto individual PEs. This mapping resulted in a much better load distribution among processors, as shown in Fig. 8.14.

Looking at the activity chart of Fig. 8.14, we see that processors #1, #3 and #4 feature nearly identical activity levels of 62%–67%. This is to be expected, since all three are running the same instance of a 512-point FFT transform (no difference in performance for the iFFT runningon PE #1). The largest activity in the chart comes

from the channel model running on PE #2 which is not trivially parallelizable. This means that additional coding effort would be required, in order to have gains from parallelization of this task.

8.6 Metrics Derived from Platform Evaluation

We will discuss in this section a number of performance metrics associated with running the application on the simulated multi-core platform and also the application development time when compared against conventional design flows.

8.6.1 Power Performance Figures for the Test Cases

The figures for number of instructions executed, instruction count and power consumed per case are tallied in Table 8.1. From these figures, the figure for MIPS/Watt (i.e. the power efficiency of the NoC/DME platform) can be extracted. The calculated figure varies from case to case, due to the variable percentage of memory-fetch vs instruction-execution from case to case.

The execution time for the FPGA (Virtex-5 XCV5FX130T) coincides with the application's real-time constraint, i.e. this is the time available for running it.

From the above table we see that the MIPS/Watt figure for the multi-core platform does not change by more than about 30% from case to case. This is to be expected, since this figure is not a function of platform extension but is an intrinsic property of the particular processor & memory choice, clock speed and underlying

Table 8.1 Power-performance figures for single- and multi-core test cases

Test case	Proc. power (mW)	Mempower (mW)	Instruction count	Time (ms)	Mips/Watt
Sequential Code on Pentium 4 @ 2.4GHz	∼50,000	∼10,000	∼400,000,000	180	6.67
5 threads on a single PE	31.60	44.09	492,462,047	17,792	366
5 threads on a 4-PE platform	75.15	158.51	488,920,405	6,987	300
8 threads on a single PE	31.6	47.22	281,663,593	8,171	437
8 threads on a 5-PE platform	111.53	206.53	287,104,188	2,667	338
FPGA	400		–	4	–

Table 8.2 Data storage requirements for MOSART vs conventional methods

Test case	RAM requirements	ROM requirements
Sequential 'C' code running on x86 PC	57 kbytes	–
Multithreaded code running on a single-PE NoC	142 kbytes	–
5-thread code running on a 4-node PE NoC	142 kbytes	–
8-thread code running on a 5-PE NoC	152 kbytes	–
VHDL-derived netlist mapped onto FPGA	98 kbytes	4 kbytes

technology assumptions. Differences in the reported figures can be explained by taking into account the relatively small differences in the percentage of memory accesses vs instruction execution per processor from case to case. An additional reason for the more-or-less constant reported figures is that no modification was made to the node memory size for the various test cases.

Regarding the FPGA test-case, the instruction-count entry is not applicable here, as the dedicated HW datapath does not execute a program. If we define elementary logical and arithmetic operations as the unit of activity (OPS as in "Operations Per Second"), we come up for the particular demonstrator with a figure of approximately 20GOPS/Watt. One must note that not all of the operations in the datapath carry the same weight (e.g. multiplications compared to shifts), so the figure is not as impressive on closer examination. Also, while the efficiency of the multi-core platform may seem low, it should be stressed that it is based on general-purpose processors (ARM) and not on application-optimised ASIPs.

NOTE: Pentium figures are from literature (power for processor and memory) and from indirect instruction-count calculations. Execution time was measured with the 'time' Linux facility.

8.6.2 Data Storage Efficiency

Table 8.2 is a comparison of the memory overhead for the application, mapped onto the tried NoC platforms and a FPGA netlist, compared to the storage requirements of the sequential application running on a commodity PC (Intel x86 processor).

A few remarks are in order, to clarify the above comparisons: First, we take as a baseline the figure for sequential execution on a PC platform. Second, the figure for the FPGA includes internal storage for the iFFT/FFT macros, which feature SRAM

buffers for in-place data-processing by the complex multiplication butterflies. Third, the figures for the NoC memory usage refer to the data-storage alone- not code area, since we want to compare similar resource requirements between implementation techniques. Note that most of the storage overhead associated with the MOSART platforms is due to the FIFOs used as a data-exchange mechanism between threads. Improvements to the inter-thread communication mechanism can significantly reduce this overhead. Note: The FIFO count is the same for the 5-threaded code irrespective of processor count, hence the identical figures for memory use.

