Michael Hübner
Jürgen Becker
Editors

Multiprocessor System-on-Chip

Hardware Design and Tool Integration

Springer

Multiprocessor System-on-Chip

Michael Hübner • Jürgen Becker
Editors

Multiprocessor System-on-Chip

Hardware Design and Tool Integration

Springer

Editors
Michael Hübner
Karlsruhe Institute of Technology (KIT)
Institut für Technik der
Informationsverarbeitung
Vincenz-Prießnitz-Straße 1
76131 Karlsruhe
Germany
michael.huebner@kit.edu

Jürgen Becker
Karlsruhe Institute of Technology (KIT)
Institut für Technik der
Informationsverarbeitung
Vincenz-Prießnitz-Straße 1
76131 Karlsruhe
Germany
juergen.becker@kit.edu

ISBN 978-1-4419-6459-5 e-ISBN 978-1-4419-6460-1
DOI 10.1007/978-1-4419-6460-1
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: xxxxxxx

Printed on acid-free paper

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

Preface

For the next decade, Moore's Law is still going to bring higher transistor densities allowing Billions of transistors to be integrated on a single chip. However, it became more and more obvious that exploiting significant amounts of instruction-level parallelism with deeper pipelines and more aggressive wide-issue superscalar techniques, and using most of the transistor budget for large on-chip caches has come to an dead end. Especially, scaling performance with higher clock frequencies is getting more and more difficult because of heat dissipation problems and too high energy consumption. The latter is not only a technical problem for mobile systems, but is even going to become a severe problem for computing centers because high energy consumption leads to significant cost factors in the budget. Improving performance can only be achieved by exploiting parallelism on all system levels.

Therefore, for high-performance computing systems, for high-end servers as well or for embedded systems, a massive paradigm shift towards multicore architectures is taking place. Integrating multiple cores on a single chip leads to a significant performance improvement without increasing the clock frequency. Multicore architectures offer a better performance/Watt ratio than single core architectures with similar performance.

Combining multicore and coprocessor technology promise extreme computing power for highly CPU-time-consuming applications in scientific computing as well as for special purpose applications in the embedded area. Especially FPGA-based accelerators not only offer the opportunity to speedup an application by implementing their compute-intensive kernels into hardware but also to adapt to the dynamical behavior of an application.

The purpose of this book is to evaluate strategies for future system design in MPSoC architectures. Both aspects, hardware design and tool-integration into existing development tools will be discussed. Also the novel trends in MPSoC combined with reconfigurable architectures are a topic in this book. The main emphasis is on architectures, design-flow, tool-development, applications and system design.

We kindly want to thank all authors and co-authors for their excellent contributions which enabled finally the development of this book. Furthermore we want to thank the Springer Team, namely Mrs. Amanda Davis, Mr. Charles Glaser and Ms. Jeya Ruby for their great support and patience.

Karlsruhe, Germany

Jürgen Becker
Michael Hübner

Contents

Chapter 1
An Introduction to Multi-Core System on Chip – Trends and Challenges

Lionel Torres, Pascal Benoit, Gilles Sassatelli, Michel Robert, Fabien Clermidy and Diego Puschini

1.1 From SoC to MPSoC

The empirical law of Moore does not only describe the increasing density of transistors permitted by technological advances. It also imposes new requirements and challenges. Systems complexity increases at the same speed. Nowadays systems could never be designed using the same approaches applied 20 years ago. New architectures are and must be continuously conceived. It is clear now that Moore's law for the last two decades has enabled three main revolutions. The first revolution in the mid-eighties was the way to embed more and more electronic devices in the same silicon die; it was the era of System On Chip. One main challenge was the way to interconnect all these devices efficiently. For this purpose, the Bus interconnect structure was used for long time. Anyway, in the mid-nineties the industrial and academic communities faced a new challenge when the number of processing cores became two numerous for sharing a single communication medieum. A new interconnection scheme based on the Network Telecom Fabrics, the Network On Chip was born; over the past decade intense research efforts have led to significant improvements. The last breakthrough was due to the need to interconnect a set of processors on the same chip, in early 2000. When previously developed systems embedded a single processor, the master of the chip, multiple masters must now share the overall control. The first Multi-processors System-on-Chip (MPSoCs) emerged [1]. They combine several embedded processors, memories and specialized circuitry (accelerators, I/Os) interconnected through a dedicated infrastructure to provide a complete integrated system. Contrary to SoCs, MPSoCs include two or more master processors managing the application process, achieving higher performances. Since then, an important number of research and commercial designs have been developed [2]. They have started to get into the marketplace and are expected to be widely available in even greater variety in the next few years [3]. It is now

L. Torres (✉)
University of Montpellier 2, UMR CNRS, France
e-mail: Lionel.Torres@lirmm.fr

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_1,

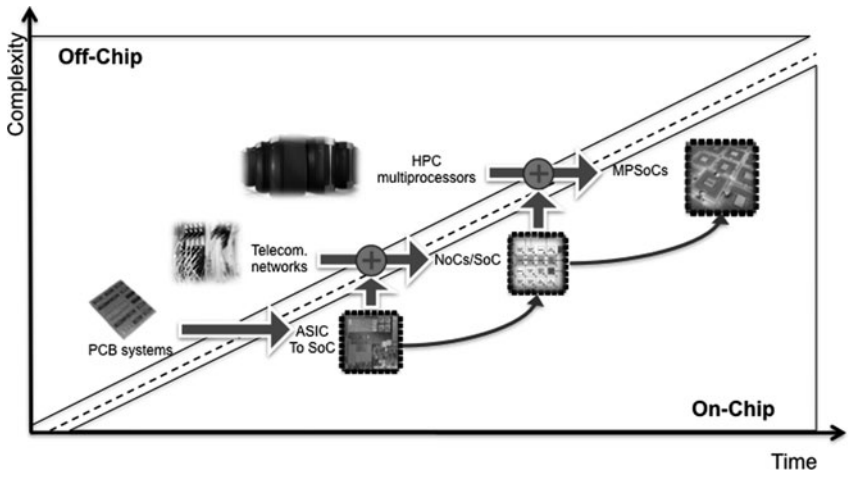


Fig. 1.1 From SoC to MPSoC

clear that this third revolution will change drastically the way to consider System On Chip Architecture. Figure 1.1 summarizes these 3 revolutions that occured in less than 20 years.

1.2 General Structure of MPSoC

This section describes a generic MPSoC, only introducing the key elements in order to formulate valid assumptions on the architecture. In general MPSoC is composed of several Processing Elements (PE) linked by an interconnection structure as it is presented in Fig. 1.2.

1.2.1 Processing Elements

The PEs of an MPSoC are related to the application context and requirements. We distinguish two families of architectures. From one side, heterogeneous MPSoCs are composed of different PEs (processors, memories, accelerators and peripherals). These platforms were certainly pioneered: the C-5 Network Processor [4], Nexperia [5] and OMAP [6], as shown in [2]. The second family represents homogeneous MPSoCs, pioneered by the Lucent Daytona architecture [2, 7], where the same tile is instantiated several times. This chapter targets both families and Fig. 1.2 represents either a homogeneous or heterogeneous design. For instance numerous works consider that processors as well as flexible hardware such as reconfigurable fabrics compose heterogeneous PEs.

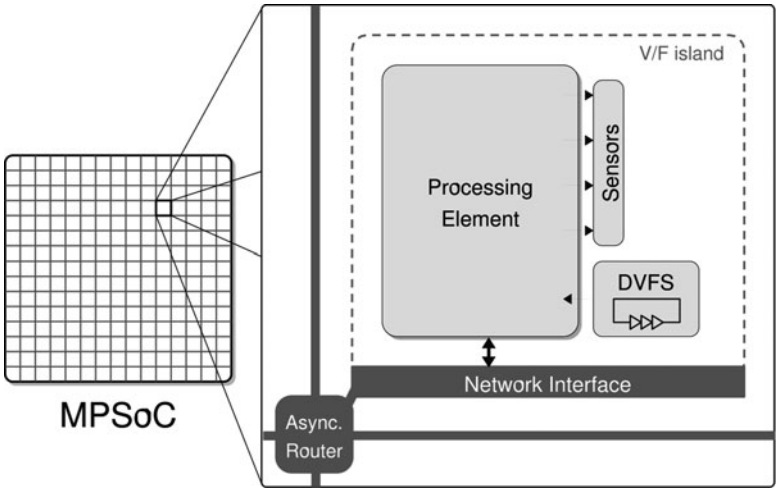


Fig. 1.2 General MPSoC architecture

1.2.2 Interconnection

The PEs previously described are mostly interconnected by a Network-on-Chip (NoC) [8–11]. A NoC is composed of Network Interfaces (NI), routing nodes and links. The NI implements the interface between the interconnection environment and the PE domain. It decouples computation from communication functions. Routing Nodes, also called routers, are in charge of routing are arbitrating the data between the source and destination PEs through the links. Several network topologies have been studied [12, 13]. Figure 1.2 represents a 2D mesh interconnect. The sizing of the offered communication throughput must be enough for the targeted application set.

The NoCs facilitate the design of Globally Asynchronous Locally Synchronous (GALS) property by implementing asynchronous-synchronous interfaces in the NIs [14, 15]. In Fig. 1.2, an example of an asynchronous router is presented to highlight this property.

1.2.3 Power Management

One of the major challenges nowadays is the way to achieve energy efficiency for embedded systems. The GALS feature allows partitioning the MPSoC into several voltage/frequency islands (VFI) [16]. In this example, each VFI contains a PE clocked at a given frequency and voltage. This approach allows fine-grain power management [17]. As in [18, 19], the considered MPSoC incorporates distributed Dynamic Voltage and Frequency Scaling (DVFS): each PE includes a DVFS device. The power optimization consists in adapting the voltage and frequency of

each PE in order to balance power consumption and performance. In more advanced MPSoCs, a set of sensors integrated within each PE provides information about consumption, temperature, performance or any other metric needed to manage the DVFS. Anyway, due to the cost of adding dedicated circuitry, coarser grain power management including multiple PEs in one VFI are used in many MPSoCs, providing a different level of control for the power management.

1.3 Power Efficiency and Adaptability

As presented in the introduction, MPSoCs are following Moore's law [20]. This empirical law has demonstrated to be true during several decades. Figure 1.3 shows some examples of processor with their transistor counts. But for MPSoCs, what are the challenges coming with Moore's law? More transistor density also means more performance (but also increased power consumption) thanks to a multiplication of the number of cores. But it also means more power consumption. During recent years power optimization has become one of the hottest design topics not only for battery-powered devices but also for large variety of application domains such as household electronic to high performance computing. The ITRS [21] predicts an increase by a factor of 2 for the next five years in the power consumption of stationary consumer devices (see Fig. 1.4). Moreover, it is predicted that leakage and dynamic power consumption will be equivalent for such devices for both logic and memory parts. These trends, combined with the increasing performance demand, turn the problem into a real challenge for MPSoC architects [5, 4]. How can we manage the power/performance trade-off on multi-million transistor designs? It is

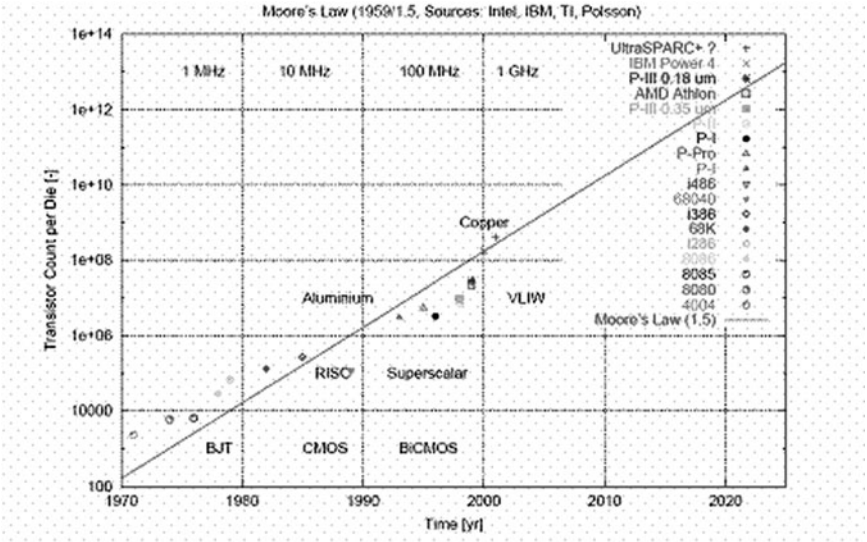


Fig. 1.3 CPU transistors count (http://www.ausairpower.net/moore-iw.pdf)

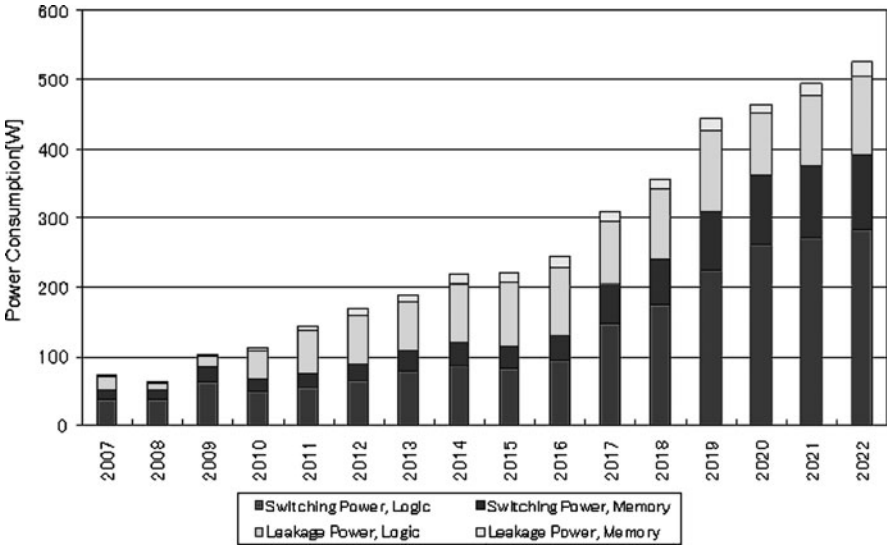


Fig. 1.4 SoC consumer stationary power consumption trends [21]

admitted that advanced energy management is mandatory to achieve efficiency, not only for mobile devices but also for all kind of electronic equipments.

While MPSoC should be designed to be power efficient, the operating environment can no more be considered as static. Let's take a simple example to understand the concept considering fourth generation of telecommunication applications. Computation-intensive complex channel estimation algorithms are needed to sustain a high throughput with bad quality transmission channels. Anyway, when the mobile terminal goes near to a base station a simpler scheme can be used to save energy. How can we manage these modifications in the environment?

A second example of the environmental conditions considers technological variability. Moore's law predicting more and more transistors with improved performance also carries variability problems. Variability is a phenomenon, which always existed in the manufacturing process of CMOS transistors and has been historically taken into account with design margins using statistics of discrepancy between chips of the wafer. However, as transistor size shrinks this phenomenon increases, coping with variability has become a real challenge: the dispersion of parameters within the same chip has now an unquestionable impact on system operation. MPSoCs are affected by this phenomenon. For example, not all PEs of the same system are able to run at the same clock frequency. As a consequence, two specimens of the same MPSoC often achieve unequal performance levels. Hence, how can designers guarantee the performance management under such variations in the manufacturing process?

In order to improve power efficiency in dynamic environments under variability, the answer can be self-adaptability. In other words, the solution can be a system able to adjust itself according to changes in its environment or in parts of the system itself in order to fulfill the requirements.

1.4 Complexity and Scalability

As stated in the introduction, the advances predicted by Moore's law have also accelerated the complexity by multiplying the number of processing elements. To illustrate this increasing complexity, Fig. 1.5 shows the trends predicted by the ITRS [21]: the number of processing cores in SoC consumer portable equipments will increase by a factor of about 3.5 times in the next five years. Moreover, the memory size and logic size will follow the same trends. In this context, how will we manage the more than six hundred processors predicted in 10 years?

There is an underlying problem to the complexity wall: the scalability. Scalability is a property of a system, which indicates its ability to be scaled uo to larger realizations. For MPSoCs, it refers to the capability of a system to increase the total computational power when resources are added. A system, whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system. An algorithm, design, networking protocol, program, or other system is said to scale if it is suitably efficient and practical when applied to large situations.

The good solution for today is probably not the good solution for tomorrow: platform based design and core reuse have driven industrial system designers for obvious productivity and performance reasons. Thes design techniques are increasigly questioned and may not scale any further. One major drawback is that these solutions are poorly scalable in terms of software and hardware. We strongly believe that an alternative is possible from a basis of a scalable hardware and software framework. For this, the distribution of the management functions of an MPSoC is crucial.

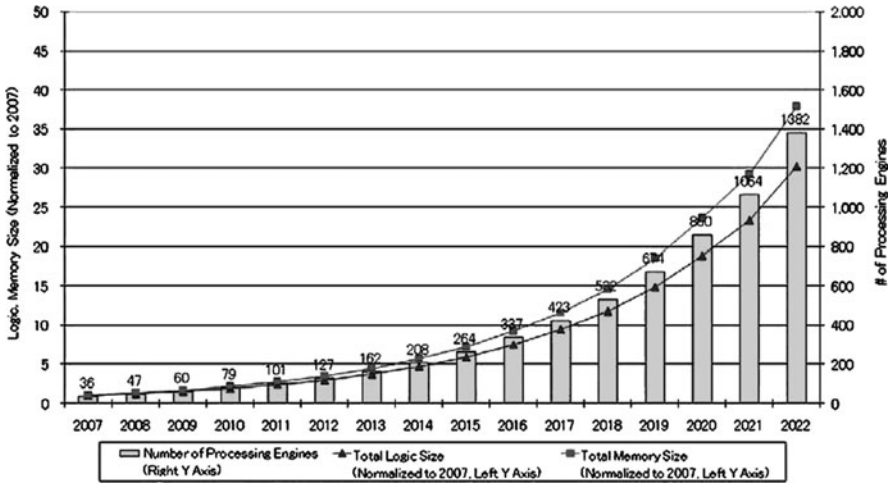


Fig. 1.5 SoC consumer portable design complexity trends [21]

1.5 Heterogeneous and Homogeneous Approaches

In the context of scalability requirement associated with self-adaptability need, MPSoCs are becoming an increasingly popular solution that combines flexibility of software along with potentially significant speedups. As stated in the introduction section, we will make a difference between:

- *Heterogeneous MPSoC*, also referred to Chip Multi-Processing or Multi (Many) Core Systems: these systems are composed of PEs of different types, such as one or several general purpose processors, Digital Signal Processors (DSPs), hardware accelerators, peripherals and an interconnection infrastructure like a NoC.
- *Homogeneous MPSoC*, in this approach, the basic PE embeds all the elements required for a SoC: one or several processors (general purpose or dedicated), memory and peripherals. This tile is then instantiated several times, and all these instances are interconnected through a dedicated communication infrastructure.

Basically, the first approach offers the best performance on power consumption trade-off and the second one is obviously more flexible and scalable but less power efficient. Due to their good power efficicency, heterogeneous MPSoC approaches are used for portable systems, and more generally embedded systems, while homogeneous approaches are commonly used for video game consoles, desktop computers, servers and supercomputing.

1.5.1 Heterogeneous MPSoC

A heterogeneous MPSoC is a set of interconnected cores with different functionalities. The Fig. 1.6 provides an overview of a generic heterogeneous MPSoC, composed of a set of general-purpose processor (CPU), several accelerators (video, audio, etc.), memory elements, peripherals and an interconnection infrastructure.

Beyond its hardware architecture, an MPSoC system is generally running a set of software applications divided into tasks and an operating system devoted to manage

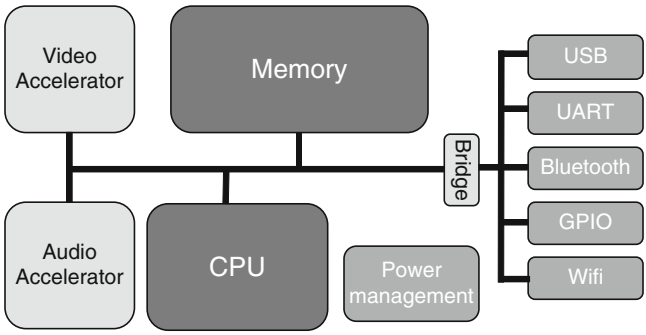


Fig. 1.6 Simplified overview of a heterogeneous MPSoC

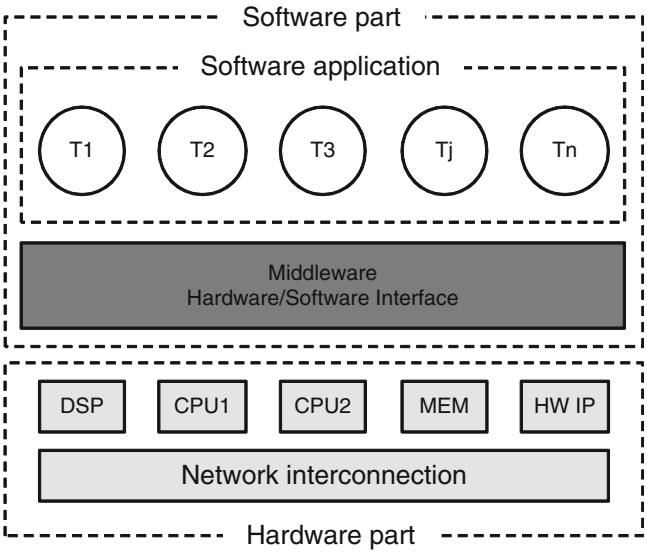


Fig. 1.7 MPSoC abstract view

both hardware and software through a middleware layer (*e.g.* drivers). Figure 1.7 illustrates an abstract view of an MPSoC, and the interaction between software and hardware.

In order to illustrate the general principles presented in the previous section, we can cite The Philips Nexperia, or ST Nomadik or the well-known TI OMAP Platform, or the MORPHEUS MPSoC [22] from the MORPHEUS European project. The functional and structural heterogeneity of these platforms permits obtaining good performance and energy efficiency, allowing them to be integrated in portable devices such as mobile phones.

The term "platform" also confers some flexibility to this approach. Indeed, it is possible with the same platform to customize the system for some specific applications thanks to a basic processor-memory-bus infrastructure, and a library of optional accelerators and peripherals. This approach allows reducing NRE costs and the Time-to-Market, but also presents some drawbacks. The flexibility is limited to the design phase or to some minor extent after fabrication since dedicated accelerators functionalities cannot be reconfigured. The scalability is also a problem of such platforms, since required communication bandwidth depends on the number and types of accelerators and thus can require some adaptation for each design.

1.5.2 Homogeneous MPSoC

As discussed in the previous section, heterogeneous MPSoC systems provide today the best performances/power efficiency tradeoffs and are natural choice for embedded systems, but they also suffer from limited flexibility and scalability.

An alternative lies in building a homogeneous system based on the same programmable building block instantiated several times. This architectural model is often referred in the literature to as parallel architecture model. Parallel architectures were particularly studied in Computer Science and Computer Engineering during the past 40 years. There is nowadays a growing interest for such approaches in embedded systems. The basic principle of an architecture that exhibits parallel processing capabilities relies on increasing the number of physical resources in order to divide the execution time of each resource. Theoretically, an architecture made of N processing resources may provide a speedup of at most N; however this speedup is difficult (or impossible) to obtain in practice. Another benefit of using multiple processing elements versus a single one is that this allows decreasing the frequency correspondingly; and therefore the power supply voltage: as the consumed power is bound to voltage to the power of 2, this decreases the dynamic power consumption significantly. The dynamic power consumption is: $P_{dyn} = \alpha.C_{LOAD}.VDD^2.F_{CLK}$ with P_{dyn} the dynamic power consumption, α is the activity factor, i.e., the fraction of the circuit that is switching, C_{load} the circuit equivalent capacitance, VDD the supply voltage, and f_{clock} the clock frequency. Assuming that it is possible to reduce the clock frequency by a factor r (with $0 < r < 1$), it is then possible to reduce the supply factor by the same factor thanks to DVFS techniques (Dynamic Voltage and Frequency Scaling). Finally, the dynamic power consumption is: $P_{dyn} = \alpha.C_{LOAD}.(r.VDD)^2.(r.F_{CLK})$; with $r = 0{,}8$, the dynamic power is almost divided by 2.

A homogeneous MPSoC based on programmable parallel processors could provide performance thanks to the "speed-up" and a reduced power-consumption by decreasing the operating frequency and the power supply and could be considered as a real alternative to heterogeneous MPSoC. Moreover, their inherent structure is more flexible and more scalable than heterogeneous systems. Practically, exploiting efficiently the parallelism is not straightforward; flexibility and scalability could also be limited due to several factors such as the organization of the memory, the interconnection infrastructure, etc.

Parallel architectures have been studied intensively during the past 40 years; there is consequently a huge amount of books and references related to this topic and we will therefore only focus on general concepts.

The first famous classification was proposed by Flynn [23]. He classifies architectures according to the relationship between processing units and control units. He defines four execution models: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) and MIMD (Multiple Instruction Multiple Data). The SISD model is the classical Von Neumann model [24], where a single processing resource executing a single instruction per unit time processes a single data flow. In SIMD architecture, a single control unit shares the data flows and distributes data to each processing resource. The MISD architectures execute several instructions simultaneously on a single data flow. Finally, several control units manage several processing units in the MIMD architectures.

In Fig. 1.8, Flynn's classification has been extended to take into account the organization of the memory that can be shared (8.a) or distributed (8.b). In shared memory architecture, processes (executed by different processors) can easily exchange information through shared variables; however it requires handling

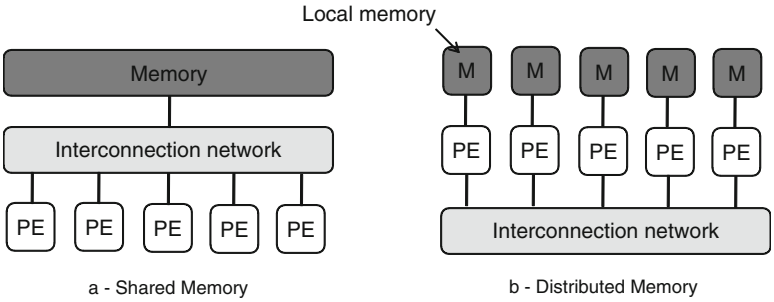


Fig. 1.8 Memory organization

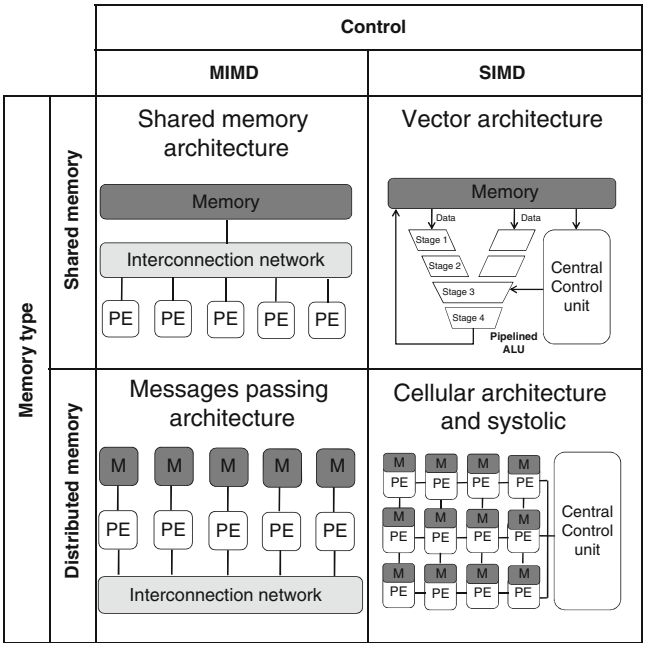


Fig. 1.9 Architecture taxonomy

carefully synchronization and memory protection. In distributed memory architecture, a communication infrastructure is required in order to connect processing elements and their memories and allow exchanging information.

Based on the memory and the control organization, the Fig. 1.9 depicts an architecture classification of parallel homogeneous processing architectures. It distinguishes the centralized control (SIMD) and decentralized control (MIMD), shared and distributed memories.

It is important to observe here that the organization of the control and the memory will provide different trade-offs in terms of scalability and management of the system. For instance, an architecture based on a fully distributed control and

memory organization, will be more scalable but less flexible to manage, than architecture based on a centralized control and a shared memory.

1.6 Multi variable Optimization

Due to the increasing complexity of MPSoC architectures, optimization is a real challenge since it may target multiple opposite objectives: application performance, power consumption/energy, temperature, load balacing, etc. In the literature, there are several methods developed to address this problem. Classical approaches are static and try to optimize the system at design time. More recent techniques are employed at run-time and try to adapt the system dynamically. Most advanced methods aim at taking advantage of the distributed decision capabilities of the processing elements in order to improve the scalability of the system.

1.6.1 Static Optimization

In the context of MPSoC, a static optimization approach is a way to improve the system at design time. Several authors have proposed static optimization techniques to improve the power efficiency. For example, in [25] authors use genetic algorithms to solve the optimization problem at design time. They explore metrics including communication traffic, memory occupation and throughput aspects.

In [26], authors analyze three static optimization methods: greedy algorithm, tabu search and simulated annealing. The problem of how to find a task schedule with the minimum power consumption while satisfying some timing constraints is studied as a part of the design space exploration process. Firstly, the system description is decomposed into Synchronous Data Flow (SDF) graphs [27] in a single frequency domain including timing constraints. Then, an extension of traditional SDFs to multi-frequency domain graphs is proposed.

In [28], a static policy based on linear models is proposed to optimize the power consumption while guaranteeing real-time constraints. The problem of selecting the best operating frequency for each block of a distributed design is studied. The author models a set of frame-based pipelined applications by using SDF graphs. Then, applications are mapped on a distributed platform integrating fine-grain DVFS.

1.6.2 Dynamic Optimization

Static explorations are always necessary to make design-time decisions. Nevertheless, considering the increasing uncertainty of implementation technologies and

applicative scenarios of such systems, dynamic optimizations are becoming mandatory to provide flexible approaches and reliable designs [29]. Centralized and distributed approaches are reported in the following subsections.

1.6.2.1 Centralized Approaches

Contrarily to static optimization, dynamic approaches offer adaptability. Figure 1.10 shows a schematic view representing the common dynamic approaches existing for MPSoC: a centralized optimization subsystem is in charge of the whole system management. It analyzes global information and optimizes each processing element in the system.

In [30], the frequency and voltage selection for GALS systems based on VFIs is addressed. A centralized method based on non-linear Lagrange optimization is used to select the frequencies and voltages. They present static and dynamic algorithms. Moreover, authors affirm that ideally in latency-constrained systems, the assignment of optimal voltages would need a global strategy decision.

Similarly, in [31, 32], authors propose a centralized energy management using models inspired by Kirchhoff's current law. They argue that while local energy dissipation of each PE can be minimized using DVS techniques based on workload predictions, it can be shown that these local minimums usually do not represent the

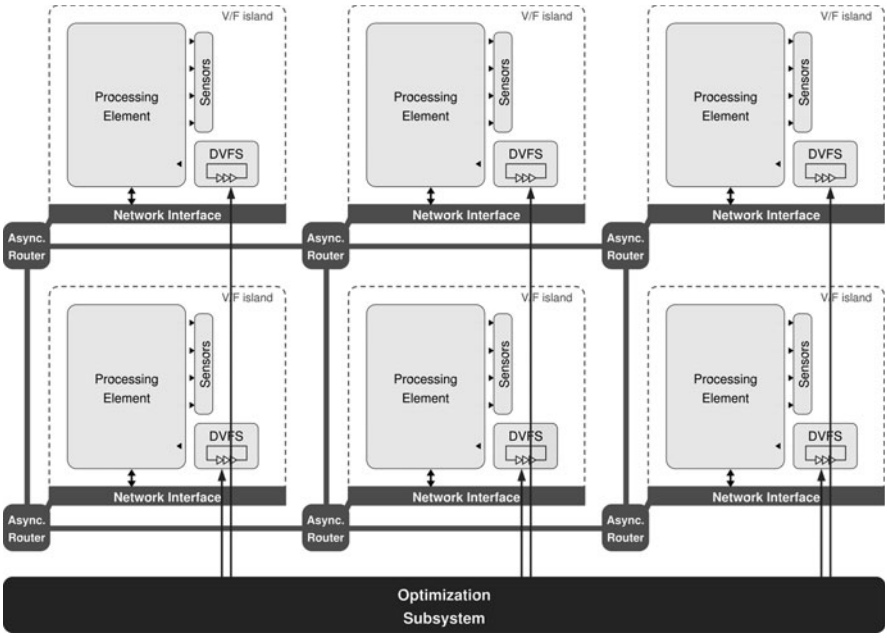


Fig. 1.10 Centralized dynamic optimization on MPSoC

global optimum. Moreover, the global optimum can be reached by considering the relative timing dependencies of all tasks running in the system. Their approach is based on an online global energy management unit that controls the PEs through a power source and a clock generator. The block diagram of such approach is represented in Fig. 1.11. Authors exploit the analogy between the energy minimization problem under timing constraints in a general task graph and the power minimization problem under Kirchhoff's current law constraints in an equivalent resistive network.

In [33], authors use convex optimization for temperature-aware frequency assignment on MPSoC. Firstly, they present a complex temperature model. Then, the problem is formulated for both, steady-state and dynamic-state: assign a single frequency to each processor, maintaining the temperature and power consumption below user-defined thresholds. In steady state, frequency and voltage are assigned once and remained constant, without taking advantage of DVFS. In dynamic state, the frequencies and voltages are varied over time to better optimize the system performance.

Authors formulate both scenarios as convex optimization problems. Then, they propose the steady state and dynamic-state optimization procedures. For the dynamic case, a 2-phase algorithm is used. Nevertheless, authors only present the mathematic formulation. Using a Matlab convex-optimization solver solves the demonstrative scenario shown in this work. The same authors propose in [34] to pre-calculate some valid solution at design time by using the convex-optimization method, and to implement a control to choose at run-time the best solution for each case. Figure 1.12 shows the flow for the design-time table construction.

In [35], authors survey some studies for energy-efficient scheduling in real-time systems on platforms integrating DVFS. After a long review of techniques applied to single-processor systems, the article divides multiprocessor platform into homogeneous and heterogeneous ones. For the first one, it briefly describes some techniques applied to frame-based real-time task scheduling, periodic real-time

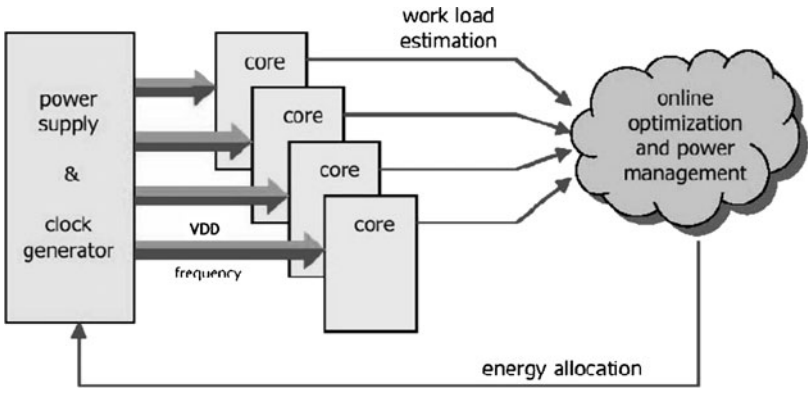


Fig. 1.11 Centralized online global energy management [32]

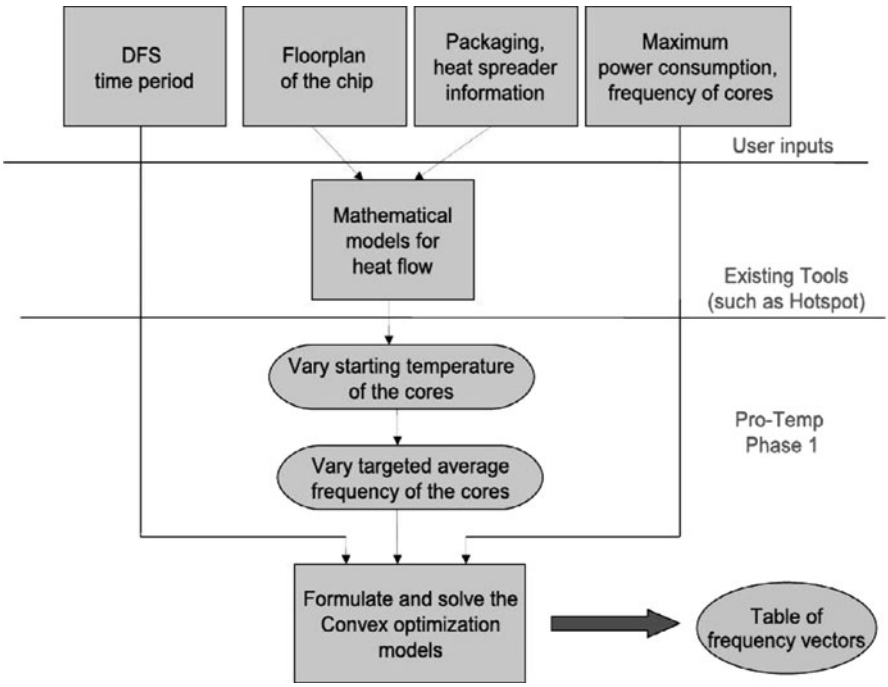


Fig. 1.12 Design-time table construction by convex optimization [34]

scheduling, leakage-aware energy-efficient scheduling and slack reclamation scheduling. For heterogeneous systems, it presents some techniques for periodic real-time tasks and allocation cost minimization under energy constraints.

In [36], heuristics for optimal task mapping are discussed in a NoC-based heterogeneous MPSoC. There are several approaches where tasks are moved in order to balance computation workload and homogeneously dissipate the power, for example in [37]. In that sense, in [38], hot-spot avoidance and thermal gradient control become an important optimization problem regarding the reliability of MPSoC.

In [38], authors investigate dynamic OS-based schedulers for thermal management of MPSoCs. In [39], thermal gradients are also minimized. They focus on MPSoCs where workload is not known a priori and generally not easy to predict. They propose an OS-based task migration and scheduling policy that optimizes the thermal profile of the chip by balancing the system load. Authors claim to obtain significant reductions in temporal and spatial temperature variations.

In [40, 41], authors propose a design time Pareto exploration and characterization combined with run-time management. They firstly make a multi-dimensional design-time exploration. The space includes costs (e.g. energy consumption), constraints (e.g. performance) and used platform resources (e.g. memory usage, processors, clocks, communication bandwidth). A low-complexity run-time manager implemented in the OS takes critical decisions during a second phase.

In [42], the interest is how to select at run-time an energy-efficient mapping on heterogeneous multi-processor platforms. Considering that many different possible implementations per application can be available and the selection must meet the application deadlines under the available platform resources, authors model as a NP-hard problem: the Multi-dimension Multi-choice Knapsack Problem. In order to find a near-optimal solution, they propose a heuristic-based OS implementation.

1.6.2.2 Distributed Approaches

With forecasted hundreds of processing elements (PE), scalability is also a major concern for the optimization process. For this reason, an alternative approach is to handle optimization dynamically in a distributed way. Static optimization does not provide adatability at run-time. On the contrary, existing dynamic approaches provide reactivity during execution time but they are centralized solutions. They do not provide scalability since they are not based on distributed models

An alternative solution to centralized approaches is to consider distributed algorithms. One interesting approach is to conceive the architecture illustrated in Fig. 1.13: each processing element of an MPSoC embeds an optimization subsystem based on a distributed algorithm. This subsystem manages the local actuators (DVFS in the figure) taking into account the operating conditions. In other words, the goal is to conceive a distributed and dynamic optimization algorithm.

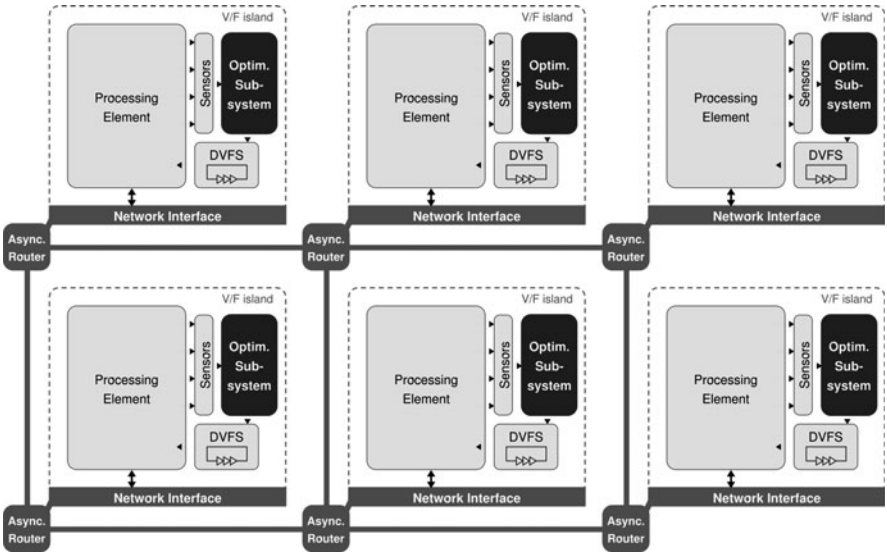


Fig. 1.13 Distributed dynamic optimization on MPSoC

To avoid hotspots and control the temperature of the tiles, dynamic voltage-frequency scaling (DVFS) can be applied at PE level. At system level, it implies to dynamically manage the different voltage-frequency couples of each PE in order to obtain a global optimization. In [43], an original approach based on game theory is presented, which adjusts at run-time the frequency of each PE. It aims at reducing the tile temperature while maintaining the synchronization between the tasks of the application graph. A fully distributed scheme is assumed in order to build a scalable mechanism. Results show that the proposed run-time algorithm find solutions in few calculation cycles achieving temperature reductions of about 23%. In [44], results show that the proposed run-time algorithm requires an average of 20 calculation cycles to find the solution for a 100-processor platform and reaches equivalent performances when comparing with an offline method.

In [45], this adaptive technique is applied to reduce power consumption. It optimizes the frequencies of local processors while fulfilling applicative real-time constraints. The obtained power consumption gains on a telecommunication test case are between 10% and 25%, while the reaction time to temporal variations due to application reconfiguration is less than 25µs.

1.7 Static vs Dynamic Centralized and Distributed Approaches

Table 1.1 summarizes the optimization methods described in the previous section. Several approaches have been considered, representing the directions of MPSoC optimizations. This table allows a qualitative comparison. The methods are compared regarding the off-line and dynamic phases, their complexities, and their implementations (centralized or distributed). Approaches presented in [34, 33] and [30] have complex off-line optimization phases. In [40, 41, 42], most of the work is done at design time. These approaches can be hardly used at run-time due to their high complexity.

The solution proposed in [30, 32] operates with a dynamic optimization subsystem with a low complexity. In the same direction, approaches [38, 39] and [40, 41, 42] provide run-time management. Nevertheless, all these approaches fail when distributed aspects are considered. When OS-based implementation is used ([38, 39] and [41, 42, 43]) a distributed implementation can be imagined. However, we do not consider doing that since they are not based on distributed models. The approach based on Game Theory [44, 45] is intrinsically based on a distributed model, which improves the scalability of the system. Furthermore, this low complexity method can be easily implemented at run-time, which implies a good adaptability required by dynamic systems.

Finally, Table 1.1 also compares the metrics used in each case. One can note that not every approach includes constrained scenarios but all of them propose multi-objective optimization. Some of these models can be reused in novel formulations.

Table 1.1 MPSoC optimization synthesis

	EPFL	Stanford & EPFL	Carnegie Melon	SUN & U. California	IMEC	LIRMM & CEA LETI
Reference	[32, 33]	[34, 35]	[31]	[39, 40]	[41, 42, 43]	[43, 44, 45]
Model	Kirchoff's current law analogy	Convex Optimization	Non-linear Lagrange Optimization	Integer Linear Programming	Off-line exploration	Game Theory
Offline phase	Yes	Yes	Yes	No	Yes	No
Complexity	Medium	High	High	–	Very High	–
Dynamic Phase	Yes	–	–	Yes	Yes	Yes
Complexity	Low	–	–	Medium	Medium	Low
Distributed Imp.	No	No	No	Possible	Possible	Yes
Distrib. model	No	No	No	No	No	Yes
Objective metrics	Energy	Performance	Energy, Throughput	Hot-spot, Temp. gradient	Multiple	Mulitple
Constraints	Latency	Temperature	Latency	–	–	Latency, Energy
Actuator	Global power manager	DVFS	Vdd & freq.	OS-based task migration scheduling	OS-based	Local DVFS

1.8 Conclusion

Semiconductors currently undergo profound changes due to several factors such as the approaching limits of silicon CMOS technology as well as the inadequacy of the machine models that have been used until now. These challenges require devising new design approaches and programming of future integrated circuits. Hence, parallelism appears as the only solution for coping with the ever-increasing demand in term of performance. The solutions that are suggested in the literature often rely on the capability of the system to take online decisions for coping with these issues, such as scaling supply voltage and frequency for increasing energy efficiency, or testing the circuit for identifying faulty components and discarding them from the functionality.

MPSoCs are certainly the natural target for bringing these techniques into practice: provided they comply with some design rules they may prove scalable from a performance point of view. Further, since they are in essence distributed architectures they are well suited to locally monitoring and controlling system parameters.

In this chapter, we have studied multiprocessor systems and proposed an overview of a template that we believe is representative of tomorrows' MPSoCs. The important characteristics that have been considered are mostly flexibility, scalability and adaptability, considering decentralized control, Homogeneous or Heterogeneous array of processing elements, Distributed memory, Scalable NoC-style communication network.

Finally, we think that adaptability is an approach that will in the near future be widely adopted in the area. Not only because of the here mentioned limitations such as technology shrinking, power consumption and reliability but also because computing undoubtedly go pervasive. Pervasive, or ambient computing is a research area on its own and in essence implies using architecture that are capable of self-adapting to many time-changing execution scenarios. Be it mobile sensors deployed for monitoring various natural phenomena or computing devices embedded in clothes (wearable computing), such systems have to cope with many limitations such as limited power budget, interoperability, communication issues, and finally, scalability.

GLOSSARY

SoC	System On Chip
MPSoC	Multiprocessor System On Chip
NoC	Network On Chip
HPC	High Performance Computing
ASIC	Application Specific Integrated Circuit
PE	Processing Element
SW	Software

HW	Hardware
OMAP	Open Multimedia Applications Platform
DVFS	Dynamic Voltage and Frequency Scaling
GALS	Globally Asynchronous Locally Synchronous
NI	Network Interfaces
VFI	Voltage/Frequency Islands
ITRS	International Technology Roadmap for Semiconductors
CPU	Central Processing Unit
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
OS	Operating System

References

1. Ahmed Jerraya and Wayne Wolf. Multiprocessor Systems-on-Chips. Elsevier Inc, 2004.
2. Wayne Wolf, Ahmed Jerraya, and Grant Martin. Multiprocessor system-on-chip (MPSoC) technology. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 27(10):1701–1713, Oct. 2008.
3. Wayne Wolf. The future of multiprocessor systems-on-chips. In DAC '04: Proceedings of the 41st annual Design Automation Conference, pages 681–685, New York, NY, USA, 2004. ACM.
4. Freescale Semiconductor, Inc. C-5 Network Processor Architecture Guide, 2001. Ref. manual C5NPD0-AG http://www.freescale.com.
5. S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. Design & Test of Computers, IEEE, 18(5):21– 31, Sep-Oct 2001.
6. Texas Instruments Inc. OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide, 2006. Tech. article SPRU748C. http://www.ti.com.
7. B. Ackland, A. Anesko, D. Brinthaupt, S.J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C.J. Nicol, J.H. O'Neill, J. Othmer, E. Sackinger, K.J. Singh, J. Sweet, C.J. Terman, and J. Williams. A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP. Solid-State Circuits, IEEE Journal of, 35(3):412–424, Mar 2000.
8. P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In DATE '00: Proceedings of the 2000 Design, Automation and Test in Europe Conference and Exhibition, pages 250–256, 2000.
9. William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In DAC '01: Proceedings of the 38th Design Automation Conference, pages 684–689, New York, NY, USA, 2001. ACM.
10. L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. IEEE Computer, 35 (1):70–78, Jan 2002. [cited at p. 3]
11. Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of Network-on-chip. ACM Comput. Surv., 38(1):1, 2006.
12. Partha Pratim Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Perfor- mance evaluation and design trade-offs for network-on-chip interconnect architec- tures. Computers, IEEE Transactions on, 54(8):1025–1040, Aug. 2005.

13. D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigas- cale systems-on-chip. Circuits and Systems Magazine, IEEE, 4(2):18–31, 2004.
14. E. Beigńe, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. Asynchronous NOC Architecture Providing Low Latency Service and Its Multi-Level Design Framework. In ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems, pages 54–63, Washington, DC, USA, 2005. IEEE Computer Society.
15. J. Pontes, M. Moreira, R. Soares, and N. Calazans. Hermes-glp: A gals network on chip router with power control techniques. In Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual, pages 347–352, April 2008.
16. Umit Y. Ogras, Radu Marculescu, Puru Choudhary, and Diana Marculescu.Voltage- frequency island partitioning for GALS-based Networks-on-Chip. In DAC '07: Proceedings of the 44th Annual Design Automation Conference, pages 110–115, New York, NY, USA, 2007. ACM.
17. James Donald and Margaret Martonosi. Techniques for multicore thermal man- agement: Classification and new exploration. In ISCA '06: Proceeding of the 33rd International Symposium on Computer Architecture, pages 78–88, 2006.
18. Edith Beigńe, Fabien Clermidy, Sylvain Miermont, and Pascal Vivet. Dynamic voltage and frequency scaling architecture for units integration within a gals noc. In NOCS, pages 129–138, 2008.
19. Edith Beigńe, Fabien Clermidy, Sylvain Miermont, Alexandre Valentian, Pascal Vivet, S Barasinski, F Blisson, N Kohli, and S Kumar. A fully integrated power supply unit for fine grain dvfs and leakage control validated on low-voltage srams. In ESSCIRC'08: Proceeding of the 34th European Solid-State Circuits Conference, Edinburg, UK, Sept. 2008.
20. G. E. Moore. Cramming More Components onto Integrated Circuits. Electronics, 38 (8):114–117, April 1965.
21. The International Technology Roadmap for Semiconductors. International Technology Roadmap for Semiconductors 2008 Update Overview. http://www.itrs.net.
22. Davide Rossi, Fabio Campi, Antonello Deledda, Simone Spolzino and Stefano Pucillo, A Heterogeneous Digital Signal Processor Implementation for Dynamically Reconfigurable Computing, IEEE Custom Integrated Circuits Conference (CICC) September 13 - 16 2009,
23. M. Flynn. Some Computer Organizations and Their Effectiveness, IEEE Trans. Computer, vol. 21, pp. 948, 1972
24. A. W. Burks, H. Goldstine, and J. von Neumann. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Inst. Advanced Study Rept., vol. 1, June, 1946
25. Issam Maalej, Guy Gogniat, Jean Luc Philippe, and Mohamed Abid. System Level Design Space Exploration for Multiprocessor System on Chip. In ISVLSI '08: Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI, pages 93–98, Washington, DC, USA, 2008. IEEE Computer Society.
26. Bastian Knerr, Martin Holzer, and Markus Rupp. Task Scheduling for Power Opti- misation of Multi Frequency synchronous Data Flow Graphs. In SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design, pages 50–55, New York, NY, USA, 2005. ACM.
27. Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput., 36(1):24–35, 1987.
28. Philippe Grosse, Yves Durand, Paul Feautrier: Methods for power optimization in SOC-based data flow systems. ACM Trans. Design Autom. Electr. Syst. 14(3): (2009)
29. A. K. Coskun, T. Simunic Rosing, K. Mihic, G. De Micheli, and Y. Leblebici. Analysis and Optimization of MPSoC Reliability. Journal of Low Power Electronics, 2(1):56–69, 2006.
30. Koushik Niyogi and Diana Marculescu. Speed and voltage selection for GALS sys- tems based on voltage/frequency islands. In ASP-DAC '05: Proceedings of the 2005 Conference on Asia South Pacific Design Automation, pages 292–297, New York, NY, USA, 2005. ACM.
31. Zeynep Toprak Deniz, Yusuf Leblebici, and Eric Vittoz. Configurable On-Line Global Energy Optimization in Multi-Core Embedded Systems Using Principles of Analog Computation. In IFIP 2006: International Conference on Very Large Scale Integration, pages 379–384, Oct. 2006.

32. Zeynep Toprak Deniz, Yusuf Leblebici, and Eric Vittoz. On-Line Global Energy Optimization in Multi-Core Systems Using Principles of Analog Computation. In ESSCIRC 2006: Proceedings of the 32nd European Solid-State Circuits Conference, pages 219–222, Sept. 2006.
33. Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen Boyd, and Giovanni De Micheli. Temperature-aware processor frequency assignment for MPSoCs using convex optimization. In CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, pages 111–116, New York, NY, USA, 2007. ACM.
34. Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen Boyd, Luca Benini, and Giovanni De Micheli. Temperature control of high-performance multi-core platforms using convex optimization. In DATE'08: Design, Automation and Test in Europe, pages 110–115, Munich, Germany, 2008. IEEE Computer Society.
35. Jian-Jia Chenand Chin-Fu Kuo. Energy-Efficient Schedulingfor Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms. In RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pages 28–38, Washington, DC, USA, 2007. IEEE Computer Society.
36. Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Heuristics for dynamic task mapping in noc-based heterogeneous MPSoCs. In RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping, pages 34–40, Washington, DC, USA, 2007. IEEE Computer Society.
37. G. M. Link and N. Vijaykrishnan. Hotspot prevention through runtime reconfiguration in Network-on-Chip. In DATE '05: Proceedings of the 2005 Conference on Design, Automation and Test in Europe, pages 648–649, Washington, DC, USA, 2005. IEEE Computer Society.
38. Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Keith Whisnant. Temperature aware task scheduling in MPSoCs. In DATE '07: Proceedings of the conference on Design, automation and test in Europe, pages 1659–1664, San Jose, CA, USA, 2007. EDA Consortium.
39. Ayse Kivilcim Coskun, Tajana Simunic Rosing, Keith A. Whisnant, and Kenny C. Gross. Temperature-aware mpsoc scheduling for reducing hot spots and gradients. In ASP-DAC '08: Proceedings of the 2008 conference on Asia and South Pacific design automation, pages 49–54, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.
40. Ch. Ykman-Couvreur, E. Brockmeyer, V. Nollet, Th. Marescaux, Fr. Catthoor, and H. Corporaal. Design-Time Application Exploration for MP-SoC Customized Run- Time Management. In SOC'05: Proceedings of the International Symposium on System-on-Chip, pages 66–73, Tampere, Finland, November 2005.
41. Ch. Ykman-Couvreur, V. Nollet, Fr. Catthoor, and H. Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In SOC'06: Proceedings of the International Symposium on System-on-Chip, pages 195–198, Tampere, Finland, November 2006.
42. Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor, and H. Corporaal. Pareto-based application specification for MP-SoC Customized Run-Time Management. In SAMOS'06: Proceedings of the International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation, pages 78–84, Samos, Greece, July 2006.
43. D. Puschini, F. Clermidy, P. Benoit, G. Sassatelli, and L. Torres. Temperature-Aware Distributed Run-Time Optimization on MP-SoC Using Game Theory, Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual, 2008, pp. 375-380.
44. D. Puschini, F. Clermidy, P. Benoit, and G. Sassatelli. A Game-Theoretic Approach for Run-Time Distributed Optimization on MP-SoC, International Journal of Reconfigurable Computing, vol. 2008, 2008, p. 11.
45. D. Puschini, F. Clermidy, P. Benoit, G. Sassatelli, and L. Torres. Adaptive energy-aware latency-constrained DVFS policy for MPSoC, 2009 IEEE International SOC Conference (SOCC), IEEE, 2009, pp. 89–92.

Part I
“Application Mapping and Communication Infrastructure”

Chapter 2
Composability and Predictability for Independent Application Development, Verification, and Execution

Benny Akesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens

Abstract System-on-chip (SOC) design gets increasingly complex, as a growing number of applications are integrated in modern systems. Some of these applications have real-time requirements, such as a minimum throughput or a maximum latency. To reduce cost, system resources are shared between applications, making their timing behavior inter-dependent. Real-time requirements must hence be verified for *all possible combinations* of concurrently executing applications, which is not feasible with commonly used simulation-based techniques. This chapter addresses this problem using two complexity-reducing concepts: *composability* and *predictability*. Applications in a composable system are completely isolated and cannot affect each other's behaviors, enabling them to be independently verified. Predictable systems, on the other hand, provide lower bounds on performance, allowing applications to be verified using formal performance analysis. Five techniques to achieve composability and/or predictability in SOC resources are presented and we explain their implementation for processors, interconnect, and memories in our platform.

Keywords Composability · Predictability · Real-Time · Arbitration · Resource Management · Multi-Processor System

2.1 Introduction

The complexity of contemporary Systems-on-Chip (SOC) is increasing, as a growing number of independent applications are integrated and executed on a single chip. These applications consist of communicating tasks mapped on heterogeneous multi-processor platforms with distributed memory hierarchies that strike a good

B. Akesson (✉)
Eindhoven University of Technology, Postbus 513, 5600 MB Eindhoven, The Netherlands
e-mail: k.b.akesson@tue.nl

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_2,

balance between performance, cost, power consumption and flexibility [14, 22, 38]. The platforms exploit an increasing amount of application-level parallelism by enabling concurrent execution of more and more applications. This results in a large number of *use-cases*, which are different combinations of concurrently running applications [15]. Some applications have *real-time requirements*, such as a minimum throughput of video frames per second, or a maximum latency for processing those video frames. Applications with real-time requirements are referred to as *real-time* applications, while the rest are *non-real-time* applications. A use-case can contain an arbitrary mix of real-time and non-real-time applications.

To reduce cost, platform resources, such as processors, hardware accelerators, interconnect, and memories, are shared between applications. However, resource sharing causes *interference* between applications, making their temporal behaviors inter-dependent. Verification of real-time requirements is often performed by system-level simulation. This results in three problems with respect to verification, since inter-dependent timing behavior requires that all applications in a use-case are verified together. The first problem is that the number of use-cases *increases rapidly* with the number of applications. It hence becomes infeasible to verify the exploding number of use-cases by simulation. This forces industry to reduce coverage and verify only a subset of use-cases that have the toughest requirements [14, 37]. The second problem is that verification of a use-case cannot begin until all applications it comprises are available. Timely completion of the verification process hence depends on the availability of all applications, which may be developed by different teams inside the company, or by independent software vendors. The last problem is that use-case verification becomes a *circular process* that must be repeated if an application is added, removed, or modified [23]. Together these three problems contribute to making the integration and verification process a dominant part of SOC development, both in terms of time and money [22, 23, 34].

In this chapter, we address the real-time verification problem using two complexity-reducing concepts: *composability* and *predictability*. Applications in a composable system are completely isolated and cannot affect each other's functional or temporal behaviors. Composable systems address the verification problem in the following four ways [17]: 1) Applications can be verified in isolation, resulting in a linear and non-circular verification process. 2) Simulating only a single application and its required resources reduces simulation time compared to complete system simulations. 3) The verification process can be incremental and start as soon as the first application is available. 4) Intellectual property (IP) protection is improved, since the verification process no longer requires the IP of independent software vendors to be shared. These benefits reduce the complexity of simulation-based verification, making it a feasible option with a larger number of applications. An additional benefit is that *composability does not inherently make any assumptions on the applications*, making it applicable to existing applications without any modifications.

Predictable systems, on the other hand, bound the interference from the platform and between applications. This enables bounds on performance, such as upper bounds on latency or lower bounds on throughput, to be provided. Applications

in predictable systems can hence be verified using formal performance analysis frameworks, such as network calculus [9] or data-flow analysis [36]. The benefit of formal performance verification is that conservative performance guarantees can be provided for all possible combinations of initial states of resources and arbiters, all input stimuli, and all concurrently executing applications. The drawback is that formal approaches require performance models of the software, the hardware, and the mapping [8, 25], which are not always available. Composability and predictability both solve important parts of the verification problem and provide a complete solution when combined.

The two main contributions of this chapter are: 1) An overview of five techniques to achieve composability and/or predictability in multi-processor systems with shared resources. 2) We show how to design a composable and predictable system by applying the proposed techniques to three typical resource types: processor tiles, interconnect (a network on chip), and memory tiles (with either on-chip SRAM or off-chip SDRAM).

The rest of this chapter is organized as follows. Section 2.2 describes a number of techniques to achieve composability and/or predictability for shared resources. We then proceed in Sections 2.3, 2.4, and 2.5 by explaining which of these techniques are suitable for our processor tiles, network-on-chip, and memory tiles, respectively. Section 2.6 then demonstrates the composability of our SOC platform by showing that the behavior of an application is unaffected at the cycle-level, as other applications are added or removed. Lastly, we end the chapter with conclusions in Section 2.7.

2.2 Composability and Predictability

The introduction motivates how composability and predictability address the increasingly difficult problem of verifying real-time requirements in SOCs. The next step is to provide more details on how to implement these concepts. Firstly, we establish some essential terminology related to resource sharing, which allows us to define composability and predictability formally. We then discuss five techniques to achieve these properties and highlight their respective strengths and weaknesses. This illustrates the design space for composable and predictable systems, and allows us to explain how different techniques are suitable for different resources depending on their properties, such as whether execution times are constant or variable, and whether the resource is abundant or scarce.

2.2.1 Terminology

Our context is a tiled platform architecture following the template shown in Fig. 2.1. At the high level, this platform comprises a number of processor tiles

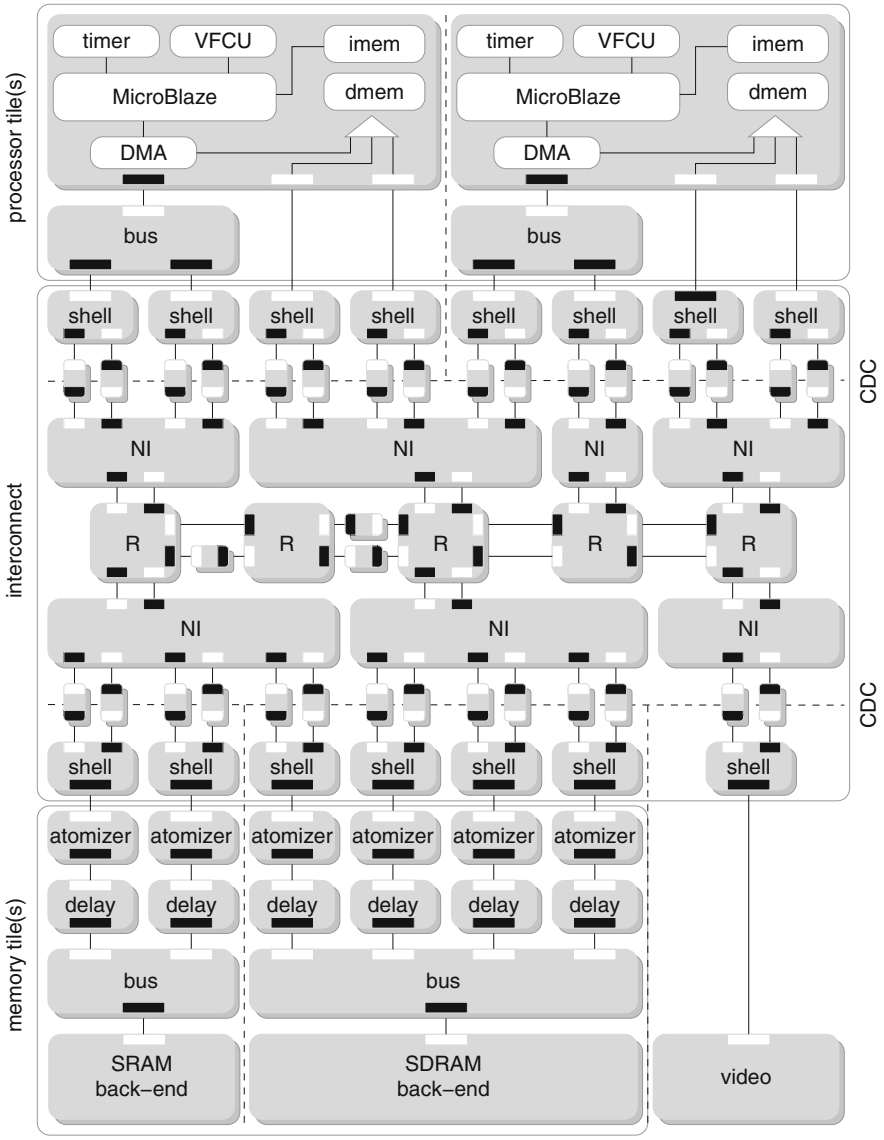


Fig. 2.1 The architecture of the considered MPSoC platform

and memory tiles interconnected by a network-on-chip. We return to discuss the details of this architecture in Sections 2.3, 2.4, and 2.5, respectively. An *application* consists of a set of *tasks* that may be split across several processor tiles to enable parallel processing. We assume a static task-to-processor mapping, which implies that task migration is not supported. Non-real-time tasks can communicate in any way they like using distributed shared memory, obeying only the restrictions

on processors, discussed later in Section 2.3.1. However, tasks of real-time applications operate in a more restrictive fashion to ensure that their temporal behavior can be bounded. Each real-time task continuously *iterates*, which means that it reads its inputs, executes its function, and writes its outputs. Inter-task communication is implemented using FIFOs, according to the C-HEAP protocol [31], with blocking read and write operations. Inside a FIFO token, data can be accessed in any order. We choose this programming model because it perfectly fits the domain of streaming applications and enables overlapping computation with communication. It furthermore allows modeling an application as a data-flow graph, which enables efficient timing analysis. Communication between processor tiles and memory tiles takes place via the interconnect.

Requests are defined as uses of a *resource*, such as a processor, interconnect, or a memory. The originators of requests, and hence the users of the resources, are referred to as *requestors*. Requests for a processor resource correspond to application tasks that are ready for execution. In case of a memory or an interconnect, requests are transactions originating from ports on IP components. These transactions are communicated using standardized protocols, such as AXI [6], DTL [33], or OCP [32]. Common examples of transactions are reads and writes of either single data words or bursts of data to a memory location.

The *execution time* (ET) of a request determines the amount of time a request uses a resource before finishing. However, a requestor may not have exclusive access to the resource, due to interference from other requestors. Interference may prevent a request from accessing the resource straight-away and its execution may be preempted several times before finishing. This is considered in the *response time* (RT) of a request, which accounts for both the execution time and the interference. The response time is hence the total time it takes from when the request is eligible for scheduling at the resource until it has been served. The point in time at which a request is scheduled to use the resource for the first time is referred to as its *starting time*. It is important to note that the execution time, response time, and starting time of a request from a requestor often depend on other requestors. The execution time may depend on others if a request from one requestor alters the state of a resource in a way that affects the execution time of a following request. A common example of this is when a memory request from a requestor evicts a cache line from another requestor, turning a future cache hit into a cache miss. The response time and starting time both typically vary with the presence or absence of requests from other requestors in systems with run-time arbitration, such as round robin or static-priority scheduling. This results in a varying interference that causes both the starting time and response time to change. We now proceed by defining composability and predictability in terms of the established terminology.

The functional behavior of a request is defined as composable when its output is independent of the behavior of requestors belonging to other applications. The temporal behavior of a request from a requestor using a resource is defined as composable if its starting time and response time are independent of requestors from other applications, since this implies that the request starts and finishes using the resource independently of others. We refer to a resource as a *composable*

resource if both functional and temporal composability holds for any set of requestors and their associated requests. A composable system contains only composable resources. Such a system enables independent verification of applications, as their constituent requestors and requests are completely isolated from each other in the time and value (functional) domains. The verification complexity hence becomes linear with respect to the number of applications. It also makes the resulting system more robust at run time, because there is no interference from unknown, failing, or misbehaving applications. In this chapter, we focus on verification of real-time requirements. We hence limit the discussion to temporal composability and do not further discuss how to achieve functional composability. For simplicity, we let composability refer to temporal composability in the rest of this chapter.

For predictability, every request on a resource must have both a *useful* worst-case execution time (WCET) and worst-case response time (WCRT). Unlike composability, which inherently considers multiple requestors and applications on a shared resource, predictability can be considered for a non-shared resource with only a single requestor. For shared resources, the WCRT can be determined if there is a bound on the interference from other requestors. A resource is a *predictable resource* if all requests from all the requestors mapped on it are predictable. Similarly, a predictable system is a system only comprising predictable resources. Predictable systems enable formal verification of real-time requirements, since applications are sets of requestors for different resources that all provide bounded WCRT. For a complete end-to-end analysis, these WCRTs have to be used in a performance analysis framework. We use data-flow [36] analysis to compute bounds on throughput and latency for real-time applications, although time-triggered [23] or network calculus [9] methods can also be used.

It is important to realize that predictability and composability are two different properties and that one does not imply the other. Predictability means that a useful bound is known on temporal behavior and is hence a property of a *single application* mapped on a set of resources. Composability, on the other hand, implies complete functional and temporal isolation between applications and is a property of *multiple applications* sharing resources, where each application may be predictable or not. We illustrate the difference by discussing four example systems, shown in Fig. 2.2, that cover all combinations of composability and predictability. The first system, depicted in Fig. 2.2a, consists of two processors (P), each executing a single application (A1 and A2, respectively). We assume that both applications are predictable and hence that worst-case execution times are known for all tasks when running on predictable hardware. Data is stored in a shared remote zero-bus-turnaround SRAM that is reached via a bus. This type of SRAM has an execution time of one clock cycle per read or written word that is independent of other requestors. The SRAM is shared using time-division multiplexing (TDM) arbitration, which is a composable and predictable arbitration scheme, since the WCRT of a requestor is both bounded and independent of other requestors. This makes this system as a whole both composable and predictable. For our second system in Fig. 2.2b, we replace the TDM arbiter with a round robin arbiter (RR). This system is not composable, since response times of requests vary depending on the presence or

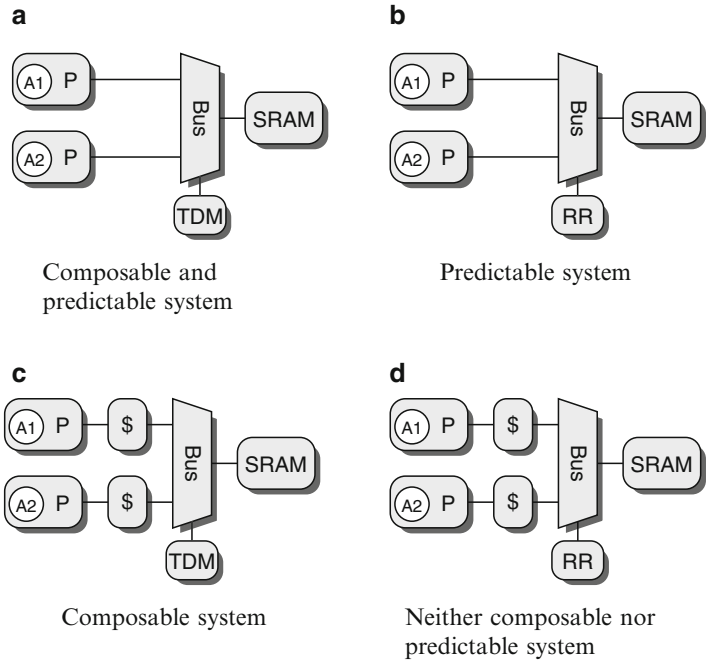
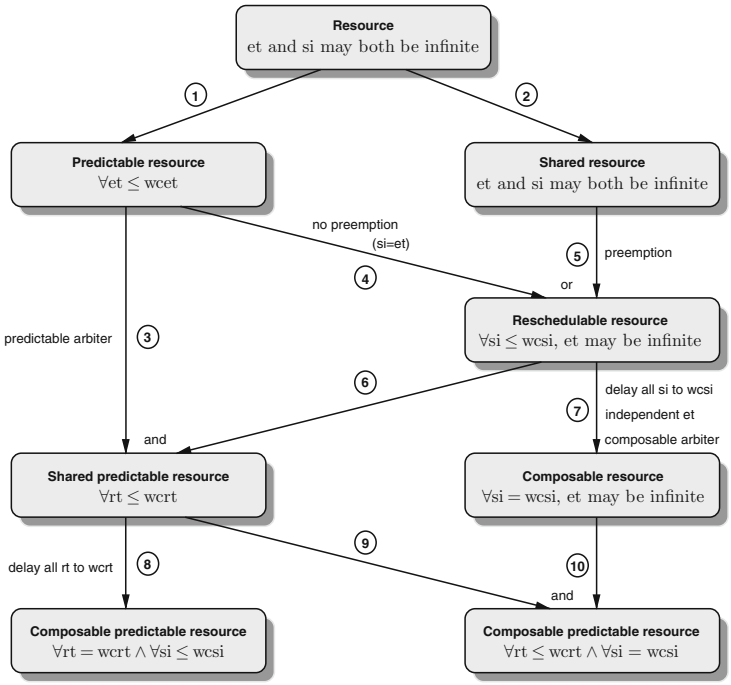


Fig. 2.2 Four systems demonstrating all combinations of the composability and predictability properties.

absence of requests from other requestors. However, it is still predictable, since this interference is easily bounded. We create our last two systems by adding private L1 caches ($) with random replacement policies to the processors in both previous systems. A private cache is composable, since it is not shared between applications. However, the random replacement policy makes the systems unpredictable, since a useful bound cannot be derived on the time to serve a sequence of requests. The third system, in Fig 2.2c, is hence composable, but not predictable. The last system, shown in Fig 2.2d, is neither composable, nor predictable.

2.2.2 Composable Resources

This section discusses designing composable resources that may or may not be predictable. As previously explained in Section 2.2.1, composability implies that the starting time and response time of a request from a requestor must be completely independent of requests from requestors belonging to other applications. Composability is trivially achieved by mapping applications to different resources, an approach used by federated architectures in the automotive and aerospace industries [24]. However, this method is prohibitively expensive for



Technique:	Path:
Composable scheduling of preemptive resources	[2, 5, 7]
Composable scheduling of non–preemptive predictable resources	[1, 4, 7]
Predictable resource scheduling	[1, 3] and ([2, 5, 6] or [1, 4, 6])
Worst–case predictable resource scheduling	[1, 3, 9] and ([2, 5, 6] or [1, 4, 6]) and [7, 10]
Predictable resource scheduling with worst–case delay	[1, 3, 8] and ([2, 5, 6] or [1, 4, 6])

Fig. 2.3 Overview of techniques to achieve composability and predictability

systems that are not safety-critical. We proceed by looking at two alternatives to composable sharing of resources. These correspond to the two paths ② → ⑤ → ⑦ and ① → ④ → ⑦ in Fig. 2.3, which provides an overview of the five techniques presented in this chapter.

The first technique is called *composable scheduling of preemptive resources* and corresponds to following the edges ②, ⑤, and ⑦. This approach considers that the execution times of requests may be variable and unknown a priori. An example of this is the time required by a video decoding task executing on a processor to decode a frame, which is highly dependent on the image contents. This results in non-composable behavior, as the starting time of a request becomes dependent on the execution time of the previous request, which may have been issued by a requestor belonging to a different application. A solution to this problem is to *preempt* an executing request after a given time, referred to as the *scheduling interval* (SI) of the resource arbiter. This is shown in Fig. 2.4a, where the request of requestor 2 is

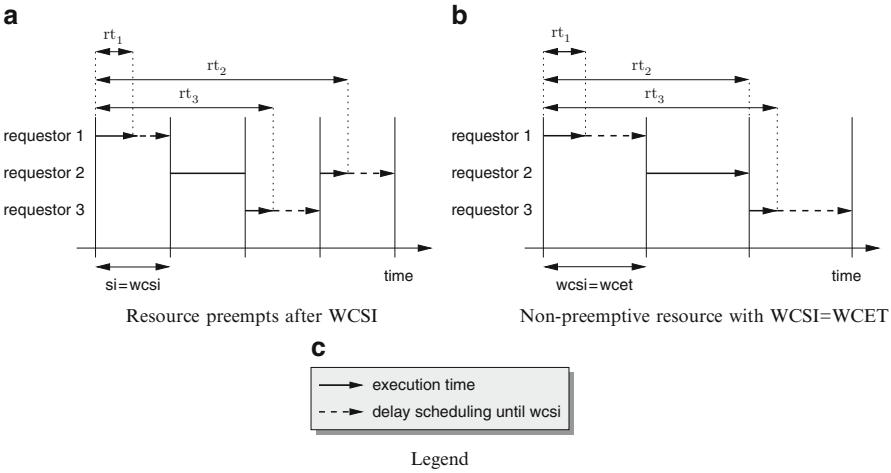


Fig. 2.4 Composable scheduling for preemptive and non-preemptive resources, respectively

preempted before finishing its execution. We refer to a resource with a worst-case scheduling interval (WCSI) as a *reschedulable resource*, as shown in Fig. 2.3, since it is guaranteed to take new scheduling decisions within a bounded time. Such a resource *ensures progress* of all requestors if it is paired with a *starvation-free* arbiter, which is a class of arbiters that guarantee that all requestors are scheduled in a finite time. Both round robin and TDM are examples of arbiters in this class. A static-priority scheduler, on the other hand, is not free of starvation, since a low-priority requestor starves if high-priority requestors are constantly requesting.

The next step with this technique is to make all scheduling intervals equal to the WCSI by delaying the arbiter in case the request finishes early, as shown in Fig. 2.4a. This step decouples the starting time of a request from the execution time of the preceding request, which is one of the two requirements to achieve composability. The second requirement is that the response time must be independent from requestors of other applications. We achieve this by using a composable arbiter, such as TDM, where the presence or absence of other requestors does not affect the interference. This results in independent response times for resources where the execution time is independent of previous requests, such as a zero-bus-turnaround SRAM. We have now fulfilled both requirements for a resource to be considered composable. Note that this type of composable resource is not necessarily predictable. It may, for example, include a cache that is private or shared between requestors belonging to the same application, which results in non-useful bounds on execution time for memory requests, although they are independent of other applications.

Next, we explore a second method of designing composable resources called *composable scheduling of non-preemptive predictable resources*, which follows the edges ①, ④, and ⑦ in Fig. 2.3. This method is motivated by the main limitation of the first approach, which is restricted to preemptive resources. Some important resources, such as SDRAM memories cannot be preempted during a burst, as they require all the data associated with a request to be transferred on consecutive clock cycles to function

correctly. Achieving composability with non-preemptive resources is still possible, assuming that the resource is predictable and hence has a known WCET. For these resources, we make the scheduling interval equal to the longest WCET of any request executing on the resource. This is illustrated in Fig. 2.4b, where the request from requestor 2 is assumed to have the longest WCET. This technique makes starting times independent of requests from other applications, which is required for composability. Supporting non-preemptive resources with bounded execution times is the major benefit of this technique. However, this method arrives at a reschedulable resource by characterizing the requests and the resource rather than by enforcement, which has three drawbacks. Firstly, it cannot be applied to mixed time-criticality systems where real-time applications share resources with non-real-time applications that do not have bounded WCET. Secondly, the system is less robust, as it becomes non-composable if the characterization is incorrect or if a requestor misbehaves. Finally, making the scheduling interval equal to the longest WCET results in low resource utilization if there is a large difference between the average and worst-case execution time. This is not acceptable for scarce resources, such as SDRAM memories.

Since composable scheduling of non-preemptive predictable resources implies that the WCET of requests have to be bounded, it may result in a system that is also predictable. This depends on whether or not the composable arbiter is also predictable. Although this is typically the case, such as for TDM, it is not inherent to composability. For example, an arbiter that randomly schedules requestors every WCSI is composable, as it is independent of applications, but it is unpredictable, since the WCRT can be infinite. We will return to discuss techniques to share resources in ways that are both composable and predictable in Section 2.2.4.

The proposed techniques for composable resource sharing make the temporal behaviors of the requestors independent of each other, thus implementing composability at the level of requestors. This is a sufficient condition to be composable at the level of applications, which is the actual requirement from Section 2.2.1. However, composability at the level of requestors is stricter in the sense that requestors belonging to the same application are allowed to interfere with each other in a composable system. It is hence possible to let requestors benefit from unused resource capacity (slack) reserved by requestors belonging to the same application to increase performance or reduce power [27]. This can be accomplished by using a two-level arbiter, as proposed in [17], where the first level is a composable inter-application arbiter, and the second an intra-application arbiter that does not have to be composable. This type of arbitration enables requestors from the same application to use slack created in the intra-application arbiter to boost performance without violating composability at the application level.

2.2.3 Predictable resources

Having discussed two ways of building resources that are composable, but not necessarily predictable, we proceed by discussing how to build resources that are

predictable, but not necessarily composable. As previously mentioned in Section 2.2.1, this requires useful bounds on both the WCET and the WCRT.

Our approach to predictable resource sharing is based on combining resources and arbiters, each with predictable behaviors. In Fig. 2.3, this intuitively corresponds to following the edges ① and ③ from a general resource to a predictable shared resource. More specifically, we require bounds on the WCET for each request executing on the resource, since these characterize the worst-case behavior of the unshared resource. Some resources, such as zero-bus-turnaround SRAMs, are predictable and have constant execution times that are easy to determine. However, other resources, such as SDRAM, have variable execution times that depend on earlier requests and cannot be usefully bounded at design time in the general case [1]. In this case, the resource controller must be implemented in a way that makes the resource behave in a predictable manner. We discuss how to accomplish this for an SDRAM resource in Section 2.5.

If the resource is shared, we require *predictable arbitration* that bounds the time within which a request finishes receiving service. Note that by this definition, *all predictable arbiters are starvation free*. Predictable arbiters enable the WCRT to be computed if the resource is reschedulable and hence makes new scheduling decisions within a bounded time, determined either by a chosen scheduling interval (preemptive resource) or by the longest WCET of any request executing on the resource (non-preemptive resource). This is illustrated in Fig. 2.3, where a predictable shared resource has to be both predictable and reschedulable and there are two possible paths to achieve the latter. Computing the WCRT takes the effects of sharing the resource into account.

An important property of our approach is that it is based on combining *independent analyses* of the resource and the arbitration. The arbiter analysis bounds the number of scheduling decisions that are made by the arbiter from a request is eligible for scheduling until it finishes receiving service. The WCRT is then conservatively computed by multiplying the number of decisions with the WCSI and adding the number of pipeline stages between the request buffer and the response buffer in the architecture. Note that this conservatively accounts for both the execution time of the request and any preemptions from other requestors during the execution. The strength of this approach is the generality, as *any combination of predictable resource and predictable arbiter* results in a predictable shared resource. This makes it easy to change the arbiter to fit with the response time requirements of the requestors in the system, which is exploited by the processor tile in Section 2.3 and the memory tile presented in Section 2.5.

2.2.4 Composable and predictable resources

Section 2.2.1 explained that composability and predictability are different properties and that one does not imply the other. We then showed in Sections 2.2.2 and 2.2.3 how to make resources that are either composable or predictable. In this

section, we discuss two ways of making resources that are both composable and predictable.

The first and most straight-forward technique to get composable and predictable resources is to simply combine the approaches in Sections 2.2.2 and 2.2.3. We call this technique *worst-case predictable resource scheduling* and it corresponds to moving from a predictable shared resource via edge ⑨ and from a composable resource via edge ⑩ to a composable and predictable resource. This implies that the resource is predictable and that each request has a useful bound on WCET that is independent of other requestors. It also means that the resource is shared using an arbiter that is both composable and predictable, such as TDM. Such an arbiter provides bounded interference from other requestors that is independent of their actual behaviors, making the resource composable and bounding the WCRT. Since the original approaches to composable and predictable resources apply to both preemptive and non-preemptive resources, the same property holds for this combination. It furthermore inherits the possibilities for slack management, previously explained in Section 2.2.2.

A benefit of this approach to make resources composable and predictable is that it is easy to conceptually understand and implement. A drawback is that it only applies to resources where the execution time of a request is independent of requests from requestors belonging to applications other, as previously described in Section 2.2.2. If this is not naturally the case, it can be achieved by delaying all executions to be equal to the WCET. However, this may be costly if the variation in execution time due to other applications is large, preventing it from being efficiently applied to scarce resources, such as SDRAM. Instead, this technique is used in the processor tile presented in Section 2.3 and for composable and predictable SRAM sharing using TDM in [17].

The second technique is called *predictable resource scheduling with worst-case delay* and addresses the problem of efficiently dealing with variable execution times and extends composability to support any predictable arbiter. The problem with most predictable arbiters is that they typically cause the times at which the resource accepts requests and sends responses to a requestor to change due to variable interference from other requestors, making it non-composable. The key idea behind this technique is to make the system composable by removing the variation in interference, both from other applications and the resource itself. We accomplish this by starting from a predictable shared resource and then delay all signals sent to a requestor to *emulate maximum interference from other requestors*. A requestor hence always receives the same worst-case service no matter what other requestors are doing. This technique corresponds to achieving composability for a predictable shared resource using edge ⑧ in Fig. 2.3. The implication of this approach is that the interface presented towards the requestor is temporally independent of other requestors. Variation in starting times and response times may be visible on the resource side of the interface, but not on the requestor side. This is similar to the composable component interfaces proposed in [23].

The technique implies delaying responses in a response buffer until their WCRT to prevent the requestor from receiving it prematurely if there is little interference,

or if the variable execution time is short. However, making the WCRT independent of other applications is only one of the two requirements for a composable resource. The second requirement states that the starting time must also be independent. This is not the case if a request is scheduled earlier than its worst-case starting time. In this case, another request may be admitted into the resource prematurely, resulting in a different starting time. This problem is addressed by basing request accept signals on worst-case starting times of previous requests, as opposed to actual starting times. Requests are hence admitted into the resource in a composable manner, regardless of the interference experienced by others.

Figure 2.5 compares 'predictable resource scheduling with worst-case delay' to 'composable scheduling of preemptive resources', previously discussed in Section 2.2.2. Figure 2.5a illustrates that requests are scheduled immediately after a finished execution using 'predictable resource scheduling with worst-case delay', but that responses are delayed until the WCRT. In contrast, Fig. 2.5b (identical to Fig. 2.4a) shows that 'composable scheduling of preemptive resources' delays scheduling until the WCRT, but releases responses immediately after a finished execution.

'Predictable resource scheduling with worst-case delay, has two major benefits compared to 'composable scheduling of preemptive resources': 1) It extends the use of composability beyond resources and arbiters that are inherently composable. It is hence not limited to resources where the execution times of requestors are independent, but can efficiently capture the behavior of any predictable resource. 2) It supports any predictable arbiter, enabling service differentiation that increases the possibility of satisfying a given set of requestor requirements [2]. For example, using an arbiter that is more sophisticated than TDM can lead to reduced over-allocation, and allow lower latencies or higher throughput on a resource. These

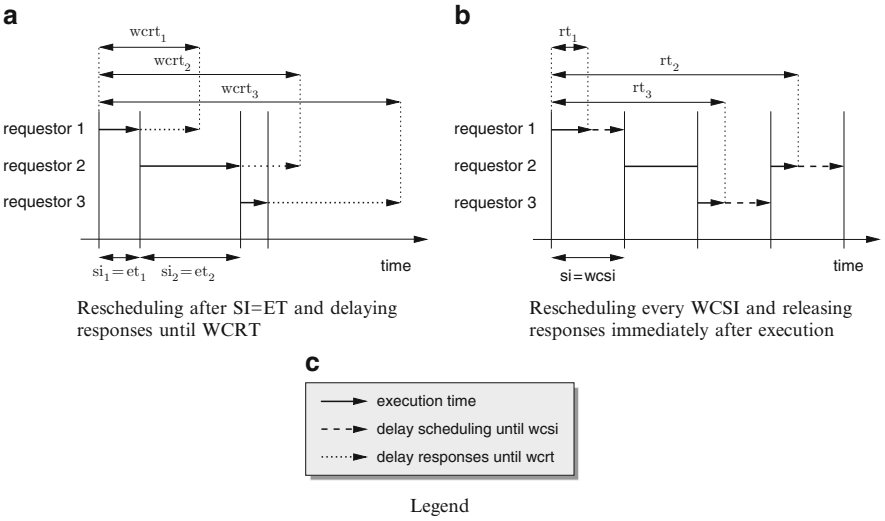


Fig. 2.5 Delaying scheduling until WCSI vs. delaying responses until WCRT

characteristics make the approach suitable for memory tiles with SDRAM, as we will further explain in Section 2.5.

The main drawback of this technique is related to slack management. This approach makes the temporal behaviors of the requestors independent of each other, thus implementing composability at the level of requestors instead of at the level of applications. It is hence not possible to benefit from unused resource capacity reserved by requestors belonging to the same application, which may negatively impact performance.

2.3 Processor tile

Having reviewed the different approaches to achieving composability and predictability, we proceed by looking at how it is actually implemented in a multi-processor system, starting with the processor tile. We consider a mixed time-criticality system, where the processor executes a mix between real-time and non-real-time applications. In this section, we first present the strategy to achieve composability of applications on a processor tile, followed by our approach to implementing predictability. The architecture of the processor tile is shown in Fig. 2.1. The components of this tile are discussed in the following sections.

2.3.1 Composability

Processors execute requests, corresponding to task iterations. The execution time of a request is hence the time it takes to execute a task iteration on the processor. Real-time tasks must have a WCET, which means that they complete an iteration in bounded time. This is not necessarily the case for non-real-time tasks. In mixed time-criticality systems, where these types of tasks share resources, the WCRT of real-time tasks can only be bounded if resources are preemptive. Composability in the processor is hence implemented using the technique 'composable scheduling of preemptive resources'. The key ingredients to achieve composability in this resource are thus found on the path ②, ⑤, and ⑦ in Fig. 2.3 and constitute: 1) preemption, 2) enforcing a constant scheduling interval equal to WCSI, and 3) using a composable arbitration scheme.

For a processor, the WCSI defines a task slot with bounded duration when a task can utilize the processor. After a task slot finishes, an operating system (OS) decides which task to execute next during an OS slot. To ensure independent starting times and response times of tasks, required for composability, not only the task slots, but also the OS slot, must have a constant duration and fixed starting times.

The execution time of the OS may depend on the number of applications and tasks it has to schedule. If the OS slot is not forced to a constant duration at least equal to its

WCET, it is impossible to ensure that task starting times and response times are independent of the presence or absence of other applications in the system. Furthermore, common OSes check if tasks are ready to execute, which depends on the availability of their input data and output space. For composability, the time at which this check is performed must be independent of other applications. 'Composable scheduling of preemptive resources' requires the execution times of tasks to be independent. The functional state of the processor tile at a task switch must hence be unable to affect the execution time of the scheduled task. This may imply that the processor instruction pipeline should be empty, and that potential caches should be cleared of all data to avoid cache pollution. In the following sections, we present the mechanisms to enforce constant-duration task and OS slots. Following this, we describe the scheduling of applications and tasks, which relies on this property.

2.3.1.1 Constant task slots

To enforce a task slot with constant duration and fixed starting times, we use a timer that interrupts the processor after a programmable fixed duration. When receiving an interrupt, the first instruction of the interrupt service routine jumps to OS code, giving control to the OS. This can be implemented with a dedicated timer per tile that is accessed via a memory-mapped peripheral bus or an instruction-mapped port. By using a timer outside the processor, in an always-on clock domain, the processor can enter a low-power state during idle periods without stopping the timer [13].

To get a constant-duration task slot, the processor should be interruptible in (preferably short) bounded time. However, processors are typically not interruptible while instructions are still in the pipeline. The time to start the interrupt service routine, referred to as the interrupt latency, thus depends on the execution time of the currently executing instructions. The time it takes to finish executing an instruction depends exclusively on the processor, except for instructions that involve other resources. For example, a load from non-local memory also uses the interconnect and a remote memory. Depending on the predictability and sharing of those resources, such a load may take thousands of cycles to complete (e.g. when it has a low priority in the NOC and memory tile).

By restricting the number of outstanding remote-read transactions, the WCET of a task and its worst-case interrupt latency can be computed, but will be prohibitively high (thousands of cycles). We hence use an alternative approach by restricting the processor to only using local (instruction and data) memories and use Remote Direct Memory Access (RDMA) engines to communicate outside the processor tile. Remote accesses may stall the RDMA, while the processor only polls locally, resulting in a short interrupt latency. Note that even with only local reads, the execution time of the interrupt service time is bounded, but not constant. For example, division and multiplication instructions take more cycles than NOP or jump instructions.

The processor programs the RDMA to read or write data on remote memories residing inside another processor tile, or in a memory tile. Programming the RDMAs is done using only local load and store instructions. An additional advantage of using RDMAs is that they decouple computation and communication, enabling them to be overlapped in time. In this chapter, we assume that the local memories of processor tiles are large enough to store the following state for all tasks mapped on the tile: 1) instructions, 2) (private) data, and 3) all the buffers (for input and output tokens) needed for an iteration. RDMAs are hence only used for inter-task communication between tasks mapped on different processors. This communication is implemented using uni-directional FIFO buffers with finite size. These FIFO buffers are located either in the local memory of the consumer (if the memory space in the processor tile is sufficiently large), or in a remote memory tile. The producer always posts the data in the buffer via a RDMA write. In Fig. 2.1, the data travels from the data memory in the producer tile, through the RDMA to the interconnect. The interconnect then delivers it to the local memory in the consumer tile. Alternatively, the producer RDMA places the data in a remote memory tile, from where it is copied by the consumer RDMA to the data memory in its tile. In all cases, the FIFO administration [31], consisting of read and write pointers, is located in the producer and consumer tiles.

To achieve composability, a RDMA has to be composable if shared between applications. Since RDMAs are simple finite state machines, we do not share them between applications. Instead, each application has its own RDMA, but for maximum performance, each FIFO of each task can be given its own RDMA. For simplicity, Fig. 2.1 shows only one RDMA per tile. Note that the local memory should also be made composable using the techniques detailed in Section 2.5.

2.3.1.2 Constant OS slot

As previously explained, the OS slot should have a constant starting time and duration. Given a constant task slot duration, the only requirement to achieve a constant OS starting time is that the task-to-OS switching time should be constant. The task-to-OS switching time is equal to the interrupt latency of the timer, which depends on the instructions in-flight on the processor. We force the interrupt latency to be constant and equal to its WCET via a mechanism to delay actions (execution) until a fixed future moment in time, as described below.

Our approach to enforce a constant OS slot is to *inhibit execution* on the processor until its WCET is reached, thus making the OS execution composable. This corresponds to the technique 'composable scheduling of non-preemptive resources', which uses edges ①, ④, and ⑦ in Fig. 2.3. This can be implemented in several ways. Polling on a timer [10] is the simplest, but prevents clock-gating of the processor. If the processor has a halt instruction, the processor can be halted after the OS finishes its execution. The tile timer, programmed before the halt instruction, wakes up the processor at the WCET. When a halt instruction is not available, the

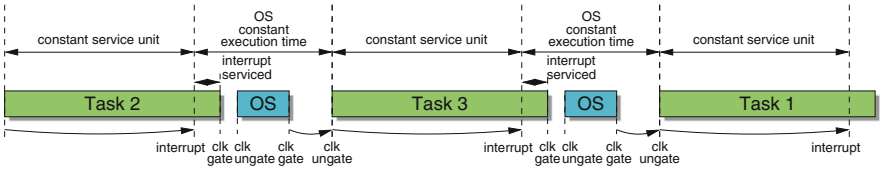


Fig. 2.6 Processor slots and task switching time line

processor clock can be disabled by a voltage-frequency control unit (VFCU in Fig. 2.1) until the WCET.

Figure 2.6 presents the time line with the seven main events when performing a task switch: 1) the interrupt is raised, 2) the interrupt is served, 3) the processor ungate moment in time is programmed, 4) the clock is gated up to the WCET of the interrupt latency, 5) the OS is executed, 6) the processor ungate moment in time is programmed, and finally 7) the clock is gated up to the WCET of the OS.

2.3.1.3 Two-level application and task scheduling

The constant-duration task and OS slots ensure that task slots start at fixed points in time, and that there is a bounded WCSI. A task iteration that has a WCET on a non-shared processor tile hence has bounded WCET and WCRT on a shared tile. As mentioned before, the functional state of the processor tile at the start of a task slot must be independent of other applications to avoid possible interference.

By using a composable scheduler, interference between all tasks is removed. However, this is unnecessarily strict, since it also prevents slack from being used by tasks belonging to the same application. Moreover, different applications benefit from using different schedulers, such as static-order, TDM, or Credit-Controlled Static-Priority arbitration [5] (CCSP, further described in Section 2.5). The processor addresses this problem by using a two-level arbitration scheme: a composable inter-application arbiter (TDM) that schedules applications, and an intra-application arbiter that schedules tasks within an application. The composable inter-application arbiter ensures the isolation between applications, while the intra-application arbiters are chosen to fit the requirements of the application tasks. The intra-application arbiters are free to distribute slack to improve performance of the tasks.

2.3.2 Predictability

As already mentioned, we target mixed time-criticality systems that concurrently execute a set of real-time and non-real-time applications. For real-time applications, we require the WCET of each task iteration to be known. The execution time of a task on a processor is hence required to be predictable, which excludes the

use of out-of-order execution, speculation, and caches with random replacement policies [40].

To derive the end-to-end application performance (e.g. throughput, latency, etc.), applications are modeled as data-flow graphs [25, 36]. The nodes in the data-flow graphs are referred to as *actors* that are connected via directional *edges*. Each actor *fires* whenever its firing rule is satisfied. A firing rule specifies for each incoming and outgoing edge, the number of input tokens required and the number of output tokens produced, respectively. The data-flow model naturally describes a streaming application: a task is an actor, and a task iteration is an actor firing. FIFO communication between two tasks is represented as a pair of opposing edges, one modeling the communicated data, and the other modeling the available inter-task buffer space.

If several tasks share the same processor, predictable inter-task arbitration is required. Examples of such arbitration are TDM, CCSP, and round robin. Moreover, the sharing and arbitration effects should be taken into account when calculating the end-to-end application performance. Modeling of different arbitration policies as data-flow graphs is presented in [19, 28].

2.4 Interconnect

The processor and memory tiles in the system communicate via a global on-chip interconnect, as shown in Fig. 2.1. Typically, processors act as memory-mapped *initiators* and memory tiles as memory-mapped *targets*. This is seen in the figure, where initiator and target ports are colored black and white, respectively. When tasks execute on a processor, they give rise to read and write requests that are delivered to the appropriate memory tile based on the address, and a response is potentially delivered back to the processor. The *requestors* of the interconnect, according to Section 2.2.1, are thus the ports of the processor and memory tiles.

To deliver the aforementioned functionality, the interconnect is subdivided into a number of architectural components [16]. We first present a brief overview of the components and then continue to discuss how they provide composability and predictability. When a request is presented to the interconnect by an initiator, it is serialized by a protocol shell into a sequence of words. These words are then passed through a clock domain crossing (CDC) to transition from the clock domain of the initiator to that of the network, making the platform globally-asynchronous locally-synchronous (GALS) [30]. The data is then sent through the network, comprising Network Interfaces (NI) and routers (R), through a logical *connection*. The NI packetizes the data and determines the route through the network. The routers merely forward the data to its destination NI where it is depacketized, before transitioning to the clock frequency of the target in another clock domain crossing. The shell then deserializes the request and presents it to the actual target port. A response, if present, follows the same logical connection back through the

network until it reaches the initiator. The interconnect resource hence comprises protocol shells, clock domain crossings, NIs, routers and links.

2.4.1 Composability

The protocol shells are not shared by connections and thus require no special attention to deliver composability. They are furthermore simple state machines that can be considered predictable. Moreover, the shells serialize the memory-mapped transactions of the tiles independently of their protocol, burst size, type of transaction etc. Thus, when presented to the NIs as a stream of words, the level of flow control and preemption is a single word (using a FIFO protocol).

Once the serialized transactions are delivered to the NIs, each logical connection has dedicated input and output buffers in the NIs. At this level, the network can thus be seen as a set of composable distributed FIFOs, interconnecting pairs of protocol shells. The NIs packetize the individual words of data in units of *flits* and send them through the network links and routers. Each packet starts with a header (flit) with the path to the destination output buffer. In contrast to many on-chip networks, our interconnect does not perform any arbitration inside the network. The routers simply obey the path encoded in the packet headers, and push the responsibility of scheduling and buffering to the NIs. Thus, all arbitration takes place in the NI, and the routers merely forward the flits until they reach the destination NI, making the network appear as a single (pipelined) shared resource.

To make the network as a whole composable (and predictable), we use the technique 'worst-case predictable resource scheduling'. We describe the implementation of this technique in three steps, corresponding to edges ⑤, ⑦, and ⑩ in Fig. 2.3. Firstly, the network resources are preemptive at the level of flits (edge ⑤). A scheduling decision is thus taken for every flit, independent of the length of the packets. Furthermore, as we have already seen, the data in the NI FIFOs has no notion of memory-mapped transactions, and there is consequently no correspondence between transactions and packets. As there is no buffering inside the router network, the NIs use *end-to-end flow control* to ensure the availability of buffer space. Consequently, flits are only injected if they are guaranteed not to stall anywhere inside the network.

Secondly, the flit size is fixed at three words, resulting in a constant scheduling interval of three cycles. If a connection's input buffer is empty or if it runs out of flow control credits, it uses only one or two words of the three-word flit. The constant flit length corresponds to making all scheduling intervals equal to the WCSI, indicated by edge ⑦ in Fig. 2.3. It is worth noting that there is no need to determine how long it takes for other requestors flits to reach their destination, only how long it takes until a new flit can be scheduled, i.e. the execution time and response time of other requestors is irrelevant.

Thirdly, the fixed flit length is combined with a global schedule of the logical connections, where each NI regulates the injection of flits using a TDM arbiter [11], such that contention never occurs on the network links. The schedule relies on a

(logical) global synchronicity of the network components, but the concept has been demonstrated on both mesochronous and asynchronous implementations of the network [18]. The TDM schedule is programmed at run time according to the running use-case, but is typically determined at design time.

The last part of the interconnect composability is enforced insertion of packet headers for non-consecutive flits. That is, if another connection could have used the link, assume it did (even if it did not), and insert a new packet header. The header insertion ensures that the arbiter is *stateless* in terms of influence from other requestors.

2.4.2 Predictability

With the aforementioned mechanisms in place, the interconnect offers composability at the level of connections, between pairs of protocol shells. Predictability additionally requires worst-case response times for the shared resources. As discussed in detail in [19], the temporal behavior of a connection depends on the TDM scheduler settings, the path length, and the size of the input and output buffers. The scheduler determines how long words have to wait in the input buffer until injected into the network, once eligible. The path, in turn, determines the time required to traverse the network (without stalling). The input and output buffers affect the time at which words are accepted and become eligible for scheduling. All these contributions can be bounded and captured in a data-flow graph, thus offering predictability.

2.5 Memory tile

This section presents our memory tile and discusses the techniques employed to implement composability and predictability. The architecture of the memory tile, shown in Fig. 2.1, is divided into a *front-end* and a *back-end*. The front-end is independent of memory technology and contains buffering, arbitration, and components to make the memory tile composable. The back-end interfaces with the actual memory device and makes it behave like a predictable resource. The back-end is hence different for different types of memories, such as SRAM and SDRAM, as indicated by the figure. The components in the architecture are discussed further in the following sections.

Although our memory tile is general and supports both SRAM and DDR2/DDR3 SDRAM, we will focus the discussion on SDRAM, since these memories have three important characteristics that make the implementation of composability and predictability challenging. 1) The execution time of a request and the bandwidth offered by the memory is variable and depends on other requestors. 2) Some memory requestors are latency critical and require low response time to reduce the number of stall cycles on the processor. 3) For cost reasons, SDRAM bandwidth is a scarce resource that must

be efficiently utilized. This section is organized as follows. Firstly, Section 2.5.1 explains how to make an SDRAM behave like a predictable shared resource. Section 2.5.2 then discusses how to make the predictable shared memory composable.

2.5.1 Predictability

Section 2.2.1 states that a predictable resource must provide a useful bound on WCET to all requests. In addition, a memory tile must bound the bandwidth offered to a requestor to ensure that bandwidth requirements are satisfied. This section elaborates on how our memory tile delivers on these requirements. The memory tile follows our general approach to predictable shared resources and combines a predictable resource with predictable arbitration. First, the concepts behind an SDRAM back-end that makes the memory behave like a predictable resource, corresponding to edge ① in Fig. 2.3, are explained. We then discuss how to share the predictable memory between multiple requestors, covering edge ③.

2.5.1.1 Predictable SDRAM back-end

SDRAM memories are challenging to use in systems with real-time requirements because of their internal architecture. An SDRAM memory comprises a number of banks, each containing a memory array with a matrix-like structure, consisting of rows and columns. A simple illustration of this architecture is shown in Fig. 2.7. Each bank has a row buffer that can hold one open row at a time, and read and write operations are only allowed to the open row. Before opening a new row in a bank, the contents of the currently open row are copied back into the memory array. The elements in the memory arrays are implemented with a single capacitor and a resistor, where a charged capacitor represents a logical one and an empty capacitor a logical zero. The capacitor loses its charge over time due to leakage and must be refreshed regularly to retain the stored data.

The SDRAM architecture makes the execution time of requests highly variable for three reasons. 1) A request targeting an open row can be served immediately, while

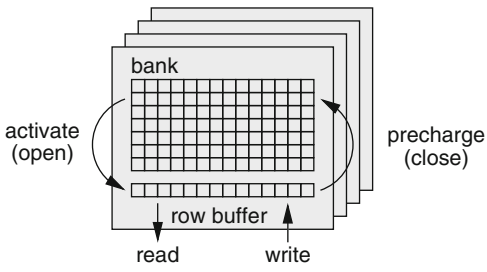


Fig. 2.7 The architecture of an SDRAM memory and behaviors of some important SDRAM commands

it otherwise needs the current row to be closed and the required row to be opened. 2) The data bus is bi-directional and requires a number of cycles to switch from read to write and vice versa. 3) The memory must occasionally be refreshed before executing the next request. The impact of these factors may cause the execution time of an SDRAM burst to vary by an order of magnitude from a few clock cycles to a few tens of cycles.

The behavior of an SDRAM memory is determined by the sequence of SDRAM commands that are communicated from the back-end of the memory tile to the memory device. These commands tell the memory to activate (open) a particular row in the memory array, to read from or write to an open row, or to precharge (close) an open row and store its contents back into the memory array. There is also a refresh command that charges the capacitors of the memory elements to ensure that the contents of the memory array are retained. The behaviors of some of these commands are illustrated in Figure 2.7. Scheduling SDRAM commands is not a trivial task, since there are a considerable number of timing constraints that must be satisfied before a command can be issued. These timing constraints are typically minimum delays between issuing particular SDRAM commands, such as two activates, or an activate and a read or a write.

Existing SDRAM controllers can be divided into two categories, depending on how they schedule SDRAM commands. Statically scheduled controllers [7] execute precomputed command schedules that are guaranteed at design time to satisfy all timing constraints of the memory. Executing precomputed schedules makes these controllers predictable and easy to analyze. However, they are also unable to adapt to the dynamic behavior of applications in contemporary SOCs, such as bandwidth requirements or read/write ratios that vary over time. The second category of controllers uses dynamic scheduling of commands, which requires the timing constraints to be enforced at run time. These controllers [20, 21, 26, 29, 35] have sophisticated command schedulers that attempt to maximize the average offered bandwidth and to reduce the average latency at the expense of making the resource extremely difficult to analyze. As a result, the offered bandwidth can only be estimated by simulation, making bandwidth allocation a difficult task that must be re-evaluated every time a requestor is added, removed or is modified.

We use a hybrid approach to SDRAM command scheduling that combines elements of statically and dynamically scheduled SDRAM controllers in an attempt to get the best of both worlds. Our approach is based on *predictable memory patterns* [1], which are precomputed sequences (sub-schedules) of SDRAM commands that are known to satisfy the timing constraints of the memory. These patterns are dynamically combined at run-time, depending on the incoming request streams. The memory patterns exist in five flavors: 1) read pattern, 2) write pattern, 3) read/write switching pattern, 4) write/read switching pattern, and 5) refresh pattern. The patterns are created such that multiple read or write patterns can be scheduled in sequence. However, a read pattern cannot be scheduled immediately after a write pattern. In this case, the read pattern must be preceded by a write/read switching pattern. This works analogously in the other direction. The refresh pattern can be scheduled immediately after either a read pattern or a write pattern. Both read and

write patterns can be scheduled immediately after a refresh without any preceding switching patterns.

The read and write patterns consist of a fixed number of SDRAM bursts, all targeting the same row in a bank. The bursts are issued to the different banks in sequence, since the data bus is shared between all banks to reduce the number of pins on the SDRAM interface. The fixed number of bursts is hence first sent to the first bank, then to the second, and so forth in an interleaving fashion until all banks have been accessed. This way of accessing the SDRAM results in a short period with frequent accesses, followed by a longer period without any accesses. The patterns exploit bank-level parallelism by issuing activate and precharge commands to the banks during the long intervals in which they do not transfer any data. The read and write patterns are hence very efficient in terms of bandwidth, since it is possible to hide a significant part of the latency incurred by activating and precharging rows. This limits the overhead cycles incurred by always precharging a bank immediately after it has been accessed, which is known as a closed page policy. We implement this policy, as it effectively removes the dependency on rows opened by earlier requests by returning the memory to a neutral state after every access. Removing this dependency between requests is a *key element* in our approach, since it *reduces the variation in the offered bandwidth and latency*, enabling tighter bounds on bandwidth and WCRT to be derived.

Although interleaving memory patterns allow us to bound the offered bandwidth, they come with two drawbacks. The first drawback is that continuously activating and precharging the banks increases power consumption compared to if a single bank is used at a time. The second drawback is that the memory is accessed with large granularity and hence requires large requests to be efficient. An efficient access requires at least one SDRAM burst to every bank. A typical burst size for SDRAM is eight words and the number of banks is either four or eight. The minimum efficient request size for a 32-bit memory interface is hence between 128-256 B, depending on the size and generation of the DDR SDRAM [3]. Working with large requests in a non-preemptive manner also means that urgent requests can be blocked longer, resulting in longer WCRT.

Requests are dynamically mapped to patterns in a non-preemptive manner by the command generator in the SDRAM back-end. A scheduled read request maps to a read pattern, possibly preceded by a write/read switching pattern. Similarly, a write request is mapped to a write pattern and potentially a preceding read/write switching pattern. Refresh patterns are scheduled automatically by the SDRAM back-end on a regular basis between requests. The mapping from requests to patterns and from patterns to SDRAM bursts is shown for an SDRAM with four banks in Fig. 2.8. The figure illustrates that the execution time of a request of four bursts varies depending on whether or not a switching pattern is required and if a refresh is scheduled before the request.

The benefit of memory patterns is that they raise SDRAM command scheduling to a higher level. Instead of dynamically issuing individual SDRAM commands, like a dynamically scheduled SDRAM controller, our back-end issues memory patterns that are sequences of commands. This implies a reduction of state and constraints that have

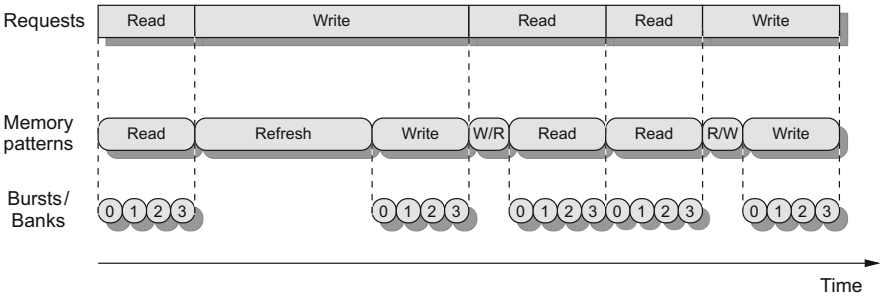


Fig. 2.8 Mapping from requests to patterns to SDRAM bursts

to be considered, making our approach easier to analyze than completely dynamic solutions. Memory patterns allow a lower bound on the offered bandwidth and WCRT to be determined, since we know the execution time of each pattern, how much data they transfer, and what the worst-case sequence of patterns is. This analysis is presented and experimentally evaluated in [3]. The use of memory patterns gives our approach the predictability of statically scheduled memory controllers. In addition, our approach has some properties of dynamically scheduled controllers, such as the ability to dynamically choose between read and write requests, and the use of run-time arbitration. The latter is discussed in the following section.

2.5.1.2 Predictable arbitration

After the previous section, we assume that we have a predictable memory, such as a zero-bus-turnaround SRAM or our SDRAM back-end based on predictable memory patterns, where useful bounds on both the offered bandwidth and the WCET of requests are known. In this section, we consider the effects of sharing the predictable memory between multiple requestors. As mentioned in Section 2.5.1, we require a predictable arbiter, where the number of interfering requests before a particular request is served is bounded. This enables the WCRT to be determined. There are a large number of predictable arbiters described in literature, such as TDM and round robin. However, most of these arbiters are unable to provide low response time to critical requestors, making them unsuitable for memory tiles. This problem is addressed by priority-based arbitration, but as previously mentioned in Section 2.2.2, conventional static-priority scheduling is not starvation-free and cannot be used to build predictable or composable systems. To address this issue, we have developed a Credit-Controlled Static-Priority (CCSP) arbiter [5]. The CCSP arbiter consists of a rate regulator and a static-priority scheduler. The rate regulator isolates requestors by enforcing an upper bound on the provided service, according to an allocated budget. It furthermore decouples allocation granularity and latency, which enables bandwidth to be allocated with an arbitrary precision without affecting latency [4]. A clean trade-off is hence provided between over allocation and area, allowing

over allocation to become negligible. This is essential for scarce SOC resources with very high loads, such as SDRAMS. The static-priority scheduler schedules the highest priority requestor that is within its budget. The use of priorities decouples latency and rate, thus enabling low latency to be provided to requestors with low bandwidth requirements without wasting bandwidth. The combination of rate regulator and static-priority scheduler makes the arbiter predictable, while still being able to satisfy the requirements of latency-critical requestors.

A rate regulator creates a separation of concerns and makes it possible to bound the WCRT of a requestor in a static-priority scheduler without relying on the cooperation of higher priority requestors. Instead, the bounds on WCRT are based on the *allocated bandwidths and burstinesses*, which are determined at design time. However, to be completely robust, we also need to be independent of the sizes of scheduled requests to prevent a malfunctioning requestor from preventing access from others by issuing very large requests. We solve this problem using preemptive service, which is enabled by the *atomizer* [17] block, shown in Fig. 2.1. The atomizer splits requests into smaller atomic service units, which are served by the memory in a known bounded time. This effectively makes the memory preemptive on the granularity of an atomic service unit. The size of the atomic requests are fixed and determined at design time. It is chosen to be the minimum request size that can be efficiently served by the resource. For an SRAM, the natural service unit is a single word, but it is much larger for an SDRAM with predictable memory patterns. For these memories, the appropriate size might be between 16 and 256 words, depending on the memory device and the desired trade-off between efficiency and latency.

2.5.2 Composability

Composability in the memory tile is achieved using the technique called 'predictable resource scheduling with worst-case delay'. This is for two reasons related to the characteristics of SDRAM, presented earlier. Firstly, because SDRAMS have highly variable execution times that depend on other requestors. This prevents the use of 'worst-case predictable resource scheduling' unless the execution time is made independent of other requestors. This is possible by delaying all executions until the WCET by setting WCSI=WCET. For most patterns, this involves assuming a read/write switch for every memory request. Although possible to implement, this may increase the response time and decrease the offered bandwidth by up to 20% [3]. This is not a feasible option, considering that SDRAM bandwidth is a scarce and expensive resource. The second reason is that the first technique is limited to composable arbiters, such as TDM or static scheduling, which cannot distinguish requestors with low response time requirements. However, the second technique works with any predictable arbiter, such as our priority-based CCSP arbiter. The technique is implemented by the *delay block*, shown in Fig. 2.1. This component emulates worst-case interference from other requestors to provide a

composable interface towards the atomizer. This makes the interface of the entire front-end composable, since the atomizer is not shared.

It is worth noting that the delay block could have been placed in the processor tile, as opposed to in the memory tile. The advantage of this is that it offers composability to platforms with predictable, but not composable, interconnect by eliminating interference from both the interconnect and the memory tile at once. However, our interconnect is composable in itself using another technique, defeating the purpose of moving the delay block. Delaying in the processor tile furthermore comes with the drawback of making debugging of the platform more difficult, since the states of both the interconnect and memory tile change if applications are added, removed, or modified.

2.6 Experiments

The proposed composability inducing mechanisms are implemented for each resource of an SOC prototyped on FPGA having four processor tiles with one MicroBlaze core each, one memory tile and an Æthereal NoC [12]. On this platform, we execute several use-cases constructed using the following applications: a simple synthetic application (*A1*), an H.264 video decoder [39] (*A2*), and a JPEG decoder (*A3*), each consisting of a set of communicating tasks. Figure 2.9 presents the task graphs and the task-to-processor mapping of these applications.

If the SOC is composable, the behavior of an application should remain the same regardless of the presence or absence of other applications. We investigate composability in two ways: first by checking the cycle-level differences between some signals of the MicroBlaze interface in multiple simulations, and second by verifying whether the response time and starting time of an application remains constant when other applications are added in the system.

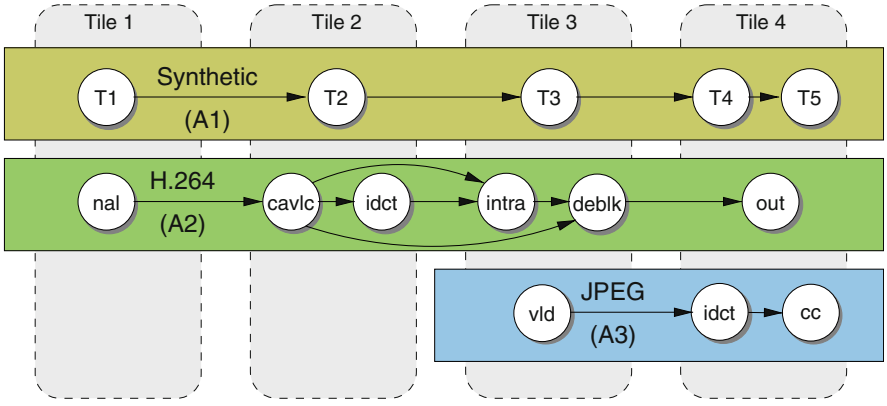


Fig. 2.9 The applications and mappings used in the experiments

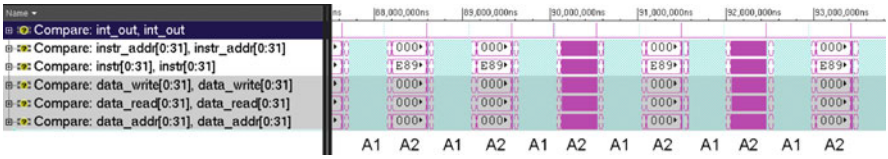


Fig. 2.10 MicroBlaze signal differences when *A1* varies its behavior

To investigate composability at the cycle level, we run two simulations and compare a number of signals in the first MicroBlaze core. For our simulations, we utilize the synthetic application, *A1*, and the H.264 application, *A2*. The *int_out* signal (the timer interrupt) indicates the border between the end of a task slot and the beginning of an OS slot. This signal is kept high until the processor acknowledges that the interrupt is being served. In the first simulation, *A1* transfers data tokens of 4 KBytes and in the second it transfers data tokens of 16 Bytes. Figure. 2.10 presents the signal differences between the two simulations. The application TDM slot assignment is shown at the bottom of the diagram. We observe that signals in the task slots of *A2* are identical, whereas, the signals in *A1*'s slots change, as expected. The striped zone represents cycles that differ between the two runs. As seen in Fig. 2.10, the timer interrupt signals are not always identical in the two simulations. The reason for this is that different instructions are interrupted in different simulations, thus the *int_out* signal has different timing. The comparison between the two traces clearly shows that the only signal differences occur in the time slots of the changed applications and in the OS slot, indicating that cycle-level composability is achieved.

To investigate the potential variations in the starting time and response time of applications, we run the H.264 and JPEG applications alone (*H.264-single* and JPEG-single, respectively), and in combination with the synthetic application (*H.264-multi* and JPEG-multi, respectively) on the FPGA. In these cases, we compare the response times and starting times of each iteration of each H.264 and JPEG task. If the system is composable, these times should be identical in different runs, regardless of the presence or absence of the synthetic application. Figures 2.11 and 2.12 present the response time differences for a JPEG and a H.264 task in two cases: 1) when all applications share a single RDMA engine (one RDMA per tile), and 2) when each application has its own RDMA engine (one RDMA per application).

As shown in the figures, the response times differ when using a single RDMA per tile, thus revealing interference. On the other hand, the response time difference is zero when using a single RDMA per application, showing no interference. Due to lack of space, we do not present the response times and starting times of all tasks. The observed behavior is the same, which means that the system is composable when using one RDMA per application. However, sharing a RDMA engine results in interference between applications, and variations in application timing behavior, just as expected.

In conclusion, we experimentally show that the processor behavior remains the same at both the cycle level and at the task-iteration level, indicating that our SOC is temporally composable. The inspected signal traces in this section only cover the processor. However, the experiments strongly suggest that the interconnect and the

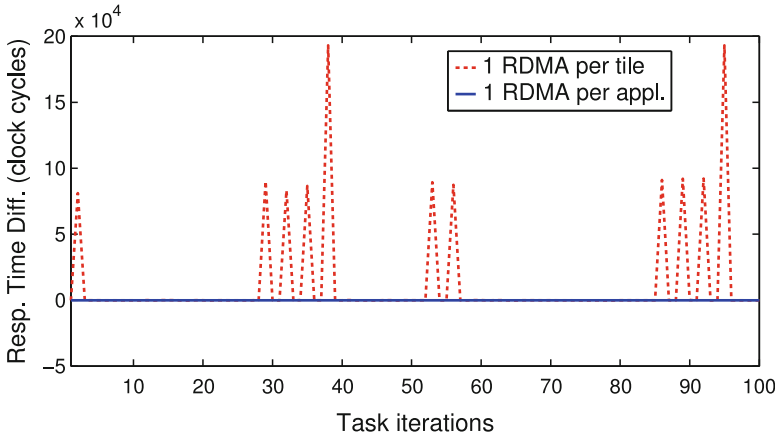


Fig. 2.11 JPEG, *vld* task response time difference between RDMA per tile or per application

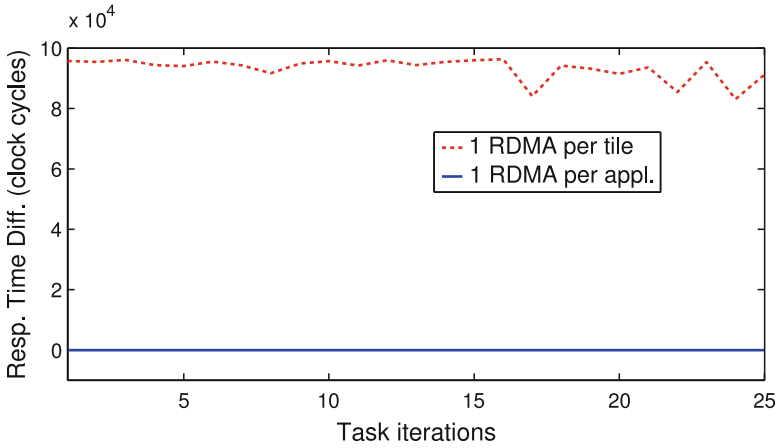


Fig. 2.12 H.264, *deblock* task response time difference between RDMA per tile or per application

memory tile are also composable. Otherwise, the timing variations in these resources would have resulted in variations in the response time of the tasks, or at the cycle-level timing of the processor signals.

2.7 Conclusions

This chapter addresses the verification and integration problem in embedded multi-processor platforms that have resources shared by a mix of real-time and non-real-time applications. We discuss two complexity-reducing concepts:

composability and *predictability*. Applications in a composable system are completely isolated and cannot affect each other's functional or temporal behaviors. Applications in a use-case can hence be verified individually instead of together, resulting in smaller state spaces. This enables a faster verification process, e.g. using simulation-based techniques, that can start as soon as the first application in a use-case is available. Predictable systems, on the other hand, provide lower bounds on application performance, such as latency and throughput. This enables applications to be verified at design time using formal performance analysis frameworks. The benefit of formal performance verification is that conservative performance guarantees can be provided for all possible combinations of initial states of resources and arbiters, all input stimuli, and all concurrently executing applications. However, formal approaches require performance models of the software, the hardware, and the mapping, which are not yet widely adopted by industry. Composability and predictability hence both solve important parts of the verification problem and provide a complete solution when combined.

Composability and predictability are different properties in the sense that predictability implies the existence of useful bounds on temporal behavior and is hence a property of a single application mapped on a set of resources. Composability implies complete isolation between applications and is a property of multiple applications sharing a resource, each of which may be predictable or not. We formally consider temporal composability achieved if the starting times and response times of an application, i.e. when it is scheduled for resource access and when it finishes receiving service, are independent of other applications.

The contributions of this chapter are twofold. Firstly, we present a thorough overview of five techniques for achieving composability and/or predictability and highlight their respective strengths and weaknesses. Secondly, we show how to build a composable and predictable system by applying the proposed techniques to three common resource types: processor tiles, interconnects (networks-on-chip), and memories (both on-chip SRAM and off-chip SDRAM).

On an unshared resource, predictability means that a request with finite size has a bounded worst-case execution time (WCET). On a shared resource, we achieve predictability by combining resources and arbiters, each with predictable behaviors. This enables the worst-case response time (WCRT) of requests to be determined for any combination of predictable arbiter and resource.

Composability can be achieved in four ways, described in the following paragraphs. The first way is useful if the execution times of all requests cannot be bounded. However, this requires that they can be preempted after a chosen worst-case scheduling interval (WCSI), which is the maximum time between two arbitration decisions. To create the premises of independent starting times, all scheduling intervals must have constant length equal to the WCSI. This decouples the starting time of a request from the execution times of previous ones. To enforce independent starting and response times, requests must be scheduled by a composable arbiter, such as time division multiplexing (TDM). The main limitation of this way to implement composability is that it only applies to preemptive resources in which

the execution time of a request is independent of requests from other requestors. This is the case for zero-bus-turnaround SRAM memories, but not for SDRAM.

The second way to implement composability applies particularly to non-preemptive resources. This technique requires that the resource is predictable and has a known WCET. The idea is to set the scheduling interval equal to the largest WCET of a request on the resource to make starting times independent of previous requests. Combining this with composable arbitration ensures that the worst-case response times are also independent. The two drawbacks of this technique are: 1) that execution times of requests have to be independent of requests from other requestors, just like for the previous method, and 2) making the scheduling interval equal to the longest WCET results in low resource utilization if there is a large difference between the average and worst-case execution time, which is the case for SDRAM memories.

The third and fourth ways to implement composability are based on predictability, resulting in resources with both properties. The third method is an extension of the first with an additional requirement that the composable arbiter is also predictable, such as TDM. This enables the WCRT to be computed for predictable applications with known WCET that is independent of other requestors.

The last way to implement composability (and predictability) applies to both preemptive and non-preemptive resources and supports variable execution times that depend on other requestors. It can furthermore be used with any combination of predictable resource and predictable arbiter. The key idea behind this approach is to make the system composable by enforcing maximum interference from other requestors to remove variation caused by other applications. This is accomplished by starting from a predictable shared resource and delay responses to emulate maximum interference from other requestors.

We experimentally demonstrate some of the proposed techniques on a tiled multi-processor system with MicroBlaze cores connected to an SRAM memory tile via a network-on-chip. Netlist simulations of this platform show that the cycle-level behavior of an application is unaffected, as the behavior of other applications changes, indicating composable execution.

References

1. B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, 2007.
2. B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *12th Euromicro Conference on Digital System Design (DSD)*, 2009.
3. B. Akesson, W. Hayes, and K. Goossens. Classification and Analysis of Predictable Memory Patterns. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2010.

4. B. Akesson, L. Steffens, and K. Goossens. Efficient Service Allocation in Hardware Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009.
5. B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.
6. ARM Limited. *AMBA AXI Protocol Specification*, 2003.
7. S. Bayliss and G. Constantinides. Methodology for designing statically scheduled application-specific sdram controllers using constrained local search. In *Field-Programmable Technology, 2009. International Conference on*, pages 304 –307, Dec. 2009.
8. M. Bekooij, A. Moonen, and J. van Meerbergen. Predictable and Composable Multiprocessor System Design: A Constructive Approach. In *Bits&Chips Symposium on Embedded Systems and Software*, 2007.
9. R. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.
10. M. Ekerhult. Compose: Design and implementation of a composable and slack-aware operating system targeting a multi-processor system-on-chip in the signal processing domain. Master's thesis, Lund University, July 2008.
11. K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.
12. K. Goossens and A. Hansson. The aethereal network on chip after ten years: goals, evolution, lessons, and future. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 306–311, 2010.
13. K. Goossens, D. She, A. Milutinovic, and A. Molnos. Composable dynamic voltage and frequency scaling and power management for dataflow applications. In *13th Euromicro Conference on Digital System Design (DSD)*, Sept. 2010.
14. P. Gumming. The TI OMAP Platform Approach to SoC. *Winning the SoC revolution: experiences in real design*, page 97, 2003.
15. A. Hansson, M. Coenen, and K. Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 954–959, 2007.
16. A. Hansson and K. Goossens. An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 99–108, 2009.
17. A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–24, 2009.
18. A. Hansson, M. Subbaraman, and K. Goossens. aelite: A flit-synchronous network on chip with composable and predictable services. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Apr. 2009.
19. A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques*, 2009.
20. S. Heithecker and R. Ernst. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 575–578, 2005.
21. E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture. ISCA '08. 35th International Symposium on*, pages 39–50, 2008.
22. International Technology Roadmap for Semiconductors (ITRS), 2009.

23. H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
24. H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *SOC Conference, IEEE International*, pages 87–90, 2008.
25. E. A. Lee. Absolutely positively on time: what would it take? *IEEE Transactions on Computers*, 38(7):85–87, 2005.
26. K. Lee, T. Lin, and C. Jen. An efficient quality-aware memory controller for multimedia platform SoC. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):620–633, 2005.
27. A. Molnos and K. Goossens. Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors. In *12th Euromicro Conference on Digital System Design (DSD)*, 2009.
28. O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 57–66, 2007.
29. O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers. *IEEE Micro*, 29(1):22–32, 2009.
30. J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, 2000.
31. A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.
32. OCP International Partnership. *Open Core Protocol Specification*, 2001.
33. Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, 2002.
34. R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, 2006.
35. J. Shao and B. Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 285–294, 2007.
36. S. Sriram and S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC, 2000.
37. L. Steffens, M. Agarwal, and P. van der Wolf. Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach. *ECRTS '08: Proceedings of the Euromicro Conference on Real-Time Systems*, pages 255–265, 2008.
38. C. van Berkel. Multi-core for Mobile Phones. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.
39. S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007, 2007.
40. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

Chapter 3
Hardware Support for Efficient Resource Utilization in Manycore Processor Systems

A. Herkersdorf, A. Lankes, M. Meitinger, R. Ohlendorf, S. Wallentowitz, T. Wild, and J. Zeppenfeld

Abstract Effective utilization of the available processing resources in current multi- and manycore systems primarily depends on the manual talent of the application programmer. This chapter analyses opportunities and suggests approaches to tackle the challenge of making proper use of parallel resources by means of a holistic, cross-layer and inter-disciplinary optimization of application, middleware and architecture aspects. Using heterogeneous network processors as an example, we show how application specific architecture optimizations in this processor domain can be adapted to benefit designs of homogeneous general purpose manycore systems. In addition, methods which have been applied successfully to HPC and scientific computing over the past decades are assessed and down-scaled to benefit manycores. Finally we show how bio-inspired principles (i.e., self-organization and self-adaptation) provide rich opportunities for meaningful adoption in both application-specific and general purpose manycores, for example to provide self-optimization of processor parameters and workload utilization. In summary, we present a set of suggestions for architectural improvements and building blocks that, from our perspective, are useful for future manycores in order to better support the exploitation of available parallel processing resources.

Keywords Manycore · Multicore · Hardware Support · Network Processing · Bio-Inspired · Self-Organization · Learning Classifier · Platform Optimization · Processing Efficiency · Hardware Accelerators · Supercomputing · Network Processing · Network-On-Chip · High Performance Computing

A. Herkersdorf (✉)
Institute for Integrated Systems, TU München, Arcisstr. 21, 80333 München, Germany
e-mail: herkersdorf@tum.de

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_3,

3.1 Introduction

Chip multiprocessors are becoming the de facto industry standard for processor architecture. In comparison with sophisticated uni-processors, multicore systems are superior in terms of scalable computing performance and power efficiency. Such multiprocessor system-on-chip (MPSoC) are found as either homogeneous or heterogeneous platforms, integrating identical or different types of programmable processing elements on a common silicon substrate.

Continued progress in scaling CMOS semiconductor technology ("Moore's Law") makes it technically feasible to implement on the order of one hundred or more such processing elements on a single chip. Commercial and academic examples, such as Tilera's TILE-Gx,[1] Intel's Rock Creek[2], or the TRIPS architecture [5], underpin the trend toward manycore architectures. In the following, we use the term core to refer to a programmable processing element or on-chip processor building block. Furthermore, whenever we refer to a manycore system, which we use as a general term in this chapter, we imply that the discussed methods also apply to multicores.

General purpose computing systems – PCs, servers or high performance computing (HPC) clusters – are usually provided with homogeneous manycores based on identical cores, such as AMD's 12-core Magny Cours[3] or Intel's 8-core Nehalem-EX.[4] The particular needs of various embedded application domains, such as mobile communications, IP data plane networking, interactive online gaming, visual computing, automotive control units, medical electronics, and industrial automation are often better addressed by heterogeneous manycore solutions. Such heterogeneous manycores are composed of different types of cores which, by means of their instruction sets and microarchitectural structuring (SIMD, VLIW, super-scalar), are particularly optimized and tailored toward the respective needs of the applications. Examples of heterogeneous multicores include TI's OMAP platform,[5] the Cell BE[6], and the CSX700 from ClearSpeed.[7]

Both homogeneous and heterogeneous manycore systems will coexist in the future for reasons of their particular advantages in specific application domains. However, both classes have a number of individual technical and methodological challenges which, for the time being, throttle their rapid adoption on an even broader scale. An omnipresent technical challenge of processors is absolute and area-density specific electrical power dissipation. As a result of all forms of

[1] Tilera. http://www.tilera.com/.

[2] Intel, Single Chip Cloud Computer. http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[3] The AMD Opteron 6000 Series Platform. http://www.amd.com/us/products/server/processors/6000-series-platform/.

[4] Intel Microarchitecture Codename Nehalem. http://www.intel.com/technology/architecture-silicon/next-gen/.

[5] Texas Instruments, OMAP platform. http://www.ti.com/OMAP_DSPs.

[6] The Cell project at IBM Research. http://www.research.ibm.com/cell/.

[7] ClearSpeed CSX700. http://www.clearspeed.com/products/csx700.php.

environmental and fabrication related variations, as well as transient perturbations from ionizing radiation, reliability exposures due to an increasing sensitivity of MOSFET transistor operation will become equally relevant in the future [4, 17].

In this chapter, we address another key shortcoming of existing manycore architectures and corresponding software development tools. Our primary concern is the limited exploitation of the nominal processing performance by the applications running on the manycore. We see a significant deficiency in today's manycore architectures and tools to allow system developers to efficiently utilize the available resources. The most critical parts are the efficient partitioning of applications into concurrent tasks on the one hand, and the spatial and temporal mapping of those tasks onto processing resources on the other. This involves static analysis or, advantageously, even dynamic runtime balancing, both of which are NP-hard problems.

Today, effective utilization of the available processing resources often depends on the manual skills and talents of the application programmer. This approach, however, is deemed to fail in the long run and is by no means scalable. In consequence, the effective processing performance seen by the application typically lags factors behind the aggregate performance of all processing cores. One might argue that diminishing returns with increasing number of processors in manycores is to be expected from Amdahl's law. However, be reminded that Amdahl's law applies to the degree an individual application can be partitioned into parallel processes. In embedded system domains – such as those listed for heterogeneous architectures above – manycores typically run numerous applications in concurrent fashion. Hence, by assigning different processes to different cores, parallel architectures should substantially improve overall system performance even though individual processes are inherently sequential.

Our proposal to tackle the parallel resource exploitation challenge implies a holistic, cross-layer and inter-disciplinary optimization of application, middleware and architecture aspects of manycore solutions. The central objective is to narrow down the processing efficiency gap from both the software and the hardware side. Thereby, we primarily focus on hardware support techniques to achieve this objective. It should be noted that we do not claim to solve the complex issues presented before. Instead our goal is to raise the awareness in the direction of hardware support for manycore systems.

Concretely speaking, in Sect. 2 we will identify opportunities where application-specific architecture optimizations in heterogeneous network processors (NPs) are beneficial for the future design of homogeneous general purpose manycore systems. Thus, learning from thorough analysis of applications characteristics may also benefit manycore architectures intended for different domains, like supercomputing, if the optimizations are of a generic nature. Networking was used here as one representative example for an application domain with high computational requirements.

The reverse direction, how application-specific manycores can benefit from homogeneous structures used in supercomputing, is covered in Sect. 3. This question implies reassessing methods that have successfully (or unsuccessfully) been applied to HPC and scientific computing over the past decades. Investigations will

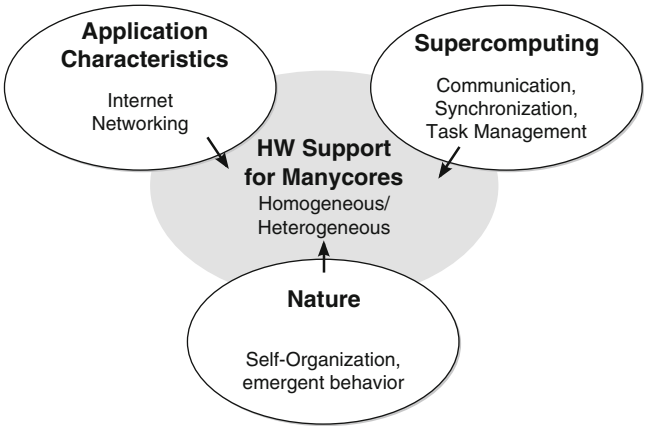


Fig. 3.1 Different processor and nonprocessor related domains contributing to manycore HW support

determine whether and how established methods in HPC can be down-scaled to benefit manycores with their specific dimensions. Finally, by example of swarm intelligence, Sect. 4 will show that bio-inspired principles found in nature (i.e., self-organization and self-adaptation) provide rich opportunities for meaningful adoption in both application-specific and general purpose manycores.

Generally speaking, we provide a set of suggestions for architectural improvements and building blocks that, from our perspective, are candidates for adoption in future manycores to support better exploitation of the available parallel processing resources. Overall, as depicted in Fig. 3.1, we encourage an open exchange of methods and techniques among different manycore application domains and different scientific disciplines – engineering, natural and computer science – for the sake of benefiting manycore performance, power efficiency, and reliability.

3.2 Learning from Network Processing Applications

Network processors (NPs) are high-performance application-specific manycore processors with dedicated hardware enhancements for TCP/IP packet forwarding and advanced networking services. While the high link rates in current networks mandate use of application-specific hardware acceleration for computationally intensive tasks such as address lookups or cryptographic operations, the flexibility requirement to cope with an increasing variety of network protocols and new transmission standards (e.g., moving away from ATM and Sonet/SDH toward Carrier Grade Ethernet) demands disposability of sufficient software-programmable processor resources. The key design challenge of NPs is to find a proper balance between software- and hardware-based processing entities with high overall flexibility and computational density.

3.2.1 Commercial Network Processors

Depending on the specific deployment scenario for NPs inside the network infrastructure (access versus core network), commercial NP vendors offer customized blends of programmable processors and domain-specific hardware accelerators integrated on a single chip. NPs can further be classified into symmetric and pipelined processor clusters, each class coming with its associated programming model: run-to-completion or pipelined. While highest link rate NPs tend toward deeply pipelined architectures (Xelerated[8]), the majority of commercial NP offerings favor the parallel cluster approach as depicted in Fig. 3.2. In a parallel data plane processor cluster, each core is capable of executing the entire set of packet processing functions and may access hardware accelerators to offload computationally intensive subfunctions. This assembly is complemented by a high-capacity on-chip interconnect, on- and off-chip memories and network as well as switch fabric specific I/Os. The cores are programmed according to the the run-to-completion model, i.e., the entire packet processing software can be seen as a single thread being executed on the same core from the beginning to its end. Parallel cores or hardware accelerators within the NP architecture are exploited by assigning arriving packets to different cores. Cores are often multithreaded, which ensures continuation in processing (of different packets) during long latency memory or hardware accelerator accesses. Proper dimensioning of shared communication, memory and hardware accelerator resources is absolutely critical in a parallel manycore cluster architecture, since on-chip interconnect speeds and memory access bandwidths can be as much as 10 times the nominal network speed rate of the NP.

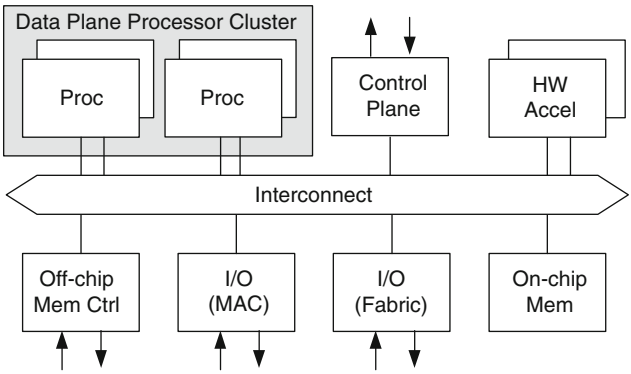


Fig. 3.2 Parallel processor cluster architecture (run-to-completion), e.g., AMCC nP3700, Cisco QFP, Cavium Octeon II

[8] Xelerated. Xelerator X11 Network Processors. http://www.xelerated.com/uploads/files/5.pdf.

3.2.2 Example Networking Applications

The most elementary Internet networking application that must be supported by every router is IP forwarding. Basic IP forwarding is defined in IETF RFC1812 [3] and requires an integrity check of the arriving packet, a next-hop route lookup on the destination address of the packet, decrementing the TTL-field in the header, recalculating the IP checksum and retransmitting the packet on the correct output interface. While the address lookup is often implemented with a dedicated TCAM-based (Ternary Content Addressable Memory) network search engine (NSE),[9] the other packet manipulations are rather straightforward tasks. Usually, commercial NPs perform these tasks on the software-programmable cores. We will show that standard IP forwarding can also be accomplished efficiently with a few hardware building blocks.

In wireless access networks, traffic from several base stations (NodeB) is aggregated toward the radio network controllers (RNC) and mobile switching centers (i.e., SGSN in UMTS) [15]. The transmission protocol architecture of wireless networks specifies complex protocol conversion and adaptation functions that have to be executed in the RNCs for traffic arriving from the NodeBs and leaving toward the SGSN and vice versa (see Fig. 3.3). On the other hand, traffic between adjacent RNCs is simply forwarded. Hence, traffic terminating at an RNC or NodeB requires intensive and flexible processing in software, while forwarding traffic between neighboring RNCs can be processed efficiently with adequate hardware support. Assuming a network topology with eight daisy-chained RNCs aggregating their traffic to a single SGSN, around 85% of the traffic could be handled by hardware forwarding, while only 15% of the traffic would be subject to more demanding protocol conversion processing.

The third networking application example deals with the IPsec security protocol suite [11]. Here, packet classification and security parameter management lean more toward a flexible software realization, while the computationally intensive data manipulations for en- and decryption of packet payloads in real-time is better addressed by dedicated hardware engines. IPsec packet processing would therefore start with analysis of the packet using software routines on a processor core, then handing the packet it then over to a hardware accelerator for en- or decryption before performing the remaining packet processing in software prior to transmission of the packet. If the packet type can be determined near the receive interfaces of the NP, it is possible to direct encrypted packets directly to a hardware accelerator for decryption. Thus the processor will be interrupted only once, after decryption has finished, and the software can perform the remaining protocol processing tasks.

In conclusion, networking applications exhibit a broad mix of tasks with greatly varying processing requirements (hundreds to several thousand operations per packet). Depending on the flexibility (likelihood for a task or protocol to change) and

[9] IDT. Network Search Engines. Product Flyer. http://www.idt.com/products/getDoc.cfm?docID=10154.

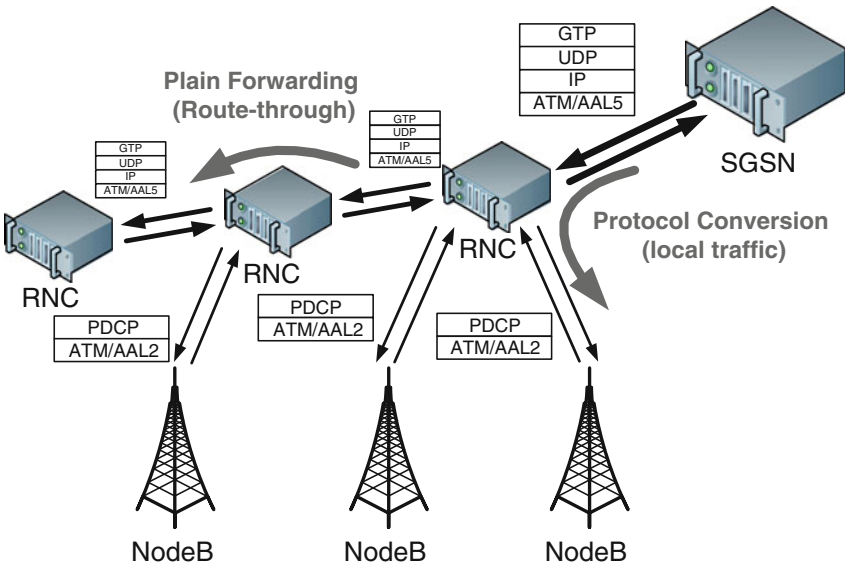


Fig. 3.3 UMTS Backhaul network architecture

processing performance requirements, some tasks are more favorably implemented in software, and some are better suited for an implementation as dedicated hardware accelerators. Irrespective of the form of realization, the sequence in which tasks are traversed within an NP matters. Commercial NP architectures typically pass control of packet processing immediately to a software-based core. After software packet inspection, processing is either completed on the core(s) or processing is interleaved with hardware accelerator calls over the shared communication infrastructure. Each and every packet traverses a core at least once, no matter how simple the packet processing is.

Based on the above analysis, we noticed the need and potential to rethink overall NP structuring. The result, and what insight this new approach bears for other manycore applications, is described in the following two subsections.

3.2.3 The FlexPath NP Approach

A key aspect for improving the performance of NP systems is finding the right mix of dedicated hardware functions and software-programmable resources, and assigning packets to the proper processing instance in an efficient manner. In consequence, we proposed the FlexPath NP architecture [16, 19], with the functional extensions as shown in Fig. 3.4, that achieves performance benefits through the following measures:

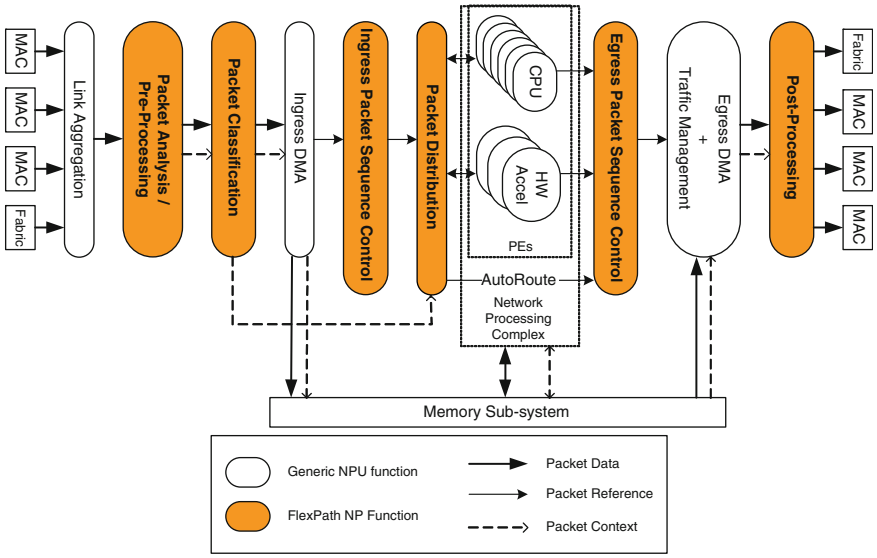


Fig. 3.4 Functional packet flow through a FlexPath NP

- Introduction of hardware-offload units (Pre-Processor, Post-Processor) that are able to relieve the central processor complex from simple, recurring tasks such that the flexibility inherent in the software-programmable resources is not wasted for "routine" tasks.
- The hardware-offload units are able to handle basic forwarding traffic, such that the central processor complex can be completely bypassed for those kinds of packets. This feature is referred to as "AutoRoute" in the context of FlexPath NP.
- The FlexPath NP provides a classification unit at the ingress side to differentiate packets from different networking applications. Thus, packets can be routed along on-chip processing paths (i.e., the precise traversal sequence of the functional units in the NP chip) that are specifically optimized for the various applications. The simplest example is choosing between a path through the central processor cluster for pure software processing and sending the packet over the AutoRoute path for pure hardware processing.
- The classification function is run-time reconfigurable, so that the system can be adapted during runtime to accommodate additional traffic types or react to short-time changes in the application mix. The classification function may then also be reused to support advanced QoS features and implement application-aware load balancing strategies in the NP that further improve the system performance.

To address the problem of reordered packets that arises when packets belonging to the same flow may (potentially) be routed over several paths, we also had to introduce a packet sequence control that resequences out-of-order packets at the egress side of the NP.

Exploiting the hardware offload of pre- and post-processor alone doubles the forwarding performance of the processor cluster. If the AutoRoute path can be chosen for certain traffic types, the processor cluster is completely bypassed and the forwarding performance is greatly increased.

Apart from increasing the forwarding performance, the AutoRoute feature has another notable benefit for the NP system. In Fig. 3.5, we compare the average latencies for packet processing in the programmable cores with the latency on the AutoRoute path through the NP. The system is offered IMIX[10] traffic in two streams, one of which will be forwarded by two PowerPC cores and the other one being AutoRouted. The line rates of both streams are then adjusted to generate different proportions of AutoRoute traffic. The aggregated traffic imposed on the system is shown on the *x*-axis, while the *y*-axis visualizes the measured average packet latency through the system, differentiated for the two processing paths. The packet latencies observed for AutoRoute packets are significantly smaller than those for packets forwarded by the cores in software. When the traffic load on the NP is increased, we are reaching a point when the processor cluster becomes fully utilized and packets start being queued in front of the processor cluster waiting to be processed. Further increasing the input load quickly drives the processor into an overload situation, where all buffers get filled and packets are being dropped.

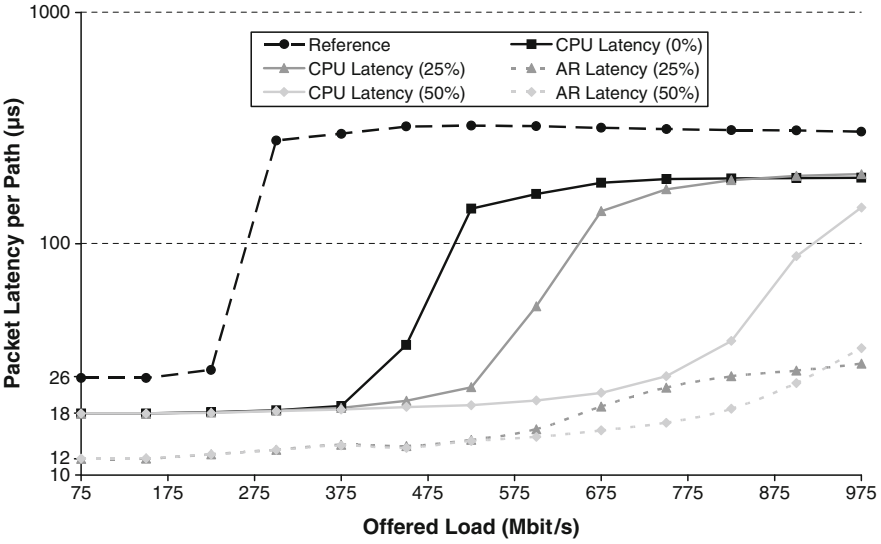


Fig. 3.5 Measurement result CPU/AutoRoute path forwarding latencies for increasing AutoRoute share

[10] Agilent. Mixed Packet Size Throughput. http://advanced.comms.agilent.com/n2x/docs/insight/2001-08/TestingTips/1MxdPktSzThroughput.pdf.

The additional queuing delays add to the core processing latency and deteriorate the performance of the NP for the processor-bound packets. For higher shares of AutoRoute traffic, we can observe that the cutoff point at which the queues in front of the processor complex start being filled is moved further to the right, showing that we are able to process more traffic with the same resources. When the NP's processors are fully overloaded, the packet latency approaches an upper bound determined by the depth of the queues multiplied with the processing latency per packet. However, even when the software processed traffic stream is in overload, the forwarding performance of the AutoRoute packets is not much deteriorated. Hence AutoRoute leads to lower processing latencies that are independent of the software core utilization, and to a higher overall throughput.

We have also performed an intensive investigation on QoS-aware load balancing strategies in the context of FlexPath NP systems. Figure 3.6 shows the simulated packet loss rates of a multicore NP system when imposed with a given backbone traffic trace for an increasing number of cores. All cores perform plain IP forwarding, but different load balancing strategies are used to distribute the arriving packets over the available cores. AHH [10] and HABS [22] refer to two state-of-the-art load balancing schemes. While AHH is an adaptive hashing-based assignment scheme, HABS refines hashing-based load assignment with an additional flow-aware burst shifting algorithm. The improved performance of HABS costs additional implementation effort in the ingress side of the NP.

We have investigated two further load assignment schemes in the context of FlexPath, namely Hash Lookup (HLU) and packet spraying. In contrast to AHH, HLU has a simpler adaptation routine and achieves similar results as AHH. The loss rates are smaller especially for systems with little performance headroom (i.e., five to seven cores in the setup shown in Fig. 3.6). It is important to realize that all load

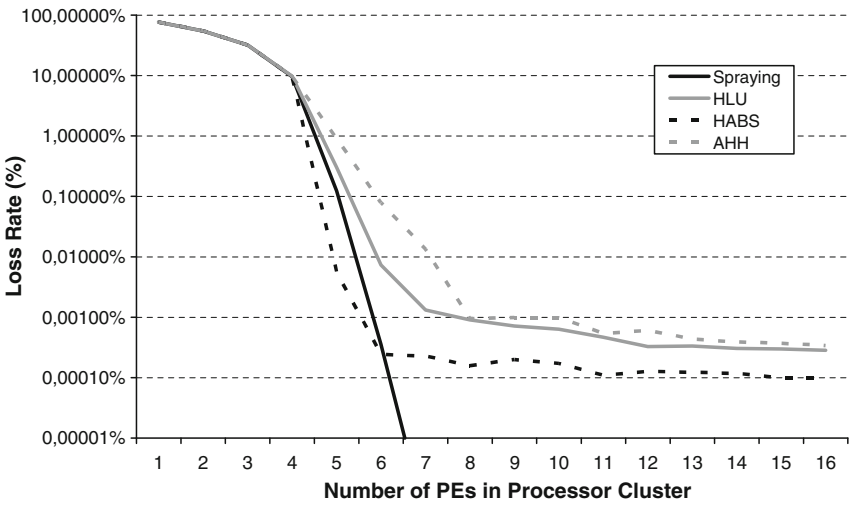


Fig. 3.6 NP packet loss rates for different investigated load balancing schemes

balancing schemes which assign certain flows or flow bundles onto a distinct core reach a minimum packet loss floor. In contrast, packet spraying always assigns incoming packets to the next available core. This leads to a well balanced system and thus minimum packet loss rates. For the given traffic traces, the packet loss rate can be reduced to zero when provisioning more than seven cores in the system for the given traffic trace. However, packet spraying may only be used for stateless traffic types (e.g., IP forwarding), as successive packets belonging to the same connection may be processed by different cores. If the processing depends on a connection state, it is strongly advisable to process an entire stream on the same core. This avoids shared connection states and maintains data consistency. In addition, the packet sequence may become out-of-order, which is solved in the FlexPath NP architecture by resequencing all packets after processing in the Path Control unit. We propose to use packet spraying for all stateless traffic flows, while hash lookup is preferential from the performance and complexity point of view for all stateful traffic flows.

In addition, we have shown [20] that, using the classification capabilities of the FlexPath ingress data path pipeline, QoS features can be applied directly on the incoming traffic stream. This leads to better performance results compared to implementations where the QoS differentiation is performed in software on one of the programmable cores.

As we have seen in the previous paragraphs, significant performance benefits can be achieved with the FlexPath NP architecture by enhancing the NP with additional hardware offload functions. However, these hardware modules consume chip area, which otherwise could be used for additional software programmable cores. To make a fair performance evaluation, it is necessary to compare the consumed chip area for the proposed hardware modules versus the resulting computational power if this chip area were instead dedicated for additional processors.

Figure 3.7 shows the area requirements (measured in FPGA slices and embedded SRAM blocks) of the FlexPath-specific functional modules. In total, the consumed area is 5,721 slices and 22 BlockRAMs, corresponding to 22.6% of the slices and 9.5% of the embedded SRAM blocks of a Xilinx Virtex-4 FX 60 device. This FPGA area could instead be utilized to implement 3.7 MicroBlaze embedded 32bit RISC

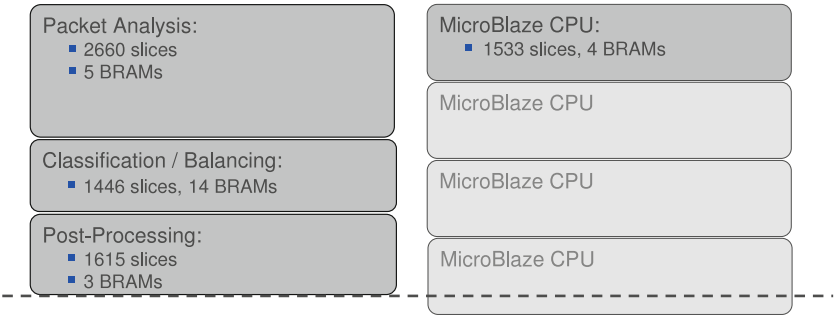


Fig. 3.7 HW enablement overhead of FlexPath NP modules compared to soft core CPU cores

processors, each of which would consume 1533 slices and 4 BlockRAMs in a typical configuration. Next we compare the packet processing performance. The FlexPath hardware pipeline that is used for the AutoRoute feature has a 32bit data path and operates at 100 MHz. It is capable of forwarding packets at a line rate of 3.2 Gbit/s irrespective of the packet length. The MicroBlaze processors would in turn be able to forward packets at a cumulated packet rate of 95 kpps. This results in a data rate of 366 Mbit/s when assuming an average packet length of 481 bytes in the Internet (IMIX). Hence, the area investment for the proposed FlexPath hardware assists are a good investment compared to additional software programmable resources.

3.2.4 What Can Other Manycore Domains Learn from Network Processing?

To open the concepts discussed in this section to other manycore domains, it is necessary to find a suitable generalization of the networking environment. Instead of considering IP packet processing, it is possible to focus the investigations on more general processing requests. Such a processing request might either be an interrupt that requests execution of a certain service routine, a sensor signal that leads to periodic execution of certain functions depending on that signal's value, or arrival of video or data blocks in a video processing or digital signal processing system. Abstractly speaking, it is the arrival of a certain chunk of data that causes execution of a certain piece of code on the manycore system. By analyzing (parts of) the arriving data, the proper action can be determined. In the most general case, one could even think of processing requests as RTOS tasks that contain both the data and the code for processing that data. As we have outlined in the packet processing domain, such processing requests can be categorized with respect to request interdependence (i.e., whether or not two consecutive processing requests may be processed individually without referral to a shared processing state) and with suitability for hardware implementation/offload. This classification of the processing request types, which has to be performed separately for each application domain, decides which and how many of the concepts derived in FlexPath NP are applicable for deployment in the more general manycore system.

- *Identify feasible abstractions which open FlexPath concepts to different application domains:* After generalizing packets to processing requests, corresponding generalizations have to be found for the pre-processing and classification functions of FlexPath. Depending on the application, a suitable processing request parser has to be implemented with a request classifier that can be used later to identify the different request types present in the investigated manycore system.
- *Traverse different cores/coprocessors in an application-specific sequence:* After request classification, we can assign the processing requests to the computing entities in an application-optimized way. In a homogeneous manycore system, it

might be enough to assign the incoming requests to a specific core (or set of cores) that execute the respective application. In a heterogeneous manycore, where processing of at least a part of all applications will be executed by both programmable cores and application-specific accelerators, the decision would include specifying the sequence in which the different involved entities have to be invoked. In both cases, a performance benefit can be achieved by offloading the classification and scheduling task from software into dedicated hardware and using the programmable cores only for the actual computations.

- *Making use of hardware offload where feasible:* In FlexPath, we have demonstrated various levels of hardware offload for different networking applications, but hardware offload is not restricted to the network processing domain. A first level of hardware offload is always achieved by the processing request preprocessing and classification function as described in the previous bullet. If a certain application is best implemented by a combination of software and hardware components – which is often the case in video processing for example – direct assignment of the processing request to the application-specific accelerator and subsequent invocation of the programmable core with the intermediate result is more efficient than calling the accelerator under the control of the core. We have shown that offloading standard packet integrity checks to hardware and only performing the plain routing function in software already doubles the system performance. The highest benefit is of course achieved when the entire processing can be offloaded to hardware (e.g., AutoRoute in FlexPath), when the cores are relieved from executing an entire application class and can focus on the remaining processing request types instead.
- *Generic hardware support for workload balancing:* The differentiated load balancing strategy presented for FlexPath NP can also be transferred to a wider class of manycore architectures. We have shown in our work that workload spraying is an optimal load balancing strategy for independent processing requests, i.e., when processing of each incoming request can be performed independently from prior and/or later requests. If the processing requests involve a shared processing state, it is beneficial to use hashing-based assignment schemes, where all processing requests sharing a common context are dispatched to the same processing resource, which may then hold a local copy of the processing state. This insures correct and consistent processing of the application while avoiding the overhead and performance degradation associated with the deployment of a distributed shared connection state, which has to be locked by every core in the correct sequence. If the manycore system processes a mix of stateful and stateless applications, a combination of the two presented workload balancing strategies can be applied.
- *Directed data preload to local memories:* When a system is scaled towards a manycore architecture, use of local memories is essential to circumvent the scaling problems and bottlenecks of a shared bus and memory infrastructure (see also the requirement for NoCs in Chap. 3.3.1). The classification function determines the type of subsequent processing entities and the sequence in which different processing entities are invoked for the arriving packet or transaction. This information can

also be used for DMA purposes. The data belonging to the current request can be stored in the local memory for the processing instance to which the task has been assigned. In consequence, the core has a faster and more efficient access to the required data and the system processing performance will be increased.

3.3 Learning from HPC and Scientific Computing

For decades, high performance computing (HPC) has developed increasingly sophisticated massively parallel computing systems. Both research and industry are dealing with optimization across all hardware- and software-abstraction layers, including efficient programming models and scalable architectures. With the transformation from huge and complex clusters composed of hundreds of thousands of cores to upcoming manycore systems on-chip, former experience with HPC is a welcome catalyst for the hardware and software ecosystem.

In comparison with HPC, manycore chips run at different time scales, reduced system dimensions, and increased bandwidth. These operating conditions have an impact on the adaptation of HPC concepts to manycore systems on-chip, where impact depends on the requirements of the concepts. Some concepts may easily be adapted, including some that may not have been practicle in HPC systems.

In this section, we will discuss the adaptability of HPC concepts to novel manycore on-chip architectures by examining certain examples for this adaptation. Network-on-chip (NoC) is the adaption of packet-based routing networks for on-chip communication. With changing conditions due to the transition from supercomputing to on-chip manycore, novel opportunities show up, for example hierarchical NoC. For *Task Management*, the HPC process and job management is not easily downscaled to changed conditions in multicore architectures. Instead, approaches with different granularity are necessary. Finally, separate *Synchronization Subsystems* are discussed. Similar systems have previously been discussed for supercomputing architectures, but the changed conditions rejuvenate this topic.

Another challenge, which is not discussed here, is the integration of novel manycore chips in HPC architectures. The integration of such systems requires multilevel approaches.

3.3.1 Hierarchical Multi-Topology Networks-on-Chip

Flat, two-dimensional-mesh network on chip (NoC) is the predominant interconnect structure for current homogeneous manycore architectures such as Tilera's Tile64 [24] with 64 cores and Intel's Teraflop [7] with 80 cores. The mesh topology is well suited for uniform traffic among cores (i.e., each core sends an equal amount of traffic with equal probability to all other cores).

However, future manycores will no longer be entirely homogeneous. Instead, different kinds of hardware accelerator resources will be integrated on chip. For example, Intel's general purpose Westmere processors contain a dedicated graphics core. In the future, heterogeneous manycores will contain different kinds of accelerator modules to assist general purpose cores in executing certain functionalities, such as en-/decryption, graphics, etc. In heterogeneous multiprocessors, the traffic will no longer be as uniformly distributed as in homogeneous multicore systems. Traffic flows to and from shared hardware accelerator modules can already be estimated at design time. Consequently, it will be possible to optimize and adapt the interconnect infrastructure according to this knowledge.

Mesh topology is not necessarily the best choice for heterogeneous manycores from a global perspective. Certain portions of the manycore might be better served with other interconnect topologies which exhibit lower latencies, higher throughput or come at lower areal cost. For a group of cores that form a processing pipeline, a ring topology might be better suited due to its lower chip area and higher throughput, enabled by higher router clock rates [13]. An example is IBM's cell processor [2], which uses a high speed ring interconnect to allow communication between the SPE processing units. Other subportions of a manycore may not have high throughput but low latency requirements. Such subportions are better interconnected with crossbar switches, star topology based NoCs or even shared buses. These multifacet requirements call for a multitopology interconnect approach for heterogeneous manycore systems.

Another issue in systems on-chip is traffic to and from shared resources. Shared resources typically attract an over proportional amount of traffic from many cores of the system or are the source of traffic toward all these cores. Examples include shared on-chip memories, transactional memory, hardware accelerators, and IOs. For certain accelerator cores or IOs, a possible option to optimize the communication cost of traffic from distant cores would be to instantiate multiple instances distributed over the whole system. However, this solution is not applicable for shared memories that, by definition, cannot be partitioned or distributed. For IOs, this approach might also be prohibitive in most cases due to pin count limitations of the chip package. To optimize the communication performance and decrease communication cost, the communication infrastructure should be optimized for traffic to and from the shared resources. By reducing the network distance between the communication partners, not only the performance but also the latency and energy consumption can be decreased.

Hierarchical multi-topology NoCs [14], as shown in Fig. 3.8, address both problems introduced above. Hierarchical networks allow the efficient adaptation of the subnetwork's communication infrastructure towards bandwidth and latency requirements. In this way, a subnetwork can either be based on a mesh topology if it requires high bandwidth, or on a ring network if the communication pattern is predefined by the application running on the cores. If allowed by the aggregate bandwidth requirements of a group of cores, even a bus can be considered for connecting them. However, the partitioning into subnetworks is not advantageous for all traffic flows. For traffic traveling from one subnetwork to another, the hop

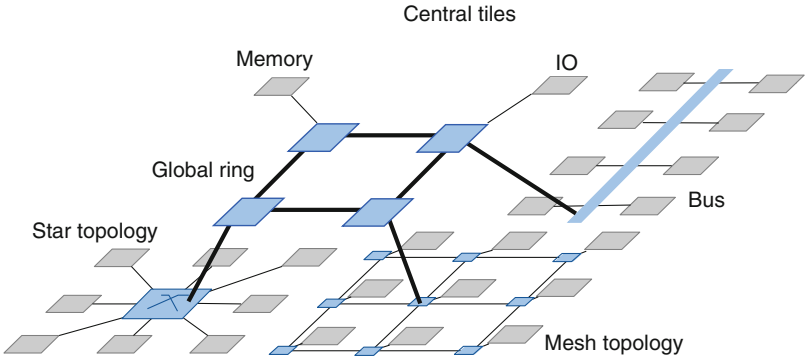


Fig. 3.8 Hierarchical NoC

count is generally lower in a full 2D mesh that is not partitioned. In case the share of this class of traffic is very high, the hierarchical network design can also be realized without partitioning the lowest hierarchy level into subnetworks.

The hierarchical network concept also allows an efficient connection of shared resources by connecting them to the upper hierarchy level(s). Since the communication structure of the upper hierarchy connects all the subnetworks, the cores connected in this hierarchy can be efficiently accessed by all other cores.

In the following, the advantages of hierarchical networks in accessing shared resources will be backed up by simulations. Two SoCs based on hierarchical network architectures are compared to one based on a conventional 2D mesh. The simulated systems consist of processing cores, with the exception of two memories, that represent the shared resources. In Fig. 3.9, the number of tiles (cores) of the systems is denoted by $n \times n$. In the conventional mesh, the central tiles (i.e., shared resources) are connected in the middle of two facing borders of the network. The hierarchical network architectures are realized according to the principle shown in Fig. 3.8. A global ring connects the subnetworks and the two central tiles. However, the simulated systems have four subnetworks which are all based on mesh topologies. In total, these subnetworks have the same number of cores as the 2D mesh based SoC. In contrast to the hierarchical NoC in Fig. 3.8, the routers of the global ring connecting the subnetworks are not implemented separately, but are integrated into submesh routers (denoted as **hMiR** in the following). The networks are built from routers based on an input and output buffered architecture and use wormhole forwarding.

For the evaluation, three types of traffic are used t0, t1, and t2. Traffic t2 is targeted to the shared resources and its rate is chosen in such a way as to fully load the network interface of these tiles. Traffic t0 is traffic that stays within a subnetwork and traffic t1 consists of packets with a destination in another subnetwork. The rate of traffic (t0+t1) is increased during the simulations, with the relation of t0/t1 being 3/1. The discrimination between traffic t0 and t1 in the mesh is done by a maximum hop count for packets of traffic t0. This hop count threshold is set in such a way that the average hop count of t0 is equal in the mesh and the hierarchical networks.

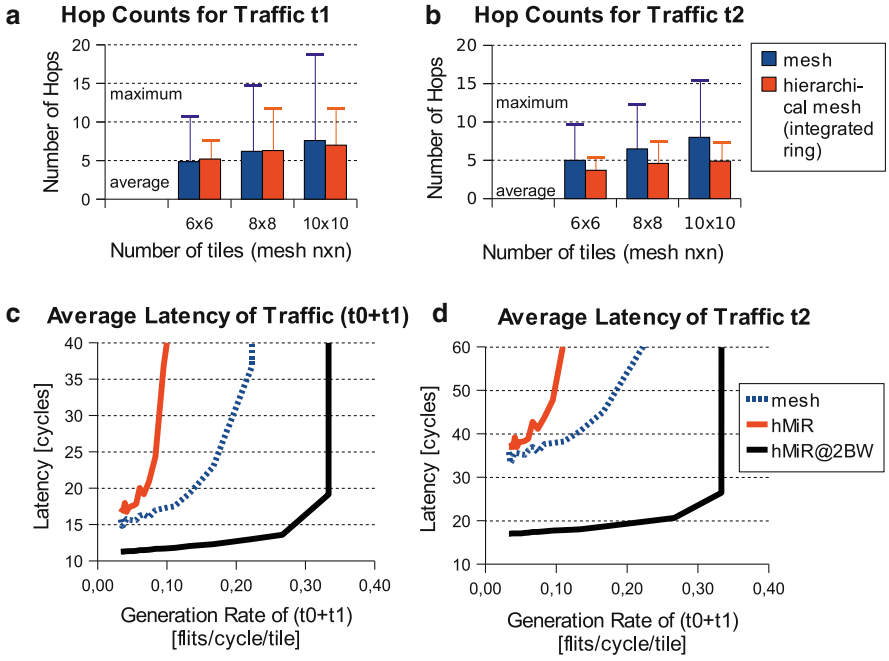


Fig. 3.9 Hop counts and latencies for different realizations of hierarchical networks

The hop counts of traffic t1 can be seen in Fig. 3.9a for different network sizes (6×6 up to 10×10). Although our proposed hierarchical topology reduces the maximum hop counts significantly, the average hop counts are only reduced for larger networks. However, for traffic of class t2 Fig. 3.9b shows a reduction in maximum as well as average hop counts for the hierarchical approach, especially for networks with a larger number of tiles.

Figure 3.9 also shows the latency measurements, although only for networks of size 6×6. By comparing the latencies of the network hMiR and the 2D mesh in Fig. 3.9d, it becomes obvious that the network hMiR cannot profit from a reduction of average hop counts by 20%. This is due to the low bandwidth of hMiR's global ring, which is not able to cope with traffic t2 and t1 aggregating in the upper hierarchy level. Both latency diagrams (Fig. 3.9c, d) show that the network hMiR has only a very low throughput for the applied traffic (3/4 t0, 1/4 t1 and t2). The result of increasing the global ring's bandwidth by doubling the ring routers' clock (denoted as **hMiR@2BW**) is a significant improvement in latency as well as in network throughput. This increase in network throughput despite the use of wormhole switching is only possible due to virtual channels in the global ring. These virtual channels are actually implemented to break the routing cycle in the channel dependency graph, and thus prevent deadlocks.

Simulation results show that hierarchical networks can considerably lower the communication costs (e.g., hop count, energy consumption, latency) of the traffic.

In particular, this is true for traffic to and from shared resources (t2), but also applies to inter-core traffic (t0 and t1). However, the effect of reduced hop counts on latencies are minimal when wormhole forwarding is used. Nevertheless, the challenge in the design of hierarchical networks is to equip the upper hierarchy with enough bandwidth to cope with the aggregating traffic of the types t1 and t2. In this way, the performance of the proposed network architectures relies greatly on the realizable routers and their throughput.

3.3.2 Task Management

The run-time management of processing resources is a crucial part of massively parallel computing. In HPC environments, the job and process management was controlled by software. With the introduction of chip multithreading, the scheduling of work portions has been addressed by modern architectures. The granularity, i.e., the number of instructions, of such work portions strongly depends on the overhead of thread management and scheduling operations. Context saving and the migration of tasks differ significantly from supercomputing environments in the granularity of such "tasks." Due to the changed conditions of manycores, the granularity of tasks can be significantly reduced. The overhead introduced can be scaled down to approaches where tasks only consist of a few instructions. Reduced latency and increased bandwidth allow for sophisticated chip-wide task management approaches.

As a result, novel approaches aim at different granularities and exhaust the freedom given by on-chip constraints. Examples for such novel approaches are discussed in the following.

The *Carbon* approach [12] is a hardware support for dynamic thread scheduling and focuses on small threads. Even optimized software schedulers can hardly cope with an increasing amount of cores. Task queues become a crucial part of a design, so that in this approach, the queues and their management are implemented in hardware. Low overhead task queues are distributed hierarchically on a chip multiprocessor: one global task unit stores hardware thread contexts, while local task units manage the tasks executed on the cores and allow for prefetching of contexts. The instruction set of the processor cores is extended with specific instructions for the queuing and dequeuing of tasks. For RMS (Recognition, Mining, and Synthesis) benchmarks, the technology shows improvements of up to 109% compared to software implementations and nearly reaches the optimal (zero-overhead) case.

Another approach to the changed conditions of novel manycore architectures is the *Self-adaptive Virtual Processor* (SVP) as proposed by [9]. Here, so-called microthreads execute tasks in a fine-grained manner. The SVP can be "considered an operating system, implemented in a core's ISA" [9, p. 247]. Thread families, with those microthreads being in the order of few instructions, are run on this specialized architecture, where the processor pipeline is augmented with special components to handle the execution of up to 256 threads in parallel and provides an

additional mutex infrastructure. The approach proposes the usage of this specialized processor in a ring with local communication. Results show that the approach can gain good scaling even for memory intensive applications.

CAPSULE [21] is another approach that introduces processor extensions for hardware-assisted dynamic multithreading. In contrast to SVP, CAPSULE utilizes standard threads and gives means to augment them with the ability to allocate additional resources. A thread can spawn another thread, and the decision of the target core is based on hardware probes of specific performance characteristics. Although this approach shows promising results, it currently only focuses on single-core support.

Similar to this, hardware probes also control the concept of *Invasive Computing* [23]. In addition to allowing threads to spawn or allocate communication resources based on hardware probes, the approach introduces a new paradigm, the *resource-aware invasive computing*. By extending prevailing programming models with primitives for invading and retreating other resources, the approach targets the self-organizing execution of standard hardware with specific extensions or tightly coupled processor arrays. The approach comprises the hardware architecture, programming models, and the runtime system. Since the approach is still in its proposal-state, results cannot yet validate this promising approach for chip multiprocessors.

3.3.3 Synchronization Subsystem

In addition to pure data communication, synchronization is an important element of concurrent programming. While the utilization of the increased data communication bandwidth and latency has been discussed before in the context of NOC, such synchronization has strong demands on the communication latency.

Prevailing approaches in HPC had to cope with the more complex conditions for communication, but novel manycore architectures allow for sophisticated synchronization subsystems. Separation of inter-processor, memory, and interrupt synchronization is generally a powerful design concept to scale system-level data throughput and bound data access latencies.

IBM Blue Gene [1] serves as an example in the HPC domain: it provides additional networks for synchronization and special data operations. The global interrupt network supports low latency inter-process synchronization across the entire CPU cluster, while the global collective network supports special collective data operations. Although promising approaches such as those in Blue Gene exist, such advanced networks have not found their way into many HPC systems.

In our opinion, special awareness is required for low overhead barriers among subclusters and their application to MPSoC. The approach of separation of concerns becomes relevant with the changed conditions of manycore systems. Examples for such separate networks are depicted in Fig. 3.10, which can be utilized to allow fast

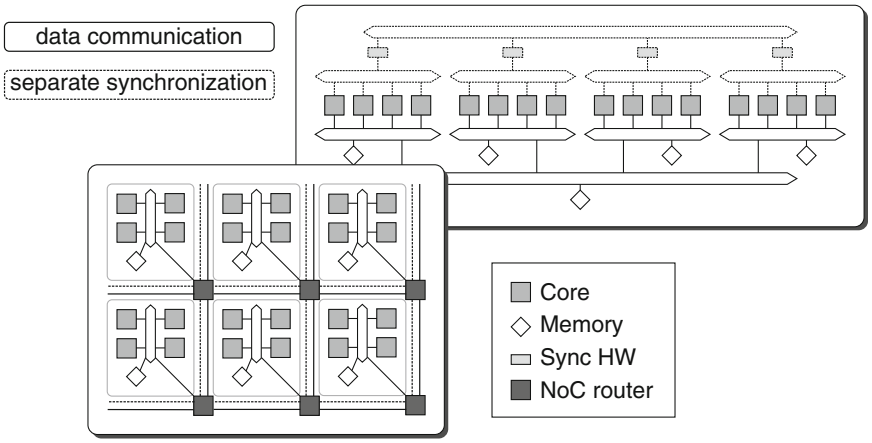


Fig. 3.10 Separate infrastructure for synchronization comes to focus with changed constraints in multicore systems. The *dashed* elements sketch approaches for a separate synchronization infrastructure

and low-latency-connections for synchronization and collective data operations. Current architectures such as Tilera manycores [11] are the first to utilize such separate systems, and we consider this as an important research topic for future manycores of hundreds to thousands of loosely-coupled cores on a chip.

3.3.4 What Can Other Manycore Domains Learn from Supercomputing?

With the changed boundary conditions we have identified different methods of transferring concepts from supercomputing to novel manycore on-chip systems:

- *Transformation of concepts to changed conditions:* Many concepts can be changed with respect to their parameters. For example, the implementation of a message passing protocol is adaptable with different buffer sizes, reduced packet sizes, operation set modifications, etc.
- *Changed conditions have a significant effect on adaption:* Other supercomputing concepts are more heavily impacted by the changed conditions in manycore systems. For example, the utilization of local memories depends on memory sizes, latencies, etc. Despite being used by the most recent IBM supercomputing architecture BlueGene/L, the on-chip instantiation of various communication systems is a topic of its own. Another important topic is task/thread management support. Here, management overhead is significantly reduced and approaches such as hardware-assisted scheduling become relevant.

[11] Tilera. http://www.tilera.com/.

- *Concepts that are not feasible in supercomputing:* In some cases, the overhead or cost of concepts does not allow for their use. An example is a separate synchronization subsystem. Other advanced technologies, such as Transactional Memory, are becoming possible to implement now.

Concretely speaking, the following concepts show this adaption of supercomputing methods:

- *Hierarchical multitopology NoC concept:* The multitopology approach allows individual and optimal adaptation of the on-chip interconnect structure for each subcluster of the system. Combining several multitopology clusters into a hierarchical network enables the efficient connection of shared resources. Both aspects optimize the communication cost and the performance of a manycore system.
- *Task management:* The granularity of tasks changed from complex jobs and processes in supercomputing down to light weight threads, or even a few instructions, in manycores. In consequence, software scheduling overheads would increasingly affect overall system performance. Hardware supported scheduling in novel manycore systems can and shall be realized with significantly reduced overhead costs.

3.4 Learning from Bio-Inspired, Self-Organizing Systems in Nature

While the previous two sections focused on the cross-architectural exchange of techniques between homogeneous and heterogeneous manycores and, thus, derived means for the hardware support of MPSoC out of the native domain of processor architecture, this section seeks inspiration for unconventional and new concepts by looking at nontechnical systems. In essence, as stated in the introduction, improving resource utilization in manycore systems boils down to a complexity problem, where parallel processes should be mapped in an optimal fashion onto several processor cores. In nature, swarm communities (such as ant colonies, insect swarms, herds of land animals, or fish schools) exhibit complex collective behaviors for purposes such as, e.g., shortest path routing or predator protection based on fully decentralized, self-organized control. Hence, if self-organization is an adequate means to cope with complexity in natural (many entities) systems, how can it be applied advantageously to technical manycores?

3.4.1 Collective Behavior of Entities in Natural and Technical Systems

In the following we use a school of fish as an analogy for a "manycore system of nature." We assume that an individual fish corresponds to a single core, and the behavior exhibited by the entire fish school corresponds to the behavior of an

application running on the manycore. The application level behavior of a manycore processor is the result of cooperative interactions among all cores. In nature, the complex three-dimensional formation of a fish school and the collective movement of the entire community emerges from the individual behavior of each fish. Behavioral biologists have discovered that all fish follow a simple and identical set of elementary rules [8], summarized in conjunction with Fig. 3.11. Depending on the distance d between a particular fish and its nearest companion in the sector of sight, the fish either shows a repulsive action to not run into the neighbor ($d < R_r$), is attracted by the companion if the next neighbor is relatively far away ($R_p < d < R_a$), or aligns in parallel to the companion if the neighbor is in a medium distance ($R_r < d < R_p$). These simple, local rules and behaviors at the individual fish level emerge into a sophisticated behavior at the fish school level. Self-organizing, emergent behavior is the consequence of hidden causal relationships among the behavior of the individuals within their community [6]. A prerequisite for self-organization is the existence of a population of interacting system constituents (i.e., fish or cores), and a higher-layer, hierarchical structuring (i.e., fish school or manycore) at which the system-level behavior is observed (see Fig. 3.12).

The fish school example illustrates the potential of exploiting self-organization or emergence – complex systems can be built from a community of individuals that execute only simple tasks. Would it not be intriguing to compose software applications for manycores from simple tasks whose behavior collectively results in the desired system function? The caveat is that, up to now, scientists have not found a calculus or systematic approach to reliably forecast what system level behavior will emerge from what component level rules and actions. Nor can we deterministically tackle the inverse problem – what local behavior would be needed to obtain a certain system level behavior. Emergence may even result in chaotic system level behavior. Hence, when applied to technical systems, it is essential to find ways to meaningfully control emergence [18].

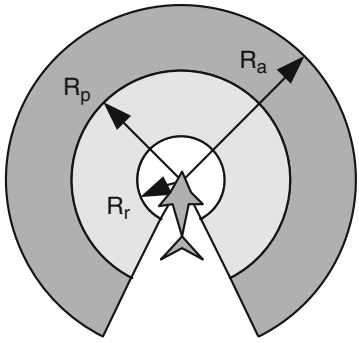


Fig. 3.11 The reactions of an individual fish depend on its proximity to neighboring companions

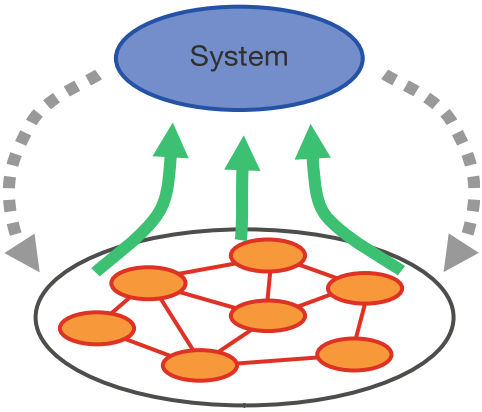


Fig. 3.12 Emergent behavior manifests itself at the system level as a result of collaborative interactions between individuals at the component level

At this point, we will go into a little more detail about the reasoning behind making a processor core the technical equivalent of a "fish." While a large number of small entities allows for very simple individual behavior, adding self-adaptive mechanisms to many entities can become prohibitive in cost. For example, the smallest entity in an integrated system could be considered a single transistor. However, even very simple self-adaptive mechanisms would require rule evaluation logic containing hundreds to thousands of transistors, clearly making the overheads unacceptable. Coming from the opposite direction, a "school" of entities can be considered equivalent to a complete MPSoC. Such a system is composed of multiple component modules, "types of fish," such as memories, interconnects, processor cores etc., generally referred to as IP cores in MPSoC design. Each of these cores is large enough such that reasonably complex self-adaptive mechanisms can be added with acceptable overheads, while sufficient components exist to allow for emerging system-wide behaviors.

3.4.2 Technical Realization of Self-Adaptive IP Cores

We now turn to the problem of choosing an approach for making MPSoC cores self-adaptive. One way would be to redesign all cores from scratch with self-adaptive properties in mind, which would be very costly and time-consuming. Instead, it is preferable to reuse existing IP libraries and extend these with self-adaptive concepts. For this, it is necessary to monitor the behavior of the underlying component, and to be able to effect changes (actions) in the component's operating parameters. For example, it should be possible to monitor a component's utilization and effect appropriate changes in the component's frequency parameter.

In the following, we will assume that various monitor and actuation interfaces have already been integrated into each component. We focus instead on the decision system needed to determine an appropriate action given a certain operating

condition reported by the monitors [26]. In accordance with the simple rules governing a fish's movement within its school, the decision system of each MPSoC component consists of individual rules that identify actions to be taken in specific situations. At its simplest, every rule contains a condition and an action. When the condition matches the incoming monitor signals, the associated action is performed.

Rather than relying on static rules that were specified by the designer, it is preferable for a system to learn which rules are best, and to create new rules that can cope with situations not previously specified. This makes it unnecessary during design time to predict all possible situations the system may encounter. It also allows the system to adjust to unpredictable circumstances, such as manufacturing defects affecting only a certain chip, or different environmental conditions under which otherwise identical chips are deployed.

In addition to a condition and an action, the rules governing the behavior of a technical system therefore also contains a measure of the rule's fitness. The fitness is an indication of how well the rule has performed in the past, simultaneously providing a prediction of how well the rule will perform in similar situations in the future. This prediction is based on a reward returned by the system after a rule has been executed. If execution of the rule causes improvements to the system's operating state, a large reward is returned. If the rule shows no improvement or even harms the system, a small or no reward is returned.

To compare the state of the system before and after a rule has been applied, a function (henceforth called the objective function) must be defined that can be used to quantify how well the system is operating at any point in time. The best possible operation of a system is achieved when the objective function returns some optimal value. To simplify calculations, we choose the objective function such that it returns zero when the optimal system state is reached. The larger the value returned by the objective function, the further away from the optimal state the system is operating. The objective function indicates an overall system state to avoid local optimization. For example, while a reduction in frequency may appear beneficial locally (lower power consumption), it could actually be detrimental to the system as a whole if the system is no longer capable of keeping up with the workload.

As shown in Fig. 3.13, the reward R is calculated by comparing the value of the objective function O before (O_{T-1}) and after (O_T) the rule has been applied. If the

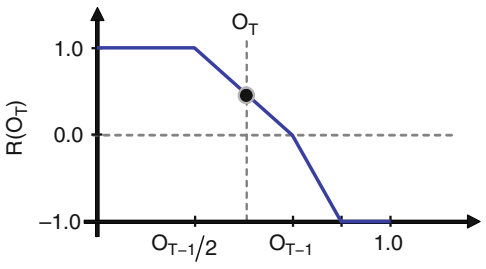


Fig. 3.13 Reward calculation from value of objective function before (O_{T-1}) and after (O_T) application of rule

value increases ($O_T > O_{T-1}$), indicating that the system state has moved further away from the optimum, a negative reward is returned. If the value has decreased ($O_T < O_{T-1}$), a positive reward is returned. The reward is factored into the fitness F of a rule using a running average,

$$F_{new} = \beta \cdot reward + (1 - \beta) \cdot F_{old},$$

where β is the learning coefficient indicating the rate at which learning should occur. A value of β close to 1.0 allows for faster learning, but is also more sensitive to fluctuations in the reward signal, which can occur as a result of noise on the monitor signals feeding the objective function.

Once the fitness of a rule has been calculated over a sufficient number (roughly $1/\beta$) of trials, it can be used to decide which rule should be chosen given multiple rules matching a given monitor state. One possibility is to ignore the rule fitness completely, randomly choosing any rule from the set of matching rules. While this maximizes the exploration of rules to determine how well each performs, it makes no use of the learned fitness knowledge. At the opposite extreme, the one rule with the highest fitness could always be chosen. This would maximize the predicted reward of the system, but might never explore undervalued actions that have the potential to lead to even better behavior. A third option acts as a compromise between the above two methods, and uses each rule's fitness as a weight to determine its probability of selection. This leads to more frequent execution of the actions proposed by rules with a high fitness, but does not entirely prevent exploration of rules with a low fitness.

In the following, the self-adaptation and learning concepts introduced above will be applied to a multicore networking application [25], a block diagram of which is shown in Fig. 3.14. The system consists of an Ethernet MAC, memory controller, and multiple processing cores (Core1-Core3). Incoming packets are stored by the MAC in the memory, from where the cores are able to fetch the packets and process them through several tasks:

Task 1: Accept packet and configure MAC for further reception
Task 2: Process packet header and determine payload processing subtask
Task 3.n: Perform one of N payload processing subtasks
Task 4: Reorder packets for in-order transmission
Task 5: Set up Ethernet MAC to transmit the packet

At system startup, all tasks are scheduled to run on a single core. The objective of the core's decision system is then to autonomically distribute the tasks evenly and power efficiently across all processing elements. To accomplish this, each core's decision system has access to two local monitor signals: the core's current operating frequency and its utilization. In addition, a monitor signal is available that indicates how the workload of the core compares with the workload of other cores in the system. Workload is simply the product of frequency and utilization, and indicates the number of cycles actually spent processing data every second.

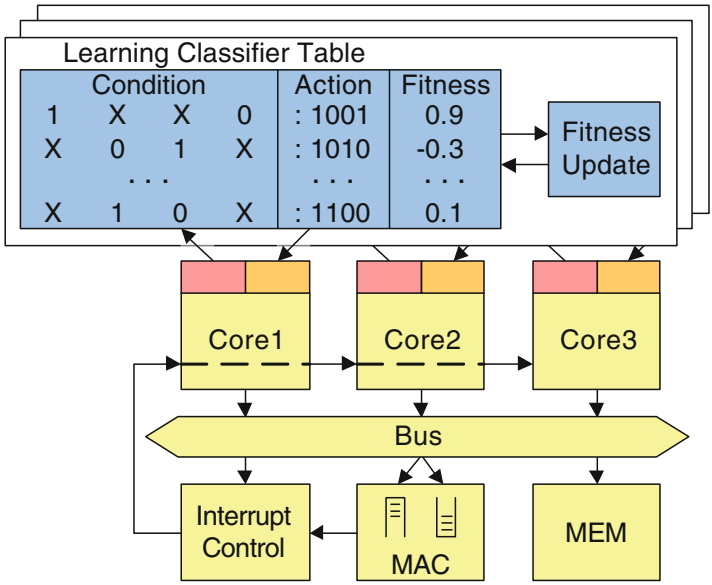


Fig. 3.14 Block diagram of a multicore networking application

Two actions are available to each core: Adjustment of the core's operating frequency and migration of one of its tasks to another core. Should the workload of a core become too high, this allows for two distinct optimization methods; either increase the frequency to cope with the workload, or migrate a task to another core to reduce the workload. Whereas in a classical system the decision of which optimization method to choose would have had to be made by the designer at design time, the decision system can learn which optimization strategy yields the best results under different operating conditions.

To transform the above stated objective function into a formal, mathematical expression useable for construction of appropriate combinatorial logic, we introduce the following delta values that incorporate available monitor information to indicate how far various design parameters deviate from their optimum value:

$$\delta_{frequency} \propto frequency$$
$$\delta_{utilization} \propto (100\% - utilization)$$
$$\delta_{workload} \propto |workload_{local} - workload_{average}|$$

For the networking application presented here, we would like the frequency (and voltage, which is assumed to scale accordingly with frequency) of each core to be as low as possible to reduce power consumption. The utilization of each core should be high to avoid wasted processing cycles, and the workload of all three cores should be similar to avoid inconsistent aging effects or temperature hot spots. Depending on the designer's optimization goals and the available monitor signals, other delta values may be chosen.

Once defined, the delta values are combined into the system-wide objective function O. First, an objective function for each core is created by weighting the various delta values. A core's objective function for the delta values given above would be:

$$O_{core} = w_1 \cdot \delta_{frequency} + w_2 \cdot \delta_{utilization} + w_3 \cdot \delta_{workload}$$

The weights can be chosen based on the optimization goal deemed most important by the designer. For the results presented below, each delta function was weighted equally. The system-wide objective function is then simply a weighted average of the objective functions of each component in the system. Again, the weights are chosen based on design goals, here the objective functions of the three cores are weighted equally.

Figure 3.15 shows the autonomic optimization of frequency and task distribution over the three processing cores in the system. As mentioned, all tasks are initially executed by core 1, causing an initial, dramatic increase in the frequency of that core to cope with the tremendous workload. As the workload is distributed more evenly across the cores, the frequencies of all three cores stabilize to a similar, low value. The resulting average packet latency shown in Fig. 3.15 also varies dramatically while the system is being optimized, but stabilizes once an appropriate frequency and task distribution have been found. Note that the latency is considered neither as a monitor signal nor as part of the system's objective function. Despite this, the autonomic system optimizes the system such that the latency stabilizes to a reasonably low value.

3.4.3 What Can Manycore Domains Learn from Nature?

- *Delegate decisions from design to run time:* By allowing the decision system to make certain decisions at run time, the designer no longer has to consider and predict all aspects of system behavior. Not only does this alleviate the burden on the designer, but when performing decisions at run time, much more information about the actual application being executed and the system's operating environment is available. In contemporary designs, the designer has to guess these operating conditions during design time.
- *Accept low overheads to improve reliability, performance, and power consumption:* The continuing increase in chip capacity results in an overabundance of resources, which are often used simply by replicating components. Unfortunately, it is difficult for the application developer to actually make use of such massively parallel architectures. By utilizing some of the extra area (the rule-based decision system discussed above requires roughly 5% of the resources of a Leon3 processor core) to add bio-inspired principles, other resources can potentially be used much more efficiently by the designer.

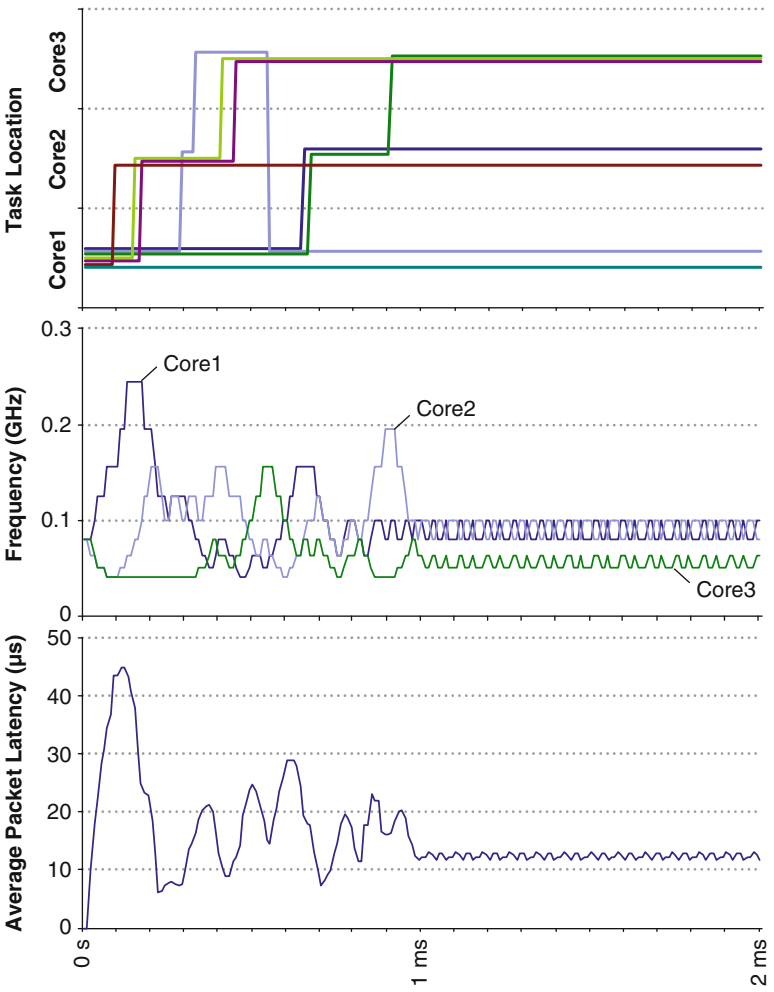


Fig. 3.15 Autonomic task migration, frequency adjustment, and resulting packet latency

- *Extend rather than replace existing IP cores:* Given monitor and actuator interfaces, bio-inspired concepts can be added to a system without requiring completely new IP. Adding monitors and actuators can often be done with minor changes to existing cores.
- *Bio-inspired concepts are applicable to any system component:* Although we have presented a decision system applied only to the processing cores of a system, similar approaches can be used to optimize other system components, such as interconnect, memory, hardware accelerators, or I/O.
- *Not all aspects of self-adaptive systems are fully explored:* Despite the benefits of bio-inspired systems, several challenges must still be faced. First and

foremost, by delegating design decisions into the hands of the system at run time, it is difficult to guarantee reliable operation. In addition, although the choice of a per-component distributed decision system is good from a scalability standpoint, the many independent decisions potentially made simultaneously across the system could lead to instabilities and oscillating behaviors. Due to the penalties associated with parameter and behavioral adjustments, it is important to ensure that even a widely distributed decision system will find a stable operating point. On the other hand, open questions such as these make bio-inspired systems a highly interesting topic of research, with initial results demonstrating that they are not only feasible, but also have the potential to enormously simplify the design of complex, manycore system architectures.

3.5 Summary and Conclusions

State-of-the-art CMOS technology enables the integration of more than a hundred programmable processing cores on a single chip. The efficient utilization of this huge amount of computing performance – for the time being – predominantly depends on the individual skills of application programmers. Today's manycore architectures and programming tools provide insufficient support for the systematic exploitation of massively parallel resources. This chapter cannot deliver a "silver bullet" to this grand challenge either. However, it attempts to provision examples for improvements in form of generic hardware support building blocks for both homogeneous and heterogeneous manycore platforms. These hardware support building blocks are the result of (1) an in-depth analysis of manycore applications, (2) successful and unsuccessful approaches deployed or dismissed in supercomputing environments, and (3) the adoption of bio-inspired principles found in nature, such as collective emergent behavior or self-organizing swarms.

We have shown that significant improvements of performance and resource utilization can be achieved by including specific hardware extensions in manycore architectures. For example, if a HW accelerated classifier analyzes processing requests of a manycore system before programmable cores take control over them, the optimum sequence for traversing different subfunctions in the course of overall processing can be determined and the processing requests can immediately be dispatched to proper resources.

Similarly, generic functions for synchronization of tasks, which is critical in massively parallel architectures, can be moved to optimized accelerators. All of these measures move utilization support functions from the programmable cores into specialized hardware assists.

Nevertheless, it is necessary to ensure that the overhead for adding such hardware extensions is viable from an economic perspective. This means that the overall benefit for the manycore due to the hardware support building blocks must be higher than using the associated chip area for further programmable cores. For application specific manycores, it is straight forward to evaluate what type of

support module may be profitable when composing the hardware architecture. In the case of general purpose manycore architectures, the major challenge for such a hardware-based support infrastructure is to find an appropriate mix of generic extensions that are useful for a wide range of applications.

Acknowledgements Particular thanks go to the German Research Foundation (DFG), the State of Bavaria and Infineon Technologies for supporting our work as part of the Priority Programmes "1148: Reconfigurable Computing" and "1183: Organic Computing", the "Munich Centre for Advanced Computing" (Project B4, MAPCO) and the BMBF Collaborative industry project "RapidMPSoC" (grant BMBF 01M3085).

References

1. N.R. Adiga et al. An overview of the BlueGene/L Supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press
2. T.W. Ainsworth and T.M. Pinkston. On Characterizing Performance of the Cell Broadband Engine Element Interconnect Bus. *Networks-on-Chip, 2007. First International Symposium on NOCS 2007*, pages 18–29, 7–9 May 2007
3. F. Baker, Cisco Systems. Requirements for IP version 4 routers, IETF RFC 1812. http://tools.ietf.org/html/rfc1812, 1995
4. S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. pages 338–342, 2003
5. D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, W. Yoder, et al. Scaling to the End of Silicon with EDGE Architectures. *Computer*, pages 44–55, 2004
6. J. Fromm. Emergence of Complexity. Kassel University Press, Kassel, 2004
7. Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor. *IEEE Micro*, pages 51–61, 2007
8. Y. Inada and K. Kawachi. Order and Flexibility in the Motion of Fish Schools. *Journal of Theoretical Biology*, pages 371–387, 2002
9. C. Jesshope, M. Lankamp, and L. Zhang. Evaluating CMPs and Their Memory Architecture. In M. Berekovic, C. Muller-Schoer, C. Hochberger, and S. Wong, editors, *Proc. Architecture of Computing Systems*, pages 246–257, 2009
10. L. Kencl. Load Sharing for Multiprocessor Network Nodes. Dissertation, EPFL, Lausanne, Switzerland, 2003
11. S. Kent et al., BBN Technologies. Security Architecture for the Internet Protocol, IETF RFC 4301. http://tools.ietf.org/html/rfc4301, 2005
12. S. Kumar, C.J. Hughes, and A. Nguyen. Carbon: Architectural Support For Fine-Grained Parallelism On Chip Multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, NY, USA, 2007. ACM, NY
13. A. Lankes, A. Herkersdorf, S. Sonntag, and H. Reinig. NoC Topology Exploration for Mobile Multimedia Applications. In *16th IEEE International Conference on Electronics, Circuits and Systems*, Dec 2009
14. A. Lankes, T. Wild, and A. Herkersdorf. Hierarchical NoCs for Optimized Access to Shared Memory and IO Resources. *Euromicro Symposium on Digital Systems Design*, pages 255–262, 2009
15. M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf. Application Scenarios for FlexPath NP. Technical Report TUM-LIS-TR-0501. Technische Universität München. Lehrstuhl für Integrierte Systeme, 2005

16. M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf. FlexPath NP – A Network Processor Architecture with Flexible Processing Paths. SoC 2008, Tampere, Finland, Nov 2008
17. G. De Micheli. Robust System Design With Uncertain Information. In *The Asia and South Pacific Design Automation Conference (ASP-DAC '03) Keynote Speech, Kitakyushu*, page 12, 2003
18. C. Müller-Schloer. Organic Computing: On The Feasibility Of Controlled Emergence. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 2–5, NY, USA, 2004. ACM, NY
19. R. Ohlendorf, A. Herkersdorf, and T. Wild. FlexPath NP – A Network Processor Concept with Application-Driven Flexible Processing Paths. CODES+ISSS 2005, Jersey City, NJ, USA, Sept 2005
20. R. Ohlendorf, M. Meitinger, T. Wild, and A. Herkersdorf. An Application-aware Load Balancing Strategy for Network Processors. HiPEAC 2010, Pisa, Italy, Jan 2010
21. P. Palatin, Y. Lhuillier, and O. Temam. CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs. In *Proc. ACM International Symposium on MICRO-39 Microarchitecture 39th Annual IEEE*, pages 247–258, 2006
22. W. Shi and L. Kencl. Sequence-Preserving Adaptive Load Balancers. ANCS 2006, San Jose, CA, USA, Dec 2006
23. J. Teich. Invasive Algorithms and Architectures. *it – Information Technology*, pages 300–310, 2008
24. D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J.F. Brown III, and A. Agarwal. On-Chip Interconnection Architecture Of The Tile Processor. *IEEE Micro*, pages 15–31, 2007
25. J. Zeppenfeld and A. Herkersdorf. Autonomic Workload Management for Multi-Core Processor Systems. In *International Conference on Architecture of Computing Systems*, 2010
26. J. Zeppenfeld, A. Bouajila, W. Stechele, and A. Herkersdorf. Learning Classifier Tables for Autonomic Systems on Chip. In *GI Jahrestagung*, pages 771–778, 2008

Chapter 4
PALLAS: Mapping Applications onto Manycore

Michael Anderson, Bryan Catanzaro, Jike Chong, Ekaterina Gonina, Kurt Keutzer, Chao-Yue Lai, Mark Murphy, Bor-Yiing Su, and Narayanan Sundaram

Abstract Parallel programming using the current state-of-the-art in software engineering techniques is hard. Expertise in parallel programming is necessary to deliver good performance in applications; however, it is very common that domain experts lack the requisite expertise in parallel programming. In order to drive the computer science research toward effectively using the available parallel hardware platforms, it is very important to make parallel programming systematical and productive. We believe that the key to designing parallel programs in a systematical way is software architecture, and the key to improve the productivity of developing parallel programs is software frameworks. The basis of both is design patterns and a pattern language.

We illustrate how we can use design patterns to architect a wide variety of real applications, including image recognition, speech recognition, optical flow computation, video background subtraction, compressed sensing MRI, computational finance, video games, and machine translation. By exploring software architectures of our applications, we achieved 10x-140x speedups in each of the applications. We illustrate how we can develop parallel programs productively using application frameworks and programming frameworks. We achieve 50%-100% of the performance while using four times fewer lines of code compared to hand-optimized code.

Keywords PALLAS · Software Architecture · Application Framework · Programming Framework · Design Pattern · Pattern Language

K. Keutzer (✉)
University of California, Berkeley, CA, USA
e-mail: keutzer@eecs.berkeley.edu

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_4,

4.1 PALLAS

PALLAS stands for Parallel Applications, Libraries, Languages, Algorithms, and Systems. We believe that productive development of applications for an emerging generation of highly parallel micro processors is the preeminent programming challenge of our time. Consequently, our goal is to enable the productive development of efficient parallel applications by domain experts, not just parallel programming experts. We believe that the key to the design of parallel programs is *software architecture*, and *software frameworks* [1] are the key to their efficient implementation. In our approach, the basis of both is *design patterns* and a *pattern language*. Borrowed from civil architecture, a *design pattern* refers to a generalizable solution to a recurring design problem. A *pattern language* is simply an organized way of navigating through a collection of *design patterns* to produce a design (Fig. 4.1). The computational elements of Our Pattern Language [2, 3] are built up from a series of computational patterns drawn largely from thirteen motifs [4] (Fig. 4.1(b)). We see these as the fundamental software building blocks that are then composed using the structural patterns of Our Pattern Language drawn from common software architectural styles [5], such as pipe-and-filter (Fig. 4.1(a)). A software architecture is then the hierarchical composition of computational and structural patterns, which we subsequently refine using lower-level design patterns.

This software architecture and its refinement, although useful, are entirely conceptual. To implement the software, we rely on frameworks. We define a pattern-oriented software framework as an environment built on top of a software architecture in which customization is only allowed in harmony with the framework's

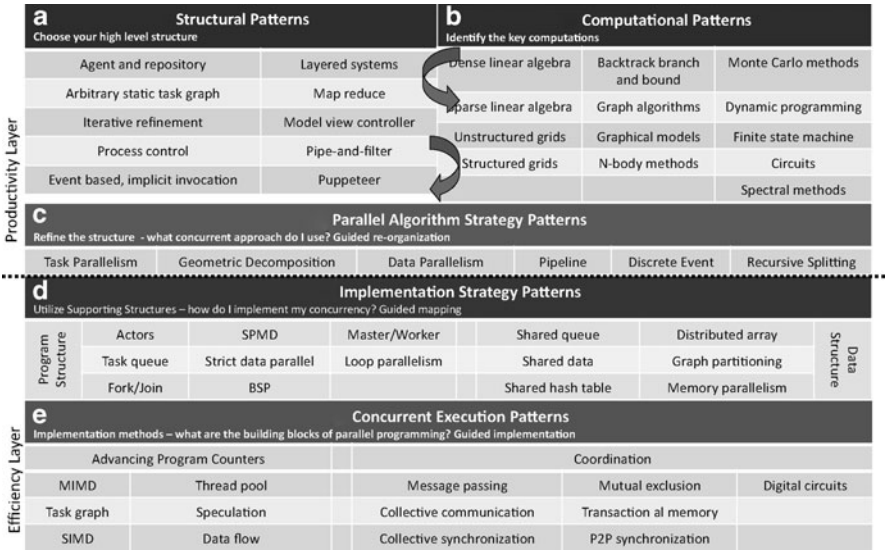


Fig. 4.1 Our Pattern Language

architecture. For example, if based on pipe-and-filter, then customization involves only modifying pipes or filters. We see application developers being serviced by application frameworks. These application frameworks have two advantages: First, the application programmer works within a familiar environment using concepts drawn from the application domain. Second, we prevent expression of many annoying problems of parallel programming such as non-determinism, races, deadlock, and starvation.

To test and demonstrate our approach to parallel software development we have applied a pattern-oriented approach to parallel software development to a broad range of applications in computer vision, speech recognition, quantitative finance, games, and natural language translation. We have first used patterns and Our Pattern Language as conceptual tools to aid in the design and implementation of the applications. This work is described in Section 4.2. As our understanding of the use of patterns matured we have used patterns to define pattern-oriented frameworks for a speech recognition application and a programming framework for data parallelism. This is defined in Section 4.4.

4.2 Driving Applications

We describe eight applications from a broad set of domains ranging including image recognition, speech recognition, optical flow computation, video background subtraction, compressed sensing Magnetic Resonance Imaging (MRI), computational finance, video games and machine translation. In each of these applications we demonstrate how our pattern-based approach establishes a common set of vocabulary, aids in understanding parallelism opportunities and bottlenecks, and leads to the development of efficient parallel implementations of the underlying algorithms. We first describe the overall software architecture of an application, then illustrate how pattern decomposition helps highlight parallelism opportunities and bottlenecks, and discuss execution speedups achieved.

4.2.1 Content-Based Image Retrieval

The Content-Based Image Retrieval (CBIR) application is used to select images that match a set of training samples from a huge image database. As shown in Fig. 4.2, the user will select some exemplar images as input, and then the CBIR application will collect features from images in the image database, train the classifier based on the chosen exemplar images, and exercise the classifier to find some of the images that match the characteristics of the exemplar. For example a user may want to find all the photos of roses in a large database of pictures of flowers. If there are incorrect classification results, the user can provide feedback

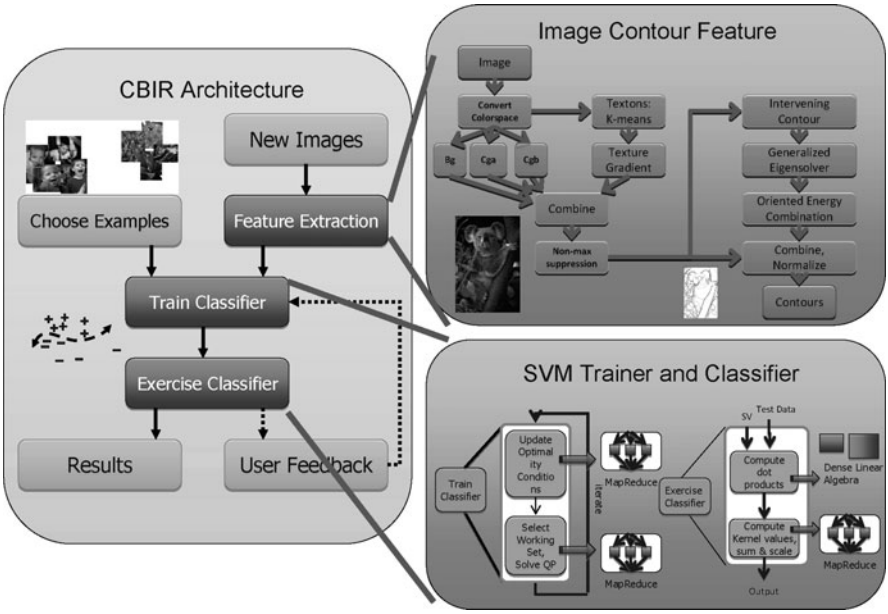


Fig. 4.2 Architecture of the CBIR application

to the system, and the system will retrain and reexamine the image database to generate more accurate results.

A wide variety of features can be used to describe an image, such as SIFT, SURF, HOG, MSER, color, texture, edges, contours, etc. We studied the state-of-the-art contour detection algorithm, the gPb algorithm [6]. This algorithm finds the boundaries between semantically meaningful objects in images without a priori knowing the content of the images. The computations of the gPb algorithm can be architected using the pipe-and-filter pattern as shown in Fig. 4.2. Gradients on color, brightness, and texture represent local cues of the image contours. The eigenvectors of an affinity matrix on pair-wise pixel similarity represent the global cues of the image contours. The gPb algorithm combines the local cues and the global cues to find image contours. Each computation in the gPb algorithm can be further architected by structural and computational patterns. For example, the k-means algorithm can be described by an iterative refinement pattern, iteratively compute sample means using the dense linear algebra pattern, and compute sample labels using the map-reduce pattern.

Given the feature vectors of images, we need to distinguish images that the end user recognizes as matches from the images the end user does not recognize as matches. Many machine learning algorithms can achieve this goal, such as k-nearest neighbor, naïve Bayes, logistic regression, support vector machine, decision tree, adaBoost, etc. We studied a particular state-of-the-art classifier approach known as support vector machines (SVM) [7]. The training phase and the classification phase of the SVM algorithm are architected in Fig. 4.2. The iterative

refinement pattern is used to describe the computation of the training phase. Within the iterator, the map-reduce pattern is used to describe the computation of updating optimal conditions, selecting working set, and solving the quadratic programming problem. For the classification phase, the dense linear algebra pattern is used to represent the computation of dot products on vectors, and the map-reduce pattern is used to represent the computation of kernel function, summation, and scaling.

By examining efficient parallel algorithms for performing image contour detection, along with careful implementation on highly parallel, commodity processors from Nvidia, we reduced the runtime of the gPb algorithm from 237 seconds on the Intel Core i7 920 (2.66GHz) platform to 1.8 seconds on the Nvidia GTX 280 GPGPU, about 130x speedup, with uncompromised results [8]. We implemented the SVM by the Platt's Sequential Minimal Optimization algorithm and an adaptive first and second order working set selection heuristic in parallel on the Nvidia GeForce 8800 GTX GPGPU, and achieved 9-35x speedup on the training phase, 81-138x speedup on the classification phase against LIBSVM [9] on the Intel Core 2 Duo (2.66 GHz) platform [10].

4.2.2 Optical Flow and Tracking

Optical flow computation is a crucial first step for almost all dense motion feature extraction in video. Optical flow models have become far more reliable over the years, and recent parallel hardware, especially GPUs, also offers the potential to meet the speed requirements of large throughput video analysis. Currently the dominant class of optical flow techniques is based on extensions of the variational model of Horn and Schunck. In particular, we use the Large Displacement Optical flow (LDOF) algorithm [11] which integrates discrete point matches with a continuous energy formulation in order to obtain accurate flow for large displacements of small structures. This helps us track objects like limbs in human motion, balls in sports videos etc. much more accurately than other techniques.

A quite general numerical scheme that can efficiently compute solutions of basically all variational models is based on a coarse-to-fine warping scheme, where each level provides an update by solving a nonlinear system given by the Euler-Lagrange equations followed by fixed point iterations and a linear solver [12]. This strategy is very general and accommodates even non-convex regularizers (We use a convex regularizer that approximates the L_1 norm). On serial hardware, a very efficient and quite straightforward linear solver is given by Gauss-Seidel with successive overrelaxation. But on parallel hardware, there is a need to investigate which algorithms perform better for optical flow problems. It is necessary to fully characterize the properties of the matrices involved - in this case, they are positive definite, enabling the use of the Conjugate gradient algorithm.

Figure 4.3(a) shows the overall architecture of the application. There are 3 main iterative refinement blocks – One for the coarse to fine refinement, one for doing the fixed-point iterations (linearizing the non-linear regularizer), and the third

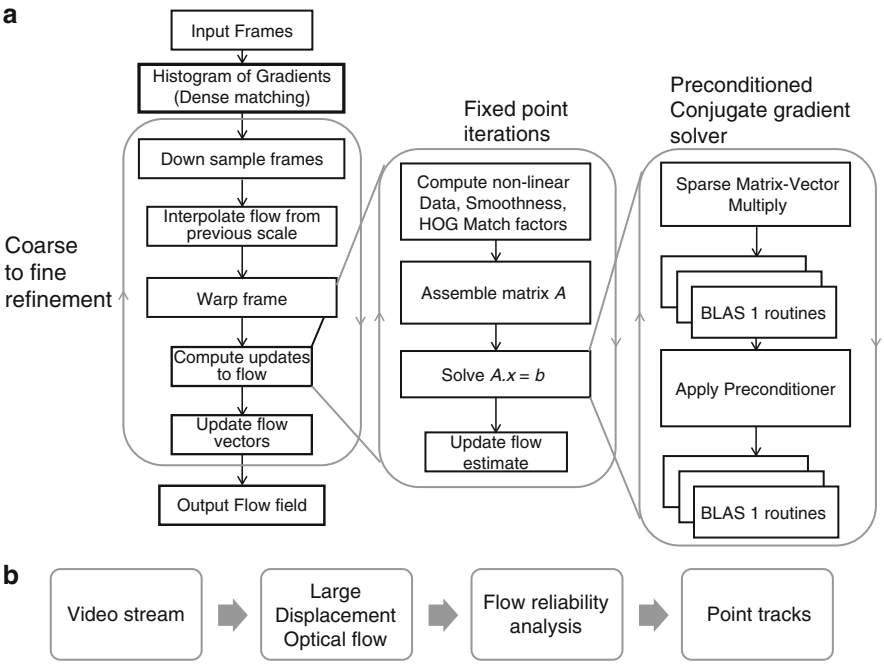


Fig. 4.3 **(a)** Architecture of the large displacement optical flow application. **(b)** High-level description of the point tracker based on large displacement optical flow

for the iterative sparse linear solver. Our implementation uses Preconditioned Conjugate gradient for solving the linear system of equations (third loop) for all the fixed points at all scales. Compared to other parallel solvers like Red-black relaxations, the preconditioned conjugate gradient algorithm performs more work per iteration (2.1x more) but requires fewer iterations (about 3x less), thus ensuring 40% better performance. For all solvers, it was necessary to take advantage of the sparse matrix structure (block penta-diagonal) to achieve high memory throughput. Compared to a serial Gauss-Seidel solver running on Intel Core2 Quad Q9550, our conjugate gradient solver on Nvidia GTX480 achieves a 47x speedup. We achieve a 78x speedup on the full application for achieving an equivalent error rate on the Middlebury optical flow dataset [13]. The runtime for running LDOF on a pair of 640x480 sized frames has been brought down from over 2 minutes to 1.8 seconds, making large displacement optical flow practical to use in a wide variety of motion estimation tasks.

By using the efficient optical flow solver, we developed a point tracking system [14]. Figure 4.3(b) shows the high level description of the tracker. The optical flow computation dominates the runtime of the tracker, taking up 93% of the total runtime even after parallelization. Compared to the most commonly used KLT tracker [15], we can track three orders of magnitude more points while achieving

46% better accuracy. Compared to the Particle Video tracker [16], we achieved 66% better accuracy while running an order of magnitude faster. In addition to the improved accuracy and speed, the tracker based on LDOF also provides improved tracking density and the ability to track large displacements. This has been possible through algorithmic exploration and an efficient parallel implementation of the large displacement optical flow algorithm on highly parallel processors (GPUs).

4.2.3 Stationary Video Background Subtraction

Stationary-video background subtraction is the problem of extracting the moving parts from a video where the camera does not move during the duration of the video, as is the case in surveillance videos. One tool used in solving this problem is a singular value decomposition (SVD) of a matrix with one column for each frame and a row for each pixel of the video [17]. The SVD operation makes it possible to extract the parts of the video that are common to every frame (i.e. the background).

The shape of this video matrix is extremely tall and skinny, because the number of pixels in a frame is typically far greater than the number of frames. For matrices of this shape, the SVD can very efficiently be found by first solving the QR decomposition of the matrix. The QR decomposition is an operation in which a matrix is factored into a product of two matrices, Q and R, where Q is orthogonal and R is upper triangular. So the main computation being done in this approach to stationary-video background subtraction is the QR decomposition of a tall-skinny matrix.

Figure 4.4 is the architecture used for a stationary-video background subtraction algorithm. The video matrix is the main data structure. We can apply geometric decomposition to break down the data structure into many small blocks that fit in cache, shown in Fig. 4.4. A recent algorithm for the QR decomposition, Communication-Avoiding QR [18], allows us to factor the entire matrix using only a sequence of small operations on these blocks. We get good performance because we are able to operate on each block in parallel in each processor's cache. Using

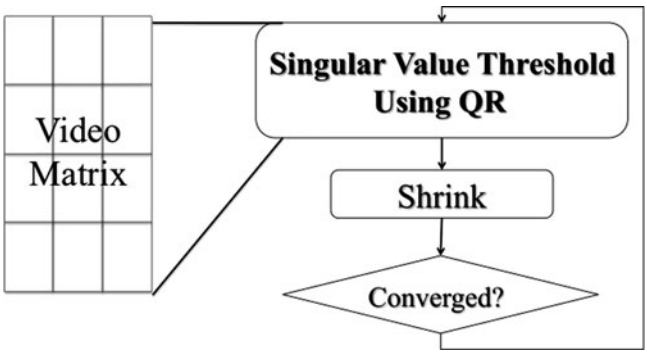


Fig. 4.4 Architecture of stationary-video background subtraction application

this approach on a Nvidia GTX480 card, we achieve a 27x speedup for the entire application compared to using Intel's Math Kernel Library.

4.2.4 Automatic Speech Recognition

The Automatic Speech Recognition (ASR) application takes a speech audio waveform as input and produces a sequence of words representing the most-likely utterance the speaker intended to communicate. As shown in Fig. 4.5, ASR does this by first extracting acoustic features from the waveform and then decodes the feature sequence to produce a word sequence.

The feature extraction process involves a sequence of signal processing steps in the form of the pipe-and-filter pattern. The filters are aimed to remove variations among speakers and room acoustics and preserve features most useful to distinguishing word sequences. The decoding process performs statistical inference on a hidden Markov model using the Viterbi algorithm. The inference is performed by comparing each extracted feature to a speech model, which is trained off-line using a set of powerful statistical learning techniques. The module that performs inference, shown in Fig. 4.5 as the Inference Engine, has an iterative outer loop (iterative refinement pattern), that handles one input speech feature vector at a time. In each of the loop iterations, the algorithm performs a sequence of data-parallel steps (pipe-and-filter

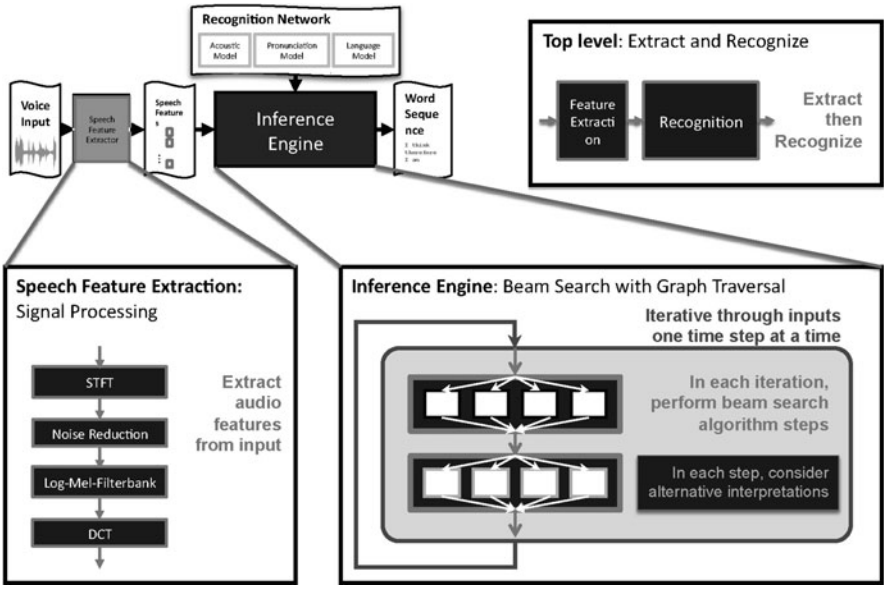


Fig. 4.5 Architecture of the ASR application

pattern). Modern manycore processors take advantage of the parallelism within each algorithmic step to accelerate the inference process (map-reduce pattern).

An implementation of such an inference engine involves a parallel graph traversal through an irregular graph-based knowledge network with millions of states and arcs (graph algorithm pattern). The challenge is not only to define a software architecture that exposes sufficient fine-grained application concurrency but also to efficiently synchronize between an increasing number of concurrent tasks and to effectively utilize parallelism opportunities in today's highly parallel processors.

Chong, You et al. [19, 20] demonstrated substantial speedups of 3.4x on Intel Core i7 and 10.5x on NVIDIA GTX280 compared to a highly optimized sequential implementation on Core i7 without sacrificing accuracy. The parallel implementations contain less than 2.5% sequential overhead, promising scalability and significant potential for further speedup on future platforms. Further parallel optimizations were demonstrated in Chong et al [21] through speech model transformations using domain knowledge and exploring other speech models on the latest manycore platforms [22].

Additional opportunities on the basis of such parallel implementations include accelerating the Hidden Markov Model (HMM) training algorithm, for example the Baum-Welch algorithm, and performing realtime multistream decoding, e.g. for audiovisual speech recognition. Both of these applications can make use of the highly parallelized likelihood computations that have already been optimized for the purpose of ASR, and can be expected to obtain similar gains in performance due to their highly regular and parallel structure.

4.2.5 Compressed Sensing MRI

Compressed Sensing is an approach to signal acquisition that enables high-fidelity reconstruction of certain signals sampled significantly below the Nyquist rate. The signal to be reconstructed must satisfy certain "Sparsity" conditions [23], but these conditions are satisfied at least approximately by signals in many aplications. Compressed Sensing has been applied to speed up the acquisition of MRI data [24], increasing the applicability of MRI to Pediatric medicine. Compressed Sensing reconstruction is computationally difficult, requiring solution of a non-linear L_1 minimization problem [23]. L_1 minimization problems are more difficult than, for example, least squares problems due to the non-differentiability of the L_1 objective function. The difficulty is compounded by the size of the problems to be solved: we must determine the value of each voxel in a 3D MRI scan, so our L_1 minimization typically involves billions of variables. This computational difficulty leads to long runtimes, limiting the clinical applicability of the technique. MRI images must be available interactively (i.e. in a few minutes) to the radiologist performing the examination, so that time-critical decisions can be made about further images to be taken.

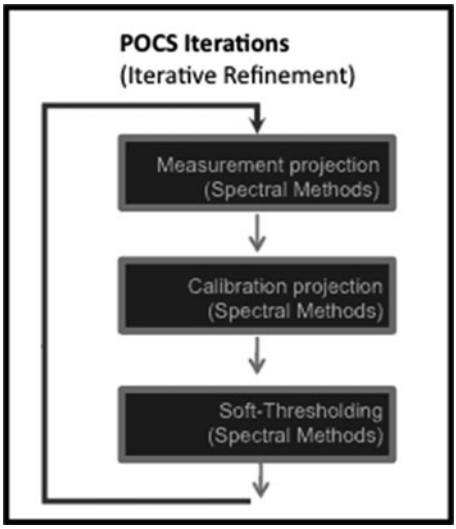


Fig. 4.6 Architecture of the compressed sensing MRI application

Our solver for this problem implements a Projections Onto Convex Sets (POCS) method, which is shown in Fig. 4.6: we iteratively project the solution onto convex sets representing sparse signals and the feasible region of the minimization problem. Since these sets are convex and their intersection is nonempty, the procedure is guaranteed to converge. While L_1 minimization problems can be cast as linear programs (LP) and solved by e.g. the Simplex method or Interior Point methods, the high numerical accuracy of LP solvers is unnecessary in our case. The POCS algorithm is much faster, and produces sufficiently high-quality images. Even still, the original Matlab implementation required approximately 30 seconds per 2D slice; a full 3D scan typically has several hundred slices, and an entire scan required hours to reconstruct. The L_1 minimization is necessarily proceeded by a "Calibration" phase, which requires the solution of a number of least-squares problems. We solve these systems directly, using standard linear algebra libraries. The solutions of these systems provide a "Self-Consistency" model that incorporates information from up to 32 redundant acquisitions (channels) of the MR image.

We have produced two highly efficient parallel implementations of the POCS algorithm. Our evaluation platform is a 12-core 2.67 GHz Intel Xeon E5650 machine with four 30-core, 8-wide-SIMD 1.3 GHz Nvidia Tesla C1060 GPGPUs. For typically sized datasets with 8 channels, our OpenMP parallelized calibration runs in 20 seconds (140 ms per slice), on average. 40 iterations of our OpenMP POCS solver, sufficient for most datasets to converge, run in 334 seconds (2.1 seconds per slice) using all 12 CPU cores. On a single GPGPU, our Cuda POCS solver runs in 75 seconds (480 ms per slice) - 4.5x faster than the version on 12 CPU cores. Using multiple GPUs we get nearly linear speedup: the POCS solver runs in 20 seconds. Our GPU wavelet implementation is bandwidth-inefficient: a more highly optimized implementation will be up to 50% faster. Also, multi-GPU parallelization will

provide additional 3-4X speedup. Using our OpenMP calibration and our Cuda POCS solver results in 40-second reconstruction times: this is the first clinically feasible compressed sensing MRI reconstruction implementation [25].

4.2.6 *Market Value-at-Risk Estimation in Computational Finance*

The proliferation of algorithmic trading, derivative usage and highly leveraged hedge funds has necessitated the estimation of market Value-at-Risk (VaR) in future market scenarios to measure the severity of potential financial losses. VaR reports are typically generated daily to summarize the vulnerabilities to market movements in the positions financial business units take. They are the central tenet of financial institutions' market risk management operations.

VaR estimation is a direct application of the Monte Carlo computation pattern. It broadly entails simulating the effects of thousands to millions of potential market scenarios to collect statistics about the portfolio loss distribution going into the future. Each VaR simulation involves four steps as illustrated in Fig. 4.7(a). There exist significant parallelism opportunities for executing each of the steps over all the scenarios (Fig. 4.7(b)). We use the loss distribution of the resulting portfolio valuation, to estimate the exposure of a portfolio to a severe loss. The VaR is

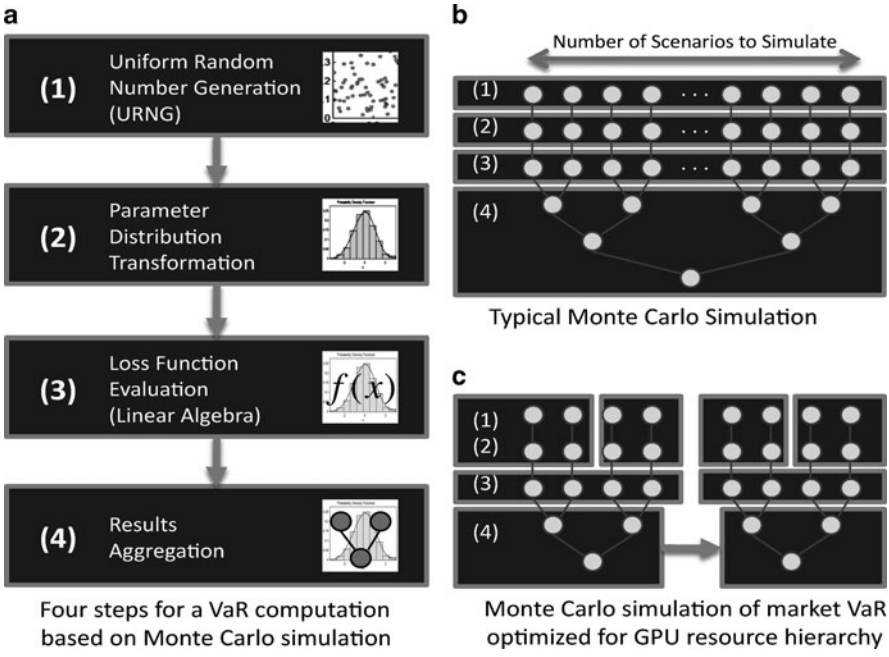


Fig. 4.7 Architecture of the market value-at-risk estimation in computational finance application

typically taken to be the value associated with a specific frequency in the range of the 1-in-100 up to the 1-in-20 loss event.

For an implementation optimized on a highly parallel platform such as today's GPUs [26], we use the geometric decomposition pattern to partition the workload into blocks such that a block of scenarios can fit into a given size of fast memory in an implementation platform. Specifically, a small block of steps (1) and (2) can be merged and made to fit in lowest level cache, and a block of all four steps should fit into the memory on the device in a GPU-based platform (Fig. 4.7(c)).

We evaluated a standard implementation of quadratic VaR estimation using a portfolio-based approach, where financial outcome for all instruments in a portfolio are aggregated from quadratic approximations of risk factor losses. For step (1) we use a Sobol quasi-random number sequence; for step (2), we used the Box-Mueller transformation; for step (3), we use the quadratic estimation for the loss estimation; and for step (4), we use parallel reduction within a block and sequential reduction across blocks.

For a portfolio with up to 4096 risk factors, we achieved a 8.21x speedup on the GPU compared to an algorithmically equivalent multicore CPU implementation. Step (1) and (2) attained a speedup of 500x by more effectively utilizing computation and memory locality from applying the geometric decomposition pattern. Step (3) attained a speedup of 5x, and is limited by capabilities of Basic Linear Algorithm Subroutine (BLAS) implementations. Step (4) takes proportionally negligible runtime. Noting the key computation bottleneck in the loss estimation, we reformulated step (3) algorithmically and gained a further 60x speedup in loss estimation.

4.2.7 Games

A typical video game is a composition of several large subsystems such as physics, artificial intelligence (AI), and graphics. Subsystems can be large reusable libraries or "engines", or functions created for a specific game. A primary concern of the game designer is how to efficiently manage communication between subsystems. The communication becomes more complex if the subsystems are to be run in parallel on a multicore device, requiring special coordination or locking for shared data. Also, each subsystem should have a well defined interface so it can easily be swapped with another similar library if necessary [27].

A solution to this problem is the application of the puppeteer pattern (Fig. 4.8). A puppeteer sits above the subsystems and acts as an intermediary for communication between subsystems. Suppose the AI subsystem changes a character's direction and needs to inform the Physics subsystem, which will in turn update the character's position and velocity. Instead of interfacing directly with the Physics subsystem, the AI subsystem informs the puppeteer of the change. The puppeteer passes on the information to any interested subsystems. The main benefit of the

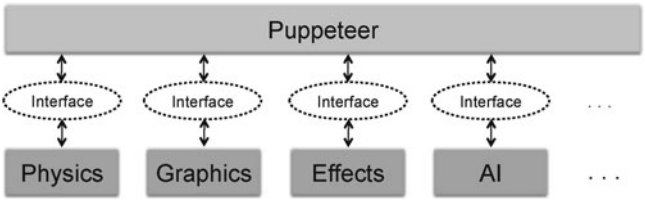


Fig. 4.8 Architecture of the game application

puppeteer pattern is that it reduces the total number of subsystem interfaces, which allows for greater flexibility and scalability.

Graphics Processing Units were developed specifically to enable more computation in the graphics subsystem. For other subsystems to take advantage of these parallel devices, they must be decomposed into their patterns and sped up individually. A simple AI subsystem, for example, is a collection of character state machines that read and write from a set of shared data. The shared data could contain the locations and orientations of the characters. This system can be architected structurally using the Agent & Repository pattern. Since the AI state machines operate independently from one another within a single frame, the task parallelism implementation pattern can be applied to speed up computation on parallel hardware.

Video games have a real-time constraint, the frame rate. The worst-case amount of computation must meet this constraint for a user with the minimum required hardware, unless the computation does not affect game play and can be optionally skipped. Another challenge is effectively managing access to the scene graph, the main data structure containing the game's state. Data transfer can also be prohibitively expensive, especially when moving between devices.

4.2.8 Machine Translation

Machine translation (MT) is one of the classic problems in computer science and a vast area of research in the field of natural language processing (NLP). High-quality and fast MT enables a variety of exciting applications, such as real-time translation in foreign environments on handheld devices as well as defense and surveillance applications. A fast machine translator will also enable people speaking different languages communicate and share resources altogether on the Internet.

The most prevalent way of machine translation is the CKY algorithm [28, 29], which is composed of three phases: To use a translation model to translate phrases, to combine the translated phrases in a bottom-up fashion, and to extract the most likely translation with a top-down traversal. The architecture of the ML application is summarized in Fig. 4.9, where the three phases are represented by the pipe-and-filter pattern. The bottleneck of the CKY algorithm is in the second phase, in which we examine the probabilities of all possible combinations over the translated phrases using an N-gram language model, and this computation can be represented

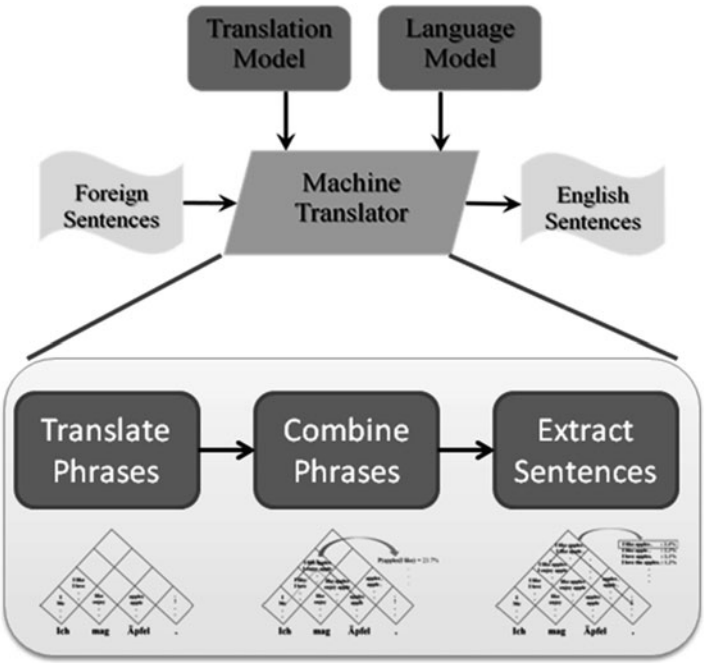


Fig. 4.9 Architecture of the ML application

by the dynamic programming pattern. We parallelized the second step of the CKY algorithm on both GPU and CPU. When translating 1000 sentences with an average length of 28 words from Spanish to English, we achieved 1.8x speedup on GTX 480 and 2.3x speedup on Core i7 using 4 threads. When translating 350 sentences with length of more than 40 words, we achieved 2.3x speedup on GTX 480 and 2.6x speedup on Core i7 using 4 threads. This shows our parallelization works better with longer sentences because more concurrency is available.

4.2.9 Summary

In this section, we have explored eight applications from a variety of domains, and demonstrated how patterns can serve as a set of vocabulary to allow software developers to quickly articulate and communicate the architecture of a piece of software. We have also touched on how patterns provide a set of known tradeoffs to inform software developers of potential bottlenecks in a design. These known tradeoffs help software developers identify key design decisions impacting the performance of an application. In the next section, we provide some perspective on the parallel speedups achieved in these applications.

4.3 Perspectives on Parallel Performance

When one writes parallel software, performance considerations are always close at hand. This is natural, since one could always forgo parallelization and use a sequential implementation, were it not for performance requirements. These performance considerations raise many questions: How can we tell if a program has been successfully parallelized? How can we compare performance between parallel platforms? How generally can we extrapolate from one performance claim to performance projections for another algorithm or architecture? Varying assumptions and perspectives lead to a surprising diversity of opinions on these questions, which is why it is important to be explicit about the assumptions one makes when making and evaluating performance claims.

We consider performance results under the following three guidelines, which we will explain, along with their justifications and implications.

1. Perfect linear speedup, under strong scaling, is not a necessary condition for successful parallelization.
2. The most useful kind of performance information comes from measured performance of a real application, running on real hardware.
3. Some algorithms are inherently more difficult to parallelize than others.

4.3.1 Linear Scaling Not Required

In the past, when one evaluated parallel software, it was important to achieve linear speedup under strong scaling, meaning that if one doubled the number of processors, keeping the problem size the same, the computation should take half the runtime. This was primarily due to economic reasons. Since a computer with twice as many cores cost at least twice as much as a smaller computer, in order to recoup one's investment, linear scaling was required.

The situation has now changed, since we integrate large numbers of cores on a single die. Consider the status quo before the advent of on-die parallelism. Processor vendors created new microarchitectures, spending ever larger amounts of transistors on increasingly sophisticated single-thread processors. However, it was widely known that new microarchitectures did not provide performance gains in proportion to their increased complexity. Pat Gelsinger, of Intel, famously stated that processor performance increases only with the square root of transistor count [30]. Although the industry did not see linear increases in performance with respect to transistor count, the resulting performance gains realized through the uniprocessor era were still sufficient to propel the industry forward, providing end users new capabilities through increased performance.

On-die parallelism has exposed architectural complexity to the programmer as increased core counts. Today, increases in transistor count, to a first order approximation, are accompanied with linear increases in exposed parallelism, although not

to linear increase in cost, due to Moore's law. Accordingly, sublinear performance scaling as we increase the number of transistors, and hence the number of cores, should still provide end-users with the increased capabilities they have come to expect from the computer industry. In addition, workload sizes tend to scale as problems get harder, making parallelization easier to use in practice than Amdahl's law and strong scaling assumptions would suggest [31]. We do not need to apply parallelism to every computation, only to those which are computationally intensive, which tend to have better parallelization characteristics because they are larger problems. Summarizing, when evaluating the success of the parallelization of a particular piece of software, we believe it is important to remember that the economics of parallelism today have made it possible for even modestly parallelized software to be successful.

4.3.2 Measure Real Problems on Real Hardware

It is often tempting to examine the parallel performance of the kernels of an application. They capture the heavy computational load of the application, and so their performance is critical. However, excessive focus on the kernels can be a mistake, since the glue that holds an application together can quickly become a bottleneck when the kernels are composed to form an application. Data structures often have to be transformed between kernels, serial work must be done to decide how the application should proceed, kernels must coordinate to ensure correct results. Accordingly, the most important performance data is achieved on complete applications, taking into account the composition of the entire application.

It is also important to examine realized, delivered application performance on concrete hardware, rather than comparing peak kernel-performance claims across various hardware platforms and trying to generalize and extrapolate expected performance. Peak, theoretical numbers are useful bounds, but they can be distracting. Most computations are not as easy to parallelize as kernels like Linpack [32], even though the kernels provide bounds on application performance. Some parallel platforms are significantly more brittle than others, in the sense that they may do very well on isolated kernels, but their general performance is fairly poor. In the end, the most important performance results concern complete applications on concrete hardware, all other performance results are useful primarily as bounds.

4.3.3 Consider the Algorithms

Successful parallelization requires consideration of the algorithms being parallelized. This is important in two senses. Firstly, we must realize that certain algorithms are harder to parallelize than others. Algorithms which require a lot of data sharing between threads, have unpredictable memory access patterns, or are

characterized by very branchy control flow, are often inherently more difficult to parallelize than others. Some algorithms are embarrassingly sequential. Some are mostly sequential, and can be parallelized only through heroics that often result in modest performance gains despite significant software complexity. For this reason, it's important not to compare speedup results for one algorithm versus another, if the algorithms accomplish different tasks. One should not expect all algorithms to parallelize with the same efficiency.

Secondly, when parallelizing an application, it is often useful to rethink the algorithms involved. Sometimes it is better to use algorithms which do more work, but are more parallelizable. And of course, if rethinking the algorithm leads to algorithmic variations which improve parallel as well as sequential efficiency, those improvements should be capitalized on.

4.3.4 Summing Up

At the end of the day, we parallelize applications because the increased performance leads to increased capabilities for end-users. Ultimately, we parallelize in order to solve bigger and harder problems, continuing to realize the full performance provided by Moore's law in real applications.

4.4 Patterns to Frameworks

We define a *software architecture* as a hierarchical composition of structural and computational patterns. A *pattern-oriented framework is a software environment* (e.g. Ruby on Rails) that is based on particular software architecture (e.g. Model View Controller) and in which all user customization must be in harmony with that software architecture. In other words only particular customization points within the software architecture (e.g. elements of the Controller) are available for end-users to customize. Patterns and pattern-oriented frameworks assist application developers in quick prototyping of parallel software and enable fast exploration of software architectural design space. There are two types of frameworks being developed to target different developer usage models. The application frameworks provide an efficient reference implementation in an application domain along with a set of extension points to allow customization of functions in selected modules without jeopardizing the efficiency of the underlying efficient infrastructures. The programming frameworks provide a set of flexible tools to take advantage of parallel scalability in hardware without the burden of particular platform details. We motivate the need for these two types of frameworks and illustrate how they can be used.

4.4.1 Application Frameworks

Developing an efficient parallel application is often a significant undertaking. It requires not only a deep understanding of an application domain, but also advanced programming techniques for a parallel implementation platform. A deep understanding of an application domain enables domain experts to discover parallelization opportunities and make application-level design trade-offs to meet the requirements of the end user. Advanced programming techniques allow parallel programming experts to exploit the parallelization opportunities to utilize available parallel resources and navigate various levels of synchronization scope of an implementation platform.

In Automatic Speech Recognition (ASR) inference engine development, application domain knowledge includes topics such as: pruning heuristics to reduce required computation while maintaining recognition accuracy, and recognition-network construction techniques to handle periods of silence between word utterances. Advanced programming techniques include designing data structures for efficient vector processing, constructing program flows to minimize expensive synchronizations, and efficiently utilizing the atomic-operations supported on the implementation platform.

With the increasing complexity of parallel systems, domain experts often must make application level design trade-offs without the full view of parallel performance implications. On the other hand, parallel programming expert may not be aware of application-level design alternatives to optimize computations and synchronizations away from the performance bottlenecks they discover.

With Our Pattern Language, an application domain expert can quickly gain insights into potential parallel performance implications of a design by architecting it using the structural and computational patterns and becoming aware of the trade-offs governing these patterns. For the most commonly reoccurring composition of patterns in a domain, we can construct application frameworks pre-optimized for various parallel platforms. An example of such an application framework is proposed for the ASR application domain in Fig. 4.10. The application framework is based on an efficient parallel implementation of large vocabulary continuous speech recognition that achieved over 11x speedup over an optimized sequential implementation on CPU [21].

The application framework for ASR is hierarchical, with the top-level containing Feature Extractor and Inference Engine as fixed components (Fig. 4.10(a)). User can customize input format, intermediate data format, recognition network format, and output format according to a specific end-user usage model. The Feature Extractor component is a pipe-and-filter pattern-based-framework where the filters can be customized according to the end application needs (Fig. 4.10(b)). The Inference Engine component contains an Inference Engine Framework where there is a fixed structure of sequential steps wrapped in an iterative loop implementing the Viterbi algorithm (Fig. 4.10(c)). The computation within each step can be customized to incorporate many variations of the application.

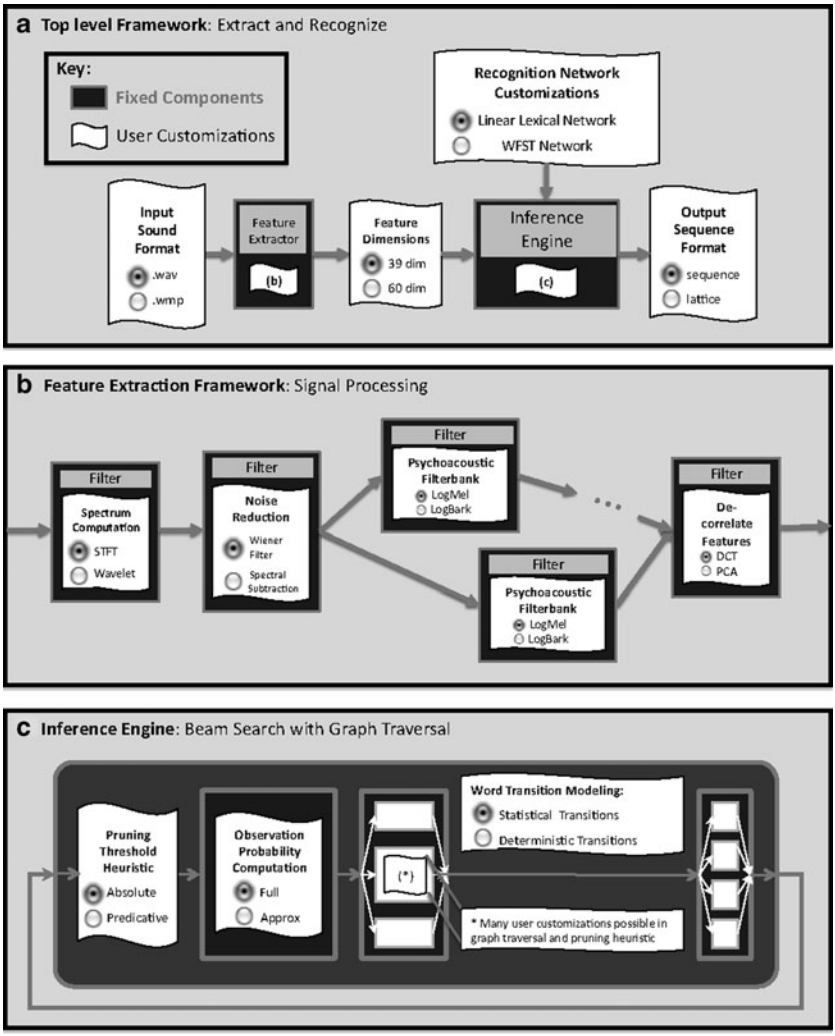


Fig. 4.10 Application framework for the ASR application

For application domain experts, an application framework serves to restrict implementations to a software architecture that is known to be efficient while providing a plethora of opportunities for user customizations. Different user customizations can result in a whole class of applications in an application domain. For parallel programming experts, the application framework serves to accentuate critical performance bottlenecks in a class of applications, where performance improvements in these bottlenecks can lead to performance improvement of the whole class of applications. We demonstrated the effectiveness of our ASR application framework by introducing it to a Matlab/Java programmer. She enabled

lip-reading in speech recognition by extending the audio-only speech recognition application framework to an audio-video speech recognition application. She was able to achieved 20x speedup in her application by implementing only plug-in modules for the Observation Probability Computation (Fig. 4.10(c)) and file input/output modules on the underlying manycore platform.

An application framework captures an efficient software architecture that implements a common reoccurring composition of patterns for an application domain. It creates a productive interface between application domain experts and parallel programming experts.

4.4.2 Programming Frameworks

4.4.2.1 Efficiency & Portability Through Programming Frameworks

While application frameworks help application developers to create new and interesting applications within a specific architecture, application domain researchers and application framework developers need more flexibility to create the tools they need. It is essential to provide frameworks that will help them take advantage of the hardware scalability from parallelism while still being shielded from the particular platform details. We believe programming frameworks provide this abstraction. It might be tempting to assume that programming frameworks should be completely agnostic about the application domain. In practice, programming frameworks need to support specific application domains so that they can take advantage of data structures and transformations that are specific to a particular domain. In other words, in order to ensure good performance it is necessary to tailor the optimizations performed to a particular domain.

Application domains like computer vision and machine learning heavily employ regular data structures (dense matrices, vectors, structured sparse matrices etc.). In these cases the important optimizations that need to be performed are figuring out how much concurrency in the application needs to be exposed, and how to map this efficiently onto the hardware. In particular, modern parallel processors have several levels of parallelism – at the SIMD level, at the thread level, at the core level etc. Hardware and programming model restrictions may or may not allow us to exploit all these levels efficiently.

In addition, programming frameworks can also handle optimizations that are specific to particular architectures. For instance, the limited physical memory of current many-core architectures like CUDA-capable GPUs and the high cost of data transfer between the CPU and GPU mean memory management through efficient scheduling is important. Programming frameworks can help the application framework developers by performing high quality task and data transfer scheduling to ensure low overheads and better efficiency [33].

4.4.2.2 Copperhead

During our work investigating application parallelization, we discovered that the Data Parallelism, Strict Data Parallelism, and SIMD patterns predominate in many important computations. The Data Parallel pattern involves finding parallelism in a computation by examining independent data elements in the computation. The Strict Data Parallelism pattern is an implementation pattern where the programmer exploits available data parallelism by mapping independent threads over independent data elements, and the SIMD pattern is an execution pattern where the programmer utilizes Single Instruction, Multiple Data hardware to efficiently execute operations over vectors.

In our opinion, Data Parallelism seems to be increasingly important, since it provides abundant, scalable parallelism for finely-grained parallel architectures, towards which the industry is headed. Accordingly, we decided to build a framework to enable more productive exploitation of Data Parallelism. This framework is called Copperhead.

Copperhead is a functional subset of the Python programming language, designed for expressing compositions of data-parallel operations, such as map, reduce, scan, sort, split, join, scatter, gather, and so forth. Parallelism in Copperhead arises entirely from mapping functions over independent data elements, and synchronization also arises entirely from joining independent arrays, or accessing non-local data.

The specifics of high-performance data-parallel programming often depend critically on the particular composition of data-parallel operations. For example, when parallelism is nested, the compiler can choose to turn parallel map invocations into sequential iterations, but the choice of whether a particular map is executed in parallel depends on its composition into the rest of the computation, as well as the particulars of the parallel platform being targeted. Consequently, Copperhead makes use of Selective, Embedded, Just-In-Time Specialization [34] to use information from the computation being performed in order to specialize the resulting code to the platform being targeted. When a data-parallel function call is invoked, the runtime examines the composition of data-parallel operations, and compiles it into parallel C, which is then dispatched on the parallel platform.

Copperhead is designed to support aggressive restructuring of data-parallel computations in order to map well to parallel hardware, with the goal of minimizing synchronization and data movement, which are the enemies of successful parallel computing. By specializing Copperhead programs to Nvidia Graphics Processors, we achieved 45-100% of the performance with about four times fewer lines of code when compared to hand-tuned CUDA C++ code on sparse matrix vector multiplication, preconditioned conjugate gradient linear solver, and support vector machine training routines. Our goal is to develop Copperhead to the point that it can provide full support of implementing the computations we have been investigating in Computer Vision and Machine Learning, providing useful performance as well as high productivity [35].

4.5 Conclusions

Our goal is to enable the productive development of efficient parallel applications by domain experts, not just parallel programming experts. As the world of computing becomes more specialized, we believe that understanding particular domains, such as computer vision, will be challenging enough, and domain experts will not have the time or inclination to become expert programmers of parallel processors as well. Thus if domain experts are to benefit from computing advances in parallel processors, new programming environments tailored for domain experts will need to be provided. We believe that the key to the design of parallel programs is software architecture, and the key to their efficient implementation is software frameworks. In our approach, the basis of both is design patterns and a pattern language. Further, we believe that patterns can empower software developers to effectively communicate, integrate, and explore software designs.

To test our beliefs we have explored eight applications from a wide variety of domains. In particular, we have successfully applied patterns to architect software systems for content-based image retrieval, optical flow, video background subtraction, compressed-sensing MRI, automatic speech recognition, and value-at-risk analysis in quantitative finance. We are in process of applying patterns to architect software systems for computer games and machine translation. Altogether these applications show very diverse computational characteristics and nearly cover the entire range of computational patterns in our pattern language. In our explorations we have demonstrated how patterns can serve as the basic vocabulary for the description of the architecture of these applications. We have also shown how the choice of patterns in which to describe an architecture naturally explores a set of known trade-offs that helps to inform software developers of potential bottlenecks in a design. These known trade-offs help software developers identify key design decisions impacting the performance of an application. In this process we have indeed convinced ourselves that patterns were not only useful in helping to conceptualize the architecture of a software system and communicate it to others, but patterns are also useful in achieving efficient software implementations. In the process of creating parallel implementations of this wide variety of applications we also gained some general insights about speeding up applications on parallel processors and we have reported those here as well.

We are also investigating how architectures based on patterns may be used to define application and programming frameworks. We define a (pattern-oriented) framework as a software environment in which all user customization must be in harmony with the underlying architecture. An application framework is a domain-specific framework that solves application-level problems like speech recognition and a programming framework is a framework that solves an programming implementation level problem like the implementation of data parallelism. The application frameworks provide an efficient reference implementation in an application domain along with a set of extension points to allow customization of functions in selected modules without jeopardizing the efficiency of the underlying efficient

infrastructures. The programming frameworks provide a set of flexible tools to take advantage of parallel scalability in hardware without the burden of particular platform details. We motivate the need for these two types of frameworks and illustrate how they can be used. There are many open questions about the relative merit of application and programming frameworks versus alternative approaches to software implementation such as domain-specific languages. We are in the process of clarifying the advantages and disadvantages of these two approaches.

4.6 Appendices

4.6.1 Structural Patterns

- Pipe-and-filter: A structure of a fixed sequence of filters that take input data from preceding filters, carry out computations on that data, and then pass the output to the next filter. The filters are side-effect free; i.e., the result of their action is only to transform input data into output data.
- Iterative refinement: A structure of an initialization followed by refinement through a collection of steps repeatedly until a termination condition is met.
- Map-reduce: A structure of two phases: (1) a map phase where items from an "input data set" are mapped onto a "generated data set", and (2) a reduction phase where the generated data set is reduced or otherwise summarized to generate the final result.
- Puppeteer: A structure of a puppeteer encapsulates and controls references of the puppets by delegating operations to the puppets and collecting return data from the puppets.

4.6.2 Computational Patterns

- Dense linear algebra: A computation is organized as a sequence of arithmetic expressions acting on dense arrays of data. The operations and data access patterns are well defined mathematically so data can be pre-fetched and CPUs can execute close to their theoretically allowed peak performance. Applications of this pattern typically use standard building blocks defined in terms of the dimensions of the dense arrays with vectors (BLAS level 1), matrix-vector (BLAS level 2), and matrix-matrix (BLAS level 3) operations.
- Graph algorithm: A computation which can be abstracted into operations on vertices and edges, with vertices represent objects, and edges represent relationship among objects.
- Monte Carlo: A computation that estimate a solution of a problem by statistical sampling its solution space with a set of experiments using different parameter settings.

- Dynamic Programming: A computation that exhibits the properties of overlapping subproblem and optimal substructure. Overlapping subproblem means a problem can be solved by smaller overlapping subproblems recursively. Optimal substructure means the optimal solution of a problem can be obtained by combining the optimal solutions of the subproblems properly.

4.6.3 Parallel Algorithm Strategy Patterns

- Data Parallelism: An algorithm is organized as operations applied concurrently to the elements of a set of data structures. The concurrency is in the data. This pattern can be generalized by defining an index space. The data structures within a problem are aligned to this index space and concurrency is introduced by applying a stream of operations for each point in the index space.
- Geometric decomposition: An algorithm is organized by (1) dividing the key data structures within a problem into regular chunks, and (2) updating each chunk in parallel. Typically, communication occurs at chunk boundaries so an algorithm breaks down into three components: (1) exchange boundary data, (2) update the interiors or each chunk, and (3) update boundary regions. The size of the chunks is dictated by the properties of the memory hierarchy to maximize reuse of data from local memory/cache.

References

1. Catanzaro B, Keutzer K (2010) Parallel Computing with Patterns and Frameworks. ACM Crossroads, vol. 16, no. 5, pp. 22-27.
2. Our pattern language. http://parlab.eecs.berkeley.edu/wiki/patterns/patterns. Accessed 15 December 2009.
3. Keutzer K, Mattson T (2009) A design pattern language for engineering (parallel) software. Intel Technology Journal, Addressing the Challenges of Tera-scale Computing, vol.13, no. 4, pp. 6–19.
4. Asanovic K et al (2006) The landscape of parallel computing research: A view from Berkeley. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183.
5. Garlan D, Shaw M (1994) An introduction to software architecture. Tech. Rep., , Pittsburgh, PA, USA.
6. Maire M, Arbelaez P, Fowlkes C, and Malik J (2008) Using contours to detect and localize junctions in natural images. CVPR 2008, pp. 1–8.
7. Cortes C, Vapnik V (1995) Support-vector networks. Machine Learning, 20: 273–297.
8. Catanzaro B, Su B, Sundaram N, Lee Y, Murphy M, Keutzer K (2009) Efficient, high quality image contour detector. ICCV 2009, pp. 2381-2388.
9. Chang C, Lin C (2001) LIBSVM : a library for support vector machines. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm. Accessed 15 December 2009.
10. Catanzaro B, Sundaram N, Keutzer K (2008) Fast support vector machine training and classification on graphics processors. ICML 2008, pp 104-111.
11. Brox T, Malik J (2010) Large displacement optical flow:descriptor matching in variational motion estimation. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 99.

12. Brox T, Bruhn A, Papenberg N, Weickert J (2004) High accuracy optical flow estimation based on a theory for warping. ECCV 2004, pp. 25–36.
13. Baker S, Scharstein D, Lewis J, Roth S, Black M, Szeliski R (2007) A database and evaluation methodology for optical flow. ICCV 2009, pp. 1–8.
14. Sundaram N, Brox T, Keutzer K (2010) Dense Point Trajectories by GPU-accelerated Large Displacement Optical Flow. ECCV 2010, pp. 438–451.
15. Zach C, Gallup D, Frahm J M (2008) Fast gain-adaptive KLT tracking on the GPU. CVPR Workshop on Visual Computer Vision on GPU's.
16. Sand P, Teller S (2008) Particle video: Long-range motion estimation using point trajectories. International Journal of Computer Vision, pp. 72–91.
17. Wang L, Wang L, Wen M, Zhuo Q, Wang W (2007) Background subtraction using incremental subspace learning. ICIP 2007, vol. 5, pp. 45–48.
18. Demmel J, Grigori L, Hoemmen M, Langou J (2008) Communication-optimal parallel and sequential QR and LU factorizations. Tech. Rep. UCB/EECS-2008-89.
19. Chong J, You K, Yi Y, Gonina E, Hughes C, Sung W, Keutzer K (2009) Scalable HMM-based inference engine in large vocabulary continuous speech recognition. ICME 2009, pp. 1797-1800.
20. You K, Chong J, Yi Y, Gonina E, Hughes C, Chen Y, Sung W, Keutzer K (2009) Parallel scalability in speech recognition: Inference engine in large vocabulary continuous speech recognition. IEEE Signal Processing Magazine, 26(6): 124-135.
21. Chong J, Gonina E, Yi Y, Keutzer K (2009) A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit. Proceeding of the 10th Annual Conference of the International Speech Communication Association, pp. 1183 – 1186.
22. Chong J, Gonina E, You K, Keutzer K (2010) Exploring Recognition Network Representations for Efficient Speech Inference on Highly Parallel Platforms. Proceedings of the 11th Annual Conference of the International Speech Communication Association, pp. 1489-1492.
23. Candès E J (2006) Compressive sampling. Proceedings of the International Congress of Mathematicians.
24. Lustig M, Alley M, Vasanawala S, Donoho D L, Pauly J M (2009) Autocalibrating parallel imaging compressed sensing using L_1 SPIR-iT with Poisson-Disc sampling and joint sparsity constraints. ISMRM Workshop on Data Sampling and Image Reconstruction.
25. Murphy M, Keutzer K, Vasanawala S, Lustig M (2010) Clinically Feasible Reconstruction for L1-SPIRiT Parallel Imaging and Compressed Sensing MRI. ISMRM 2010.
26. Dixon M, Chong J, Keutzer K (2009) Acceleration of market value-at-risk estimation. Workshop on High Performance Computing in Finance at Super Computing.
27. Worth B, Lindberg P, Granatir (2009) Smoke: Game Threading Tutorial. Game Developers Conference.
28. Cocke J, Schwartz J T (1970) Programming languages and their compilers: Preliminary notes. Courant Institute of Mathematical Sciences, New York University, Tech. Rep.
29. Kasami T (1965) An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA.
30. Pollack F (1999) Microarchitecture challenges in the coming generations of CMOS process tech-nologies. MICRO-32.
31. Gustafson J L (1988) Reevaluating Amdahl's Law, CACM, 31(5): 532-533.
32. Luszczek P, Bailey D, Dongarra J, Kepner J, Lucas R, Rabenseifner R, Takahashi D (2006) The HPC Challenge (HPCC) benchmark suite. SC06 Conference Tutorial.
33. Sundaram N, Raghunathan, Chakradhar S (2009) A framework for efficient and scalable execution of domain specific templates on GPUs. IEEE International Parallel and Distributed Processing Symposium.
34. Catanzaro B, Kamil S, Lee Y, Asanovic K, Demmel J, Keutzer K, Shalf J, Yelick K, Fox A (2009) SEJITS: Getting productivity and performance with Selective Embedded JIT Specialization. Programming Models for Emerging Architectures.
35. Catanzaro B, Garland M, Keutzer K (2010) Copperhead: Compiling an Embedded Data Parallel Language. Tech. Rep. UCB/EECS-2010-124.

Chapter 5
The Case for Message Passing on Many-Core Chips

Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob Van Der Wijngaart

The debate over shared memory vs. message passing programming models has raged for decades, with cogent arguments on both sides. In this paper, we revisit this debate for multicore chips and argue that message passing programming models are often more suitable than shared memory models for addressing the problems presented by the many-core era.

The many-core era is different. The nature of programmers, the nature of applications, and the nature of the computing substrate are different for multicore chips than the traditional parallel machines that drove parallel programming developments in the past. For example, while traditional parallel computers were programmed by highly trained computational scientists, multicore chips will be programmed by mainstream programmers with little or no background in parallel algorithms, optimizing software for specific parallel hardware features, or the theoretical foundations of concurrency. Hence, multicore programming models must place a premium on productivity and must make parallel programming accessible to the typical programmer. Similarly, although the history of parallel computing is dominated by highly specialized scientific applications, multicore processors will need to run the full range of general purpose applications. This implies a drastically increased diversity in the nature of applications and an expanded range of optimization goals. This will heavily impact the choice of the programming model for multicore chips. The programming models for multicore architectures should also be capable of adapting to and exploiting asymmetry (by design and accident) in processing cores. We argue that the above goals are often better served by a message passing programming model than programming models based on a shared address space.

R. Kumar (✉)
University of Illinois at Urbana-Champaign (UIUC), IL, USA
e-mail: rakeshk@illinois.edu

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_5,

5.1 Metrics for Comparing Parallel Programming Models

To compare shared memory models and message passing models, we could take the familiar approach of defining a series of benchmarks, thereby turning this into a quantitative performance effort. However, it is difficult to distinguish the relative impact of the programming models from the relative quality of the implementations of the underlying runtime systems in such a comparison. A true comparison should deal with qualitative "human factors" and how they impact the programming process. We believe that a fair comparison of programming models must consider the end-to-end cost of the full lifecycle of a parallel program. The full lifecycle can be summarized as:

- Write the parallel program
- Debug the program and validate that it is correct
- Optimize the program
- Maintain the program by fixing bugs, porting to new platforms, adding features, etc.

A head-to-head comparison of the programming models for different stages of the program lifecycle will allow us to make qualitative conclusions about the relative efficacy of the programming models. We modified the cognitive dimensions from [3] to define a set of concrete metrics for our comparisons:

Generality: The ability to express in the programming model any parallel algorithm, such that a comparable level of concurrency as embodied by the algorithm materializes on the execution platform.

Expressiveness: Does the programming model help programmers express the concurrency in their problem succinctly, safely, and clearly for the classes of parallel algorithms for which the model was designed? An expressive programming model provides concise abstractions that help a programmer identify concurrent tasks and specify how data is shared (or decomposed) between tasks. Expressiveness does not imply generality.

Viscosity: Does the programming model let a programmer make incremental changes to a working program? If not, the risk of adopting the programming model is high. Viscosity includes the following aspects:

- Is it possible to gradually introduce concurrency into an original serial version of a program? Usually, this is not the case if the model implies a new language.
- How much effort is required to add or change functionality of an existing parallel code?

Composition: Does the programming model provide the isolation and modularization needed to support programming by composing parallel modules?

Validation (correctness): Is it easy to introduce cognitive slips when creating a program thereby introducing errors into the code? Can the program's correctness be reasonably validated? How difficult is it to find and remove bugs? Bugs that do not manifest themselves each time a code is run are difficult to find and remove. Such bugs can be due to nondeterminism, or to the fact that there may be a big gap between formal specification and implementation of the programming model.

Portability: Does the programming model let a programmer write a single program that can be recompiled and mapped efficiently onto all systems that are relevant to the target user community? This includes the potential for support of heterogeneous systems.

Of these metrics, composition and validation will be particularly important as many core chips become ubiquitous. Composition, the ability to build complex applications by composing smaller modules, is the cornerstone of modern software development. It must be supported in parallel software if we hope to migrate our software onto multicore systems.

As for validation, these costs often exceed system acquisition and software creation costs, a situation that will only worsen as more and more software being produced exploits parallelism. Anecdotal evidence of the difficulty of validating parallel software abounds, we merely cite a single source [8]:

> "We wrote regression tests that achieved 100 percent code coverage. The nightly build and regression tests ran on a two processor SMP machine, . . . No problems were observed until the code deadlocked on April 26, 2004, four years later."

Section 5.3 discusses how message passing programming models fare against shared memory models for the above metrics.

5.2 Comparison Framework

To evaluate different parallel programming models, we need to define a framework that captures a programming model's impact on the design and implementation of the most common parallel algorithms. Our comparison framework consists of different categories of parallel algorithms strategies and different algorithm patterns within each category. Following [2], we define the following three distinct strategies for parallel algorithm design:

- *Agenda parallelism:* Parallelism is expressed directly in terms of a set of tasks
- *Result parallelism:* Parallelism is expressed in terms of the elements of the data structures generated in the course of the computation.
- *Specialist parallelism:* Parallelism is expressed in terms of a collection of tasks each of which is specialized to a distinct function. In other words, data flows between a set of specialized tasks that execute concurrently.

This provides the top level structure of our framework. In Table 5.1, we show some of the more common patterns [10] associated with these algorithm strategies.

These patterns are well known by experienced parallel programmers (details are available in [6, 9]). The framework is not complete, but we submit that it covers the broad cross-section of the most important algorithms.

Using these patterns combined with our earlier metrics, we can turn our intuition about a programming model into specific (and testable) hypothesis about why different programming models dominate.

5.3 Comparing Message Passing and Shared Memory

We start with two generalizations concerning message passing vs. shared memory programming models. These concern validation and composition. To compose software modules, you must assure isolation of the modules. Interaction can only be allowed to occur through well-defined interfaces. To validate a program, you must assure that every legal way the operations in all active threads can interleave produce a correct answer. Both of these metrics are compromised by a shared address space. Message passing by design provides a mechanism of isolation since the threads or processes in a computation by definition execute in their own address spaces. As for validation, the message passing programmer only needs to check the allowed orderings of distinct message passing events. In a shared address space programming model where all threads access a single address space, proving a program to be race free has been shown to be an NP complete problem [7]. Hence, regardless of the type of parallelism involved, we assert that message passing has a strong advantage in terms of the ease with which a program can be validated and the ability to support software composition (Table 5.1).

In the remainder of this section, we will work through the algorithm design patterns described in Sect. 5.2 using the metrics we defined in Sect. 5.1 to compare shared memory and message passing models based on software features and actions required of programmers. The results are summarized in Table 5.2.

5.3.1 Agenda Parallelism

Design patterns associated with the "agenda parallelism strategy" are expressed directly in terms of tasks. The two cases differ in how the tasks are created; either directly as a countable set (task parallelism) or through a recursive scheme (Divide and conquer).

For the task parallelism pattern, both the message passing and the shared address space programming models are highly expressive and are general enough to cover most algorithms associated with this pattern. The message passing programming model is particularly well suited since data decomposition is typically a straightforward extension of the decomposition of the problem into a set of tasks. This means that the ease of validation common to distributed memory environments is easy to exploit with message passing, task parallelism problems.

Table 5.1 Brief taxonomy of parallel algorithms

Parallel algorithm strategies	Algorithm design patterns	
Agenda parallelism	Task parallelism	Divide and conquer
Result parallelism	Geometric decomposition	Data parallelism
Specialist parallelism	Producer/consumer (pipeline)	Event-based coordination

Table 5.2 Comparing message passing (Msg) and shared memory (Shar) programming models for design patterns from Table 5.1

	Agenda parallelism		Result parallelism		Specialist parallelism	
Metrics	Task parallel	Divide and conquer	Geometric decomp	Data parallel	Pipeline	Events
Generality	=	Shar +	=	Shar+	Msg +	Msg +
Expressiveness	=	Shar +	=	Shar +	Msg +	Msg +
Viscosity	Shar +	Shar +	Shar +	Shar +	Msg +	=
Composition	Msg +	Msg +	Msg +	Msg+	Msg +	Msg +
Validation	Msg +	Shar +	Msg +	Msg +	Msg +	=
Portablity	Msg +	Msg +	Msg +	Msg +	Msg +	Msg +

A "+" indicates when a model dominates for a given case. An "=" indicates that the two models are roughly equivalent for that particular case

The divide and conquer design pattern can be mapped onto message passing and shared address space models. These algorithms, however, are difficult to express with a message passing model. The problem is that as a task is recursively divided into a number of smaller tasks, the data associated with the individual tasks must be analogously decomposed. Programming models that require explicit data decomposition are difficult to apply when tasks are created so dynamically. Shared address space programming models, however, avoid this problem altogether since all threads have access to the shared data space. Furthermore, a key feature of implementations of divide and conquer algorithms is the need to dynamically balance the load among units of execution. For example, if tasks are managed in queues, it is possible that one unit of execution will run out of work. If it only needs to steal work descriptors from a neighboring queue without the need to move data, these work-stealing algorithms are natural to express. This is clearly the case for shared address space models, but not for message passing models.

Overall, both models are well suited for the Agenda parallelism strategy. The task parallelism design pattern works well for both types of programming model, but it slightly prefers the message passing model. For the divide and conquer pattern, however, the shared address space model is substantially better suited.

5.3.2 Result Parallelism

Design patterns associated with the result parallelism strategy center on how data is decomposed among the processing elements of a system. In most cases, the decomposition is well suited to a static decomposition or if dynamic, the dynamic structure is well defined algebraically and well suited to explicit data management schemes. Hence, these algorithms work well with message passing and shared address space programming models.

The classic "result parallelism" pattern is geometric decomposition. Message passing models have been used extensively with this pattern. The sharing of data is

explicit through messages, making geometric decomposition programs that utilize a message passing programming model both robust and easy to validate. Shared address space programming models work well also, but since race conditions are possible due to the fact that data is "shared by default," these programs can be difficult to validate.

Message passing program with geometric decomposition patterns are also highly portable. Since it is natural in these problems to define how data is shared between processes, the programming models are highly portable, allowing easy movement between shared memory and distributed memory systems.

Data parallel algorithms follow a similar analysis. They work well with message passing and shared address space model. Shared address space models, however, have a slight edge over message passing, however, because they do not require complicated data movement operations when collective operations are encountered. This is only a slight advantage, however, since the most common collectives are included in message passing libraries,

Overall, both models work well for three algorithm strategies. The message passing model, however, has a slight edge due to the greater ease of validating a program once written.

5.3.3 Specialist Parallelism

These algorithms can be challenging for message passing and shared address space programming models. The essential characteristic of the design patterns associated with the specialist parallelism strategy is that data needs to flow between specialized tasks.

For the pipeline algorithms, both models work, but the message passing provides more disciplined movement of data between stages. Messages are a natural way to represent the flow between stages in the pipeline making message passing programming models both expressive and robust. Shared address space programming models work, but they require error-prone synchronization to safely move data between stages. For an API that lacks point-to-point synchronization (such as OpenMP), this can lead to the need to build complicated synchronization protocols that depend on the details of how a flush works. Even expert OpenMP programmers find flush challenging to deal within all but the most trivial cases [4].

These problems are even worse for the event-based coordination algorithms. Message passing models work but robustness is compromised since the event models require anonymous and unpredictable flow of messages between processes. This compromises the robustness and validation properties and creates one of the few situations, where race conditions can be introduced in a message passing program. The key is to use a higher level model to apply discipline to how messages are used in these algorithms. For example, an actors model maps well onto event-based coordination algorithms. Actors is by its nature a message passing model. It can be implemented in a shared address space, but it requires complex

synchronization protocols and can lead to programs that are difficult to validate. The table below summarizes how the two models fare against each other for different metrics.

5.4 Architectural Implications

Programming models place requirements on hardware that supports them. A shared memory programming model, to run efficiently, requires hardware support. In practice, this comes down to the question of hardware supported cache coherence.

As the number of cores and the complexity of the on-chip networks grow, overhead in service of the hardware cache coherency protocol limits scalability. This is obvious for snooping protocols, but it is also the case for more scalable directory based protocols. For example, each directory entry will be at least 128 bytes long for a 1024 core processor supporting fully mapped directory-based cache coherence. This may often be larger than the size of the cache-line that a directory entry is expected to track. As another example, writes in a sequentially consistent shared memory processor may not proceed until all the shared lines have been invalidated, even the ones residing in cores that may be 10's of hops away.

Hence, as the number of cores increases, the overhead associated with the cache coherency protocol grows, often superlinearly. In particular, the additional cost due to the cache coherency protocol as each core is added to a many core chip grows. This increasing cost per core means that as the core counts grow, a "cache coherency wall" eventually limits the ability of a program to extract increased performance from the system.

Compare this to the situation for a many core chip that does not support cache coherency; a chip optimized to support message passing. Without cache coherency protocols, there is no fundamental overhead that grows as cores are added to the chip. As an example of a processor designed for message passing, consider the 48-core SCC processor [5]. The SCC processor consists of 24 tiles connected by a 6×4 two-dimensional mesh. This is a large chip with 1.3 Billion transistors manufactured with a 45 nm high K CMOS technology. The power for this chip is variable (under programmer control) and ranges from 25 to 125 W. Each tile in the SCC processor contains a pair of second generation Pentium® cores (the P54c [1]). Each core has its own L1 and L2 caches and implements the standard memory model associated with the P54c processor. The cores share a connection to mesh interface unit, which connects to a router on the on-die network. The tile also contains a 16-kilobyte memory region called a "message passing buffer." The message passing buffers on each tile combine to provide an on-die shared memory. This shared memory, however, lacks any hardware support for cache coherence between cores and therefore does not add overhead that grows as the number of cores grows.

The SCC processor contains four DDR3 memory controllers providing the processor with 16–64 Gigabytes of DRAM. This DRAM can be dynamically

configured under programmer control as private memory (coherent with the L1 and L2 caches on an individual core) or as shared memory. While this memory is shared, however, there is no cache coherence maintained between multiple cores for this shared DRAM. In addition, a router is connected to an off-package FPGA to translate the mesh protocol into the PCI express protocol allowing the chip to interact with a PC serving as a management console.

The SCC processor was designed with message passing in mind. The message passing protocol is one-sided; a core "puts" an L1 cache line into the message passing buffer, and another core "gets" that cache line and pulls it into its own L1 cache. Using this mechanism, cores implement message passing by explicitly moving L1 cache lines around the chip.

For a chip utilizing the SCC architecture, as cores are added, the overheads associated with memory accesses do not fundamentally grow. Since there are no directories or broadcasts to maintain the state of a shared cache line, as cores are added with the SCC processor, the base overhead remains fixed. As cores interact, communication overheads obviously are present. But they are fixed (for nearest neighbor communication patterns) or they grow as the square root of the number of cores for communication between cores at opposite corners of the 2D mesh. Even this worse-case scenario presents a much slower rate of growth in the communication overhead compared to that found on chips based on a cache coherent shared address space. Hence, a many core processor designed around the needs of a message passing programming model avoids the "coherency wall" allowing these many core chips to scale to much larger numbers of cores.

5.5 Discussion and Conclusion

Table 5.2 summarizes our comparisons of shared memory and message passing programming models. We consider a range of design patterns for each of our metrics.

As we indicated earlier, message passing programming models have distinct advantages due to the relative ease of validation and the fact they support the isolation required for composition.. Furthermore, as we pointed out in the previous section, a message passing programming model is more portable as well due to the fact the model places fewer constraints on the hardware to support the model. The other metrics present a more mixed picture with sometimes message passing and other times shared memory models coming out ahead. So why is message passing perceived as unsuitable for mainstream programmers and therefore largely neglected for many core chips?

We believe this arises from the fact that most comparisons of programming models center on the initial steps in writing a program; i.e., how expressive and general a programming model is. We see in Table 5.2 that for the most common parallel algorithm classes (Agenda Parallelism and Result Parallelism) shared memory programming models have an advantage. In some cases, these advantages

can be quite stark. This often leads to disqualification of message passing upfront, since the most salient first impression that programmers have of a programming model is its expressiveness. Higher expressiveness is often associated with higher programmer productivity. However, validation and composition constitute a very large portion of the downstream cost of an application's lifecycle. The shared memory programmer trying to validate a program and understand its composition with other software modules must understand the underlying memory model of the system; a task that even challenges experts in the field. This makes those costs much greater for shared memory models than a message passing model.

When you look at the full software lifecycle and the full range of metrics (not just expressiveness), we submit that message passing models are more suitable than shared memory models for a large class of applications. Hence, message passing models are an important, if not the only alternative for programming multicore and manycore chips. The benefits only increase as the number of cores and the complexity of the network on a processor chip increase.

References

1. D. Anderson, T. Shanley, *Pentium Processor System Architecture*, Addison Wesley, MA, 1995
2. N. Carriero, D. Gelernter, How to Write Parallel Programs: A Guide to the Perplexed. ACM Computing Surveys, 21(3), pp. 323–357, 1989
3. T.R.G. Green, M. Petre, Usability Analysis of visual Programming Environments: a "Cognitive Dimensions" framework, Journal of Visual Languages and Computing, 7, pp. 131–174, 1996
4. J.P. Hoeflinger, B.R. de Supinski, The OpenMP memory model. In: Proceedings of the First International Workshop on OpenMP – IWOMP 2005, 2005
5. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De1, R. Van Der Wijngaart, T. Mattson, A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS, Proceedings of the International Solid-State Circuits Conference, Feb 2010
6. K. Keutzer, T.G. Mattson, A design pattern language for engineering (Parallel) software, Intel Technology Journal, pp. 6–19, 2009
7. P.N. Klein, H-I Lu, R.H.B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35, pp. 321–345, Springer, Berlin, 2003
8. E.A. Lee, The problem with threads, *IEEE Computer*, 29(5), pp. 33–42, 2006
9. T.G. Mattson, B.A. Sanders, B.L. Massingill, Patterns for Parallel Programming, Addison Wesley software patterns series, 2004
10. M.J. Sottile, T.G. Mattson, C.E Rasmussen, Introduction to Concurrency in Programming Languages, CRC, FL, 2009

Part II
"Reconfigurable Hardware in Multiprocessor Systems"

Chapter 6
Adaptive Multiprocessor System-on-Chip Architecture: New Degrees of Freedom in System Design and Runtime Support

Diana Göhringer, Michael Hübner, and Jürgen Becker

Abstract The requirements for a processor, in terms of its characteristics such as RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer), bitwidth, instruction set, and for the communication and memory bandwidth differ for each application to be implemented. Furthermore, the required characteristic can be different at runtime, because the application has to react to the demands of the environment. Image processing is a good example for this scenario, because this application domain needs to adapt, depending on the content of the camera frames. Integrated, e.g., in a robot, the time variant requirements for the image processing applications are obvious. Sometimes gestures, obstacles, moving targets, etc. need to be detected within a high-resolution picture obtained by one or more cameras. For such applications, a novel Runtime Adaptive Multi-Processor System-on-Chip (RAMPSoC) was invented to provide an adaptive hardware architecture at design- and at runtime. This way, new degrees of freedom in system design and runtime support are provided. To program such a flexible multiprocessor system, an efficient design methodology is of high importance to hide the complexity of the underlying hardware. In addition, a runtime operating system is needed to handle the resource management and the runtime scheduling of the applications. In this chapter, the hardware architecture, the design methodology, and the runtime operating system of RAMPSoC are described. Furthermore, a brief overview about reconfigurable computing and dynamic and partial reconfiguration are given.

Keywords MPSoC (Multiprocessor System-on-Chip) · FPGA (Field Programmable Gate Array) · NoC (Network-on-Chip) · Reconfigurable computing · Dynamic and partial reconfiguration · Design methodology · HW/SW Codesign · Operating system

D. Göhringer (✉)
Fraunhofer IOSB, Gutleuthausstr. 1, 76275 Ettlingen, Germany
e-mail: diana.goehringer@iosb.fraunhofer.de

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_6,

6.1 Introduction

High-performance computing applications, such as image processing or bioinformatics applications, are still limited due to insufficient processing power. In former times, the approach was to increase the clock rate of a processor. This resulted in the disadvantage of higher power consumption. Nowadays, the approach has shifted toward increasing the number of processors while keeping the clock rate stable or even reducing it. This way, the power consumption does not increase that dramatically.

Most algorithms used in high-performance computing have a high inherent parallelism, which can be exploited by such a multiprocessor system. The disadvantage is that the hardware architecture of the multiprocessor system is fixed at design and at runtime. This means that the processor architecture and the communication infrastructure as well as the memory architecture are mostly optimized for one application scenario. Therefore, the user has to carefully choose an appropriate multiprocessor system for its application. And then the user needs to partition the application for the chosen multiprocessor system. This application integration has to follow the given hardware architecture of the multiprocessor system, which often leads to an inefficient task allocation and therefore an unequal workload. Especially, if versatile algorithms are used as in image processing, some will map better and others will map worse on the chosen architecture. Besides homogeneous (also called general purpose) multiprocessor systems (see [23] and [21], there exist also heterogeneous multiprocessor systems (for some examples, see [29]). Due to their heterogeneity, they achieve a good performance/power tradeoff for their target applications, but other applications can only be mapped suboptimal. In general, the limitation of all multiprocessor systems is the lack of adaptivity at design and at runtime.

Another option to exploit this parallelism and therefore increase the computing power is the pure hardware implementation using an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). This way an even finer parallelism compared to a processor array can be exploited and a good performance with low power consumption can be achieved. While the ASIC implementation achieves the best performance/power tradeoff, it is very costly due to the mask process and also very inflexible due to the fixed hardware architecture. FPGAs offer a more flexible solution, because they can be reconfigured with a new functionality and can therefore be reused for a different application. Furthermore, some FPGA vendors, such as Xilinx offer a special feature called dynamic and partial reconfiguration. This means a part of the FPGA hardware can be modified at runtime, while the other parts stay operative and are not disturbed. This is useful if, for example, an image processing filter on an FPGA needs to be exchanged, but the connection with the camera and a monitor must be preserved. If the whole FPGA would be reconfigured, frames from the camera would be lost. Using partial dynamic reconfiguration instead assures that the frames of the camera will not be lost, because the camera interface module on the FPGA stays operative, while only the image processing module is reconfigured. Therefore, FPGAs with this dynamic and partial reconfiguration feature are extremely flexible. This way, they offer a

new degree of freedom called computing in time and space [19]. Another benefit is that only the currently needed functionality has to be implemented at a given point in time. This way a smaller FPGA can be used and the overall power consumption can be decreased. Their disadvantage is the programmability. While it is already difficult to program multiprocessor systems, the programming of an FPGA is even more difficult, because to achieve a maximum of performance hardware description languages, such as VHDL or Verilog, have to be used. There exist some ESL (Electronic System Level) design tools that ease the programmability by offering special C-to-Gates or Matlab-to-VHDL tool flows. But this is still a wide research topic and so far the power of these tools is limited and the input language has often several constraints and requires special pragmas. For most engineers with software or an application background, it is more comfortable and less time-consuming to use a software implementation or to try to program a multiprocessor system even though the achieved performance/power is worse compared to an FPGA implementation.

In this chapter, a new runtime adaptive multiprocessor system-on-chip called RAMPSoC [12] is presented, which tries to overcome the mentioned challenges by combining the best of both worlds: the reconfigurable computing and the multiprocessor world. The RAMPSoC approach consists of an FPGA-based heterogeneous network of processors with distributed memory and closely coupled hardware accelerators. It supports a faster designflow than a pure FPGA-based hardware implementation, because parts or even the complete algorithms can be implemented in software. Additionally, it is more flexible than the state-of-the-art multiprocessor systems, because it supports runtime and design-time adaptivity of the hardware architecture through exploitation of the dynamic and partial reconfiguration feature of the FPGA. Furthermore, a combination of processors and closely coupled hardware accelerators is used to map the control flow onto the processors and the dataflow intensive parts of the algorithm onto the hardware accelerators. This way high performance can be achieved by keeping the power consumption low. This extension of the multiprocessor methodology leads to a "Meet-in-the-Middle"-approach at design- and at runtime, because both the application software and the hardware architecture can be adapted to fulfill the requirements of the application. This way, a new degree of freedom is provided for the system design as well as for the efficient task allocation onto the runtime adaptive processing elements.

This chapter is organized as follows: In Sect. 2, background information about reconfigurable hardware is provided. Related work in the field of reconfigurable multiprocessor systems is provided in Sect. 3. Section 4 gives an overview about the RAMPSoC approach. In Sect. 5, the hardware architecture of RAMPSoC together with a novel Network-on-Chip (NoC) called Star-Wheels NoC is presented. The novel design methodology of the RAMPSoC approach is presented in Sect. 6. Section 7 shows the special purpose operating system called CAP-OS (Configuration Access Port – Operating System), which is responsible for the runtime application scheduling, task allocation, resource management, and configuration of the complete RAMPSoC. Finally, the chapter is summarized by presenting the conclusions and an outlook for future improvements in Sect. 8.

6.2 Background: Introduction to Reconfigurable Hardware

Today, field programmable gate-arrays are used for a wide sector of applications. The usage in former times was focused on rapid-prototyping systems for integrating test systems. After the test-phase, an ASIC often replaced these chips for mass production. Decreasing prizes for FPGAs and increasing mask-costs for ASIC design, but also lower power-consumption of novel reconfigurable hardware and high flexibility opened a market for industry and a wide research area for scientific work. Particularly, the possibility of runtime reconfiguration, which is supported by some state-of-the-art FPGA architectures, enables to introduce novel ideas for adaptive hardware. FPGAs can be reconfigured many times for different applications as long as they are SRAM- or FLASH-based. Modern state-of-the-art FPGA devices such as Xilinx Virtex FPGAs additionally support a partial dynamic runtime reconfiguration, which reveals new aspects for the designer, who wants to develop future applications demanding an adaptive and flexible hardware. The idea here is to provide the required hardware function for an application, when data have to be processed. The idea to bring the function to the data stands in opposite to the traditional von Neumann approach, where a program counter points to the current instruction and induces the data to flow to the related function. Here, a data counter points to the next data-packet, which has to be processed (see [1]). The required function is provided on demand and at runtime to enable the required data processing. New approaches to create systems, which are able to manage configuration, are runtime adaptive systems. These systems use the flexibility of an FPGA by changing the configuration partially. Only the necessary functions are configured in the chip's configuration memory. On demand a function can be substituted by another while used parts stay operative as described in [28]. These system approaches include a controller-based module to schedule the reconfiguration and data transfer to the respective substitutable functional element. Additionally, this module controls the on-chip intercommunication bus to prevent an overhead of bus controller functionality as additional module. In the following subsections, a closer view to the basic methodologies and terminology of reconfigurable computing will be described.

6.2.1 Basic Concept of Runtime Reconfiguration

Reconfigurable hardware allows the introduction of a further degree of freedom for the design of embedded electronic systems. The flexibility of microprocessor-based systems lies in the adaptability of program code within the program memory. This approach of adaptation for different kind of applications can now be extended by adaptation of the hardware architecture of a chip. This feature was former exploited in some industrial applications to update the system without having high costs of ASIC redesign in case of a required hardware improvement. Especially, the high mask costs of modern ASIC production include a high risk of a possible redesign, which is sometimes not acceptable. The increasing complexity of the circuits and

the demand of flexibility of electronic systems with a simultaneously increased requirement of performance forces to use reconfigurable hardware for current and future target platforms.

The approach mentioned above, which exploits reconfigurability, is extended by the usage of runtime reconfigurable hardware architectures. The flexibility through adaptation of the architecture is therefore shifted from the design-time to the runtime. Through this a new degree of freedom is achieved and can be clarified with the terminology "Computing in Time and Space." The coordinate system with the time axis, which describes the runtime or processing time of a system, is extended by the dimension of parallelism. This parallelism is directly derived from the configurable area of the target platform (here the FPGA).

Figure 6.1 shows this parallelism exemplarily with two running tasks. This attribute alone would not separate a hardware reconfigurable architecture from a microprocessor-based system approach, since modern processors can run more than one task in parallel due to the availability of more than one arithmetic logic unit (ALU). Regarding the nonsequential data processing of algorithms and functions realized as hardware, the parallelization in time and space enables the benefit of performance enhancement in these two dimensions.

Runtime reconfigurable hardware now enables the introduction of another dimension for a certain application, which needs to be implemented. Concerning the configurable chip area as a physical medium for integration of algorithms and functions at runtime, Fig. 6.2 shows the distribution of the tasks to different positions on the chip area. In the picture, the two axes X and Y represents the dimension of the chip area. In the example task 2 requires a larger amount of chip area than task 1. Furthermore, the two points in time t_1 and t_2 and the related cuts through the graphic parallel to the XY-plane can show the task distribution on the reconfigurable area. Figures 6.3 and 6.4 visualize these cuts, which represents the reconfigurable area of the target chip.

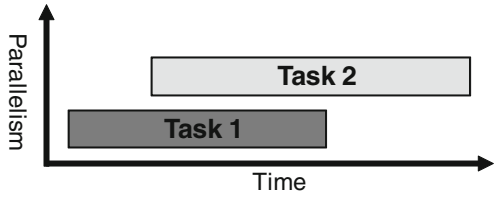


Fig. 6.1 Parallel task processing

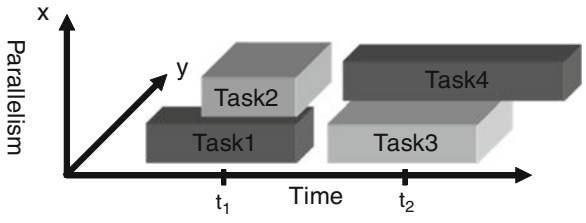


Fig. 6.2 Parallel task processing in time and space

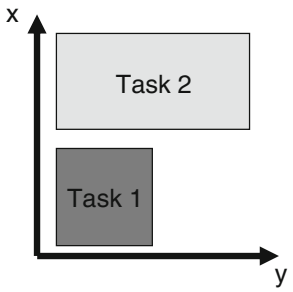


Fig. 6.3 Cut parallel to the *XY*-plane at time point t_1

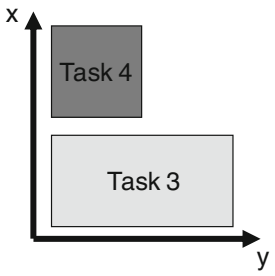


Fig. 6.4 Cut parallel to the *XY*-plane at time point t_2

As shown with this simple example, the chip area can be used to contain different tasks at different points in time and positions. The tasks can be started and terminated at different points of time and the area can be reused for other processes. This reuse now can be used for systems, whose data processing occur "on-demand," which means that the data processing will be initiated from either external trigger or internal requirements, e.g., forced by an operating system. This trigger initiates then the configuration of a task on the reconfigurable area of the architecture.

For this purpose, dynamic and partial reconfigurable FPGA architecture (e.g., Xilinx) will be used. These architectures provide the necessary runtime adaptivity capabilities, which are exploited by this system approach. One benefit by introduction of this system approach lies in the reduction of chip size, because only currently required functions are utilizing the reconfigurable area, while idle tasks are loadable on-demand from an external memory.

6.2.2 Basic Concept of Runtime Reconfiguration and Classification of Configurable Granularity

In the year 1960, the basic and fundamental idea of exchangeable hardware was described. The idea published by Gerald Estrin [11], which nowadays has the name reconfigurable hardware, could not be realized due to restrictions in technology. Initially, around 30 years later, in the year 1989 a first prototype was presented by Bertin et al. [3]. The analysis for exploitability of the flexibility and adaptivity of

reconfigurable hardware architectures provides since then a distinctive field of research. The "Anti-Machine Paradigm" introduced by Reiner Hartenstein, which describes a datastream-based paradigm in opposite of the controlflow-based paradigm of traditional von Neumann architectures, shows the possibility of the novel approach [18]. The parallel processing of operation or tasks on the hardware and the option to move the operations to the data stands in opposite to the von Neumann approach, where data needs to be moved to the hardware operation. This novel option, the reconfigurable computing, is the basis of forward looking architectures in both the academic and the industrial world. A small example should point this out.

Figure 6.5 shows a simple dataflow graph for an arithmetic function of several operations. Through the oval shaped forms within the picture, a possible clustering and scheduling of the operations within this areas in parallel at one point of time is suggested. Respectively, one cluster of different operations can be processed in one time-step. In a sequential (processor-based) processing architecture, the data, which need to be processed, have to be allocated to the ALU one after another. For this purpose, additionally control cycles to adjust the ALU to the required operation are necessary.

Table 6.1 shows the comparison of the different realizations for the example dataflow graph in Fig. 6.5 with sequential and parallel processed data. The "|||" symbol indicates that the operations are parallelized, until the next ";" symbol. It is obvious that the parallel realized dataflow graph in comparison with the sequential realized graph calculates the result in less time-steps. In this example, the parallel realized algorithm delivers the result in four time-steps, whereas the sequential flow

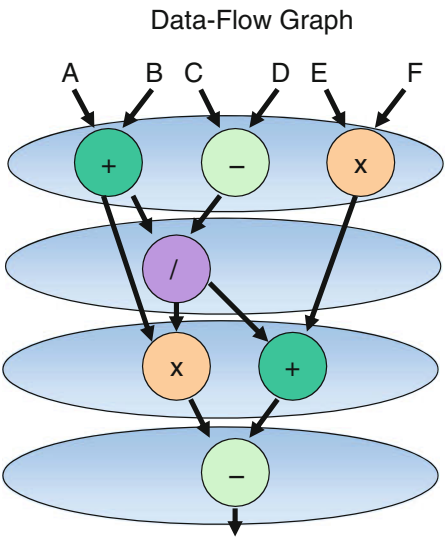


Fig. 6.5 Example data flow graph with spatial clustering (parallelization)

Table 6.1 Comparison of sequential and parallel data processing

Sequential data processing	Parallel data processing
tmp_1 = A + B;	tmp_1 = A + B |||
tmp_2 = C – D;	tmp_2 = C – D |||
tmp_3 = E*F;	tmp_3 = E*F;
tmp_4 = tmp_1/tmp_2;	tmp_4 = tmp_1/tmp_2;
tmp_5 = tmp_1*tmp_4;	tmp_5 = tmp_1*tmp_4 |||
tmp_6 = tmp_4 + tmp_3;	tmp_6 = tmp_4 + tmp_3;
Result = tmp_5-tmp_6;	Result = tmp_5-tmp_6;
Result = (A + B)/(C – D) + E*F – (A + B)/(C – D) + E*F	

delivers the data in seven time-steps (excluding additional time-steps for processor control). With this small example, a rough estimate of the performance improvement of 1.75 can be done, which shows the capability of parallelizing algorithms.

Since the first introduction of reconfigurable hardware, a steady process of optimization and specialization of COTS (Commercial off-the-shelf) chips can be monitored. Optimizations occur in the field of the integrated logic cells (configurable logic blocks), the routing resources and the reduction of power dissipation. Especially, the latter topic is a challenge for the manufacturers of reconfigurable architectures. This stands directly in relation to the competition with ASIC production, which is until now more efficient in power consumption. The goal here is to enter more and more markets, for example the market of mobile communication.

To specify the terminology of granularity, an introduction of the classification of the characteristics of reconfigurable hardware follows.

The terminology of granularity describes the size of the bitwidth of the smallest addressable unit within the reconfigurable architecture. Important in this case is that the size does not only apply to the size of the logic blocks. Also, the routing (connection) between the logic blocks has to be taken into account. The bitwidth, which can be adjusted, is first the bitwidth of the input signals of the smallest logic block and therefore the complexity of the functionality and second, the bitwidth of the connective structure between those logic blocks. It can be differentiated between three types of architecture: Coarse-grained, middle-grained, and fine-grained reconfigurable architectures (see [10] and [25]). However, the separation between those classes is not defined exactly. It can happen that a selected architecture cannot be classified into a single type of granularity.

Table 6.2 shows a rough guideline for a classification of the architectures. It is obvious, which difficulties occur, if the size of the interconnect width differs with the size of the bitwidth from the logic blocks. Also, this kind of architecture is described in academic work (see [27]).

In general, it can be noticed that the degree of granularity is in direct relation to its flexibility and ubiquitous application. This is the result of the possibility to influence the routing and content of the logic blocks at the bit level. Due to this purpose, fine