8.6.3 Multi-Core Speedup vs Uniprocessor Performance and Extra-Polations

A speedup of 2.07 was measured for the 5-thread application when running on a 4-PE platform compared to execution speed of the same parallelized application running on a single processor. This corresponds to an average idle time for the processors of approximately 50%.

A speedup of 3.05 was measured for the 8-thread application when running on a 5-PE platform, compared to speed of the application on a single processor, corresponding to average processor idle time of approximately 40%.

The above speedup figures are indicative of significant communication overheads between the threads mapping the application's computational kernels. Note that by communication we do not mean just the time taken by the NoC's links to transfer data across; indeed the figures we have from the ABSOLUT simulations on busy time for the network interfaces of the platform's nodes never exceed a fraction of 1% of simulation time. However, our application is characterized by the need to pass along very large data buffers of a few kilobytes. This fact, coupled with a feature of MPA-derived code that disallows computational activity of the communicating threads for the duration of the buffer's transfer does explain the observed communication latencies. Of course, speedup figures are also eroded by the load imbalance inherent in a coarse-grained parallel model.

A solution to these problems would be a total restructuring of the application, in order to allow fine-grain parallelism (i.e. many tens of threads at a minimum), while keeping a very tight control on communication latencies across the multi-core platform. The restructuring was not tried for lack of sufficient time in this project. It is however relatively easy to identify a few salient points to such an effort:

First, the amount of memory associated with each processor should be significantly reduced, especially if the denser but more power-hungry DRAM is chosen over the faster SRAM. Otherwise, the goal for power-efficient computation will be compromised. Second, the application to be mapped onto a large number of threads will typically require extensive redesign, if coming from a sequential (uniprocessor) execution starting point. Alternatively, code for the target application should be

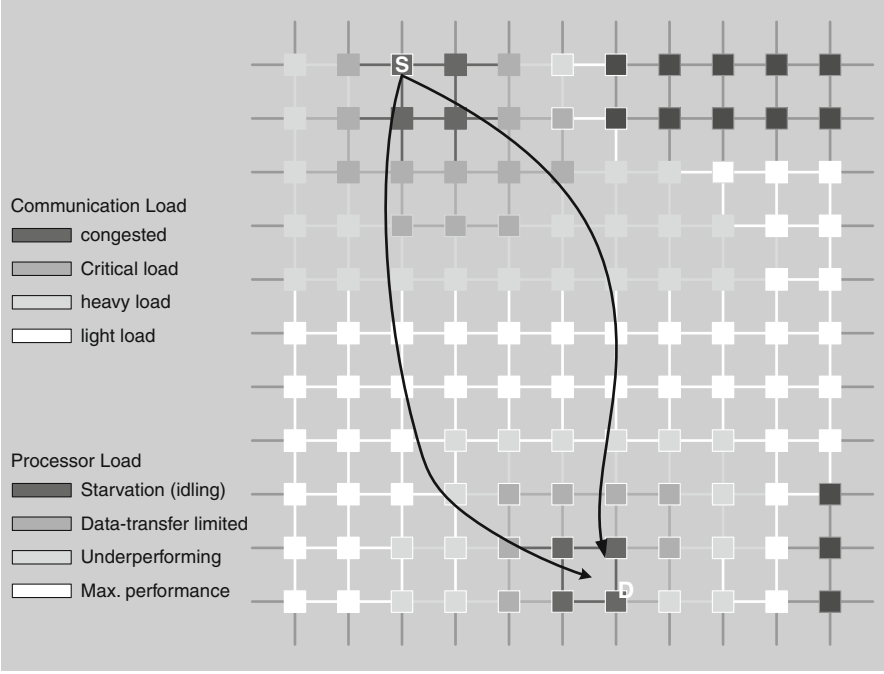

Fig. 8.15 Illustration of a communications and processing activity profile on a large 2-D NoC platform. Communication "hot-spots" are formed around the points where data enter (processor "S") and leave (processor "D") the array

developed from scratch having in mind a NoC target platform with the following crucial characteristics:

1. Large number of discreet memories associated with processors.
2. Point-to-point communication links between processors that are characterized by a minimum (ideally fixed) data-transport cost, both time- and power-wise.

The above two point lead to the conclusion that the one thing that must be avoided at all costs when such a programming paradigm is adopted is excessive inter-processor communication, especially between distant platform nodes. An abstract representation of the situation is depicted in Fig. 8.15. In the picture, a large amount of processors is busy on regular (and small) partitions of a large data-set. The results of their processing need to be concentrated and serialized at a small number of downstream nodes (one in the limiting case). The network interfaces of nodes are progressively burdened with more traffic, as they are getting closer to the serializing node(s). With a sufficiently large number of processors, a "horizon" situation is created: Processors are no longer able to keep constantly busy, because of traffic that is congested near the data "sink". The exact symmetrical situation exists with a data stream that needs to originate from a small number of input nodes: downstream nodes are idling excessively because the data-supplying node cannot

push data through its local network neighbourhood fast enough. Also, processes that are characterized by internal state may be encountered in an application and they are essentially an instance of the above described phenomenon, although algorithmic solutions for state replication and communication between processors may be available.

In light of the above, additional tool resources would probably be necessary for the exploitation of large NoC platforms. Also, the tools will need to balance the desire for a smooth power profile over a large 2-D array against the additional bottlenecks this strategy might entail: A "hot spot" would have to have its speed (and thus power consumption) reduced right where speed is needed most to avoid congestion in data-traffic.

8.6.4 Design Time with MOSART vs Current Practice

If we exclude the learning curve associated with learning and familiarizing with the MOSART flow (and also the time to remove bugs from the parts making-up the toolflow) we can measure the time taken to render our application into a multi-threaded form at approximately 2 weeks, starting from a given sequential C-source. During that time, source code evolved from its original form into one better able to exploit available parallelism, although still at the coarse-grained level. Various scenarios involving functional and data-space-driven partitioning of the application into parallel threads were developed and evaluated by means of the first part of the tested flow (the ATOMIUM suite). A subset of the developed parallelization schemes was also run through the ABSOLUT simulator and metrics regarding speedup and power efficiency were extracted.

Development of the same algorithmic chain in a netlist mapped onto an FPGA by synthesizing VHDL derived from manual translation of the same sequential C-code, has taken 10 weeks, also taking into account that the iFFT/FFT blocks were IPs supplied by the FPGA vendor. We therefore have an improvement in design time of approximately 5 times. This is a great improvement over the current state of the art, although the figure would not be as large if more elaborate schemes for fine-grain parallelism had to be developed, starting from the sequential application. Still, the fact remains that the exploration space for the engineer is characterized by fewer parameters, namely the nature of the starting sequential application and a small number of parallelization schemes offered by the MPA tool, plus a choice from among a set of target platforms, that may include custom ASIPs as well. In contrast, a VHDL designer would have to cope with various styles of transcribing a design from 'C', MATLAB or similar environments, a sizable menu of synthesis guidelines and optimization techniques and finally tradeoffs between FPGA or ASIC target families, in order to achieve optimum results. The runtime alone for the synthesis tools makes for a design iteration of several hours (involving a loop of [source-level editing −> verification of changes by RTL simulation −> synthesis −> post-synthesis verification simulation]), compared to less than one hour for the [source-editing −> 'MPA' −>'hlsim'] loop.

8.7 Required Work for a Production-Level MOSART Flow

The portion of the evaluated toolflow demonstrated a viable and promising methodology for developing parallel application for targeting onto multi-core NoC platforms. Some more work would transform the current proof-of-concept MOSART paradigm into a robust engineering tool that can withstand the demands of a production environment. A short list of additional work items according to ICOM would be:

- Addressing the stability of MPA: The tool should be made more robust, so that code that conforms to the clean-C standard does not produce cryptic error messages or code with unexpected runtime behaviour.
- Seamless integration with ABSOLUT: Handover of parallel code to the ABSOLUT flow should be entirely transparent to the user.
- Synopsys-Coware integration with MOSART: Generated ASIP processor must integrate transparently with the NoC platform hardware, so that a systmC model of the processor is embedded in ABSOLUT automatically. Also, the generated SW support (compiler, linker, etc) associated with the custom ASIP must be integrated with the ABSOLUT and CAMALA flows.
- Some support should be integrated for quick generation of user-specified processor fabrics: number of nodes, dimensions of fabric, inactive nodes, processor type for each node or groups of nodes (to support heterogeneous NoCs).
- Support should be supplied to the user for efficient mapping of threads onto processors. This task may be simple enough for trivial examples, but for the case of dozens or hundreds of threads in as many processors, the problem resembles closely the optimization problem faced by the place & route algorithms for HW design. Thus, solutions similar to the ones employed by post-synthesis P&R tools for ASICs and FPGAs will probably be required.
- Visualization of ABSOLUT simulations, with emphasis on critical performance indicators: processor & memory activity, topology and lifetime of DME-mediated transfers, communications network load indicators, trace capability associated with NoC resources (activity profiles, connectivity information, etc) or with application attributes (code profiles, variable traces, etc).
- Finally, an IDE for presenting a unified and easily manageable interface to the user would be a considerable productivity boost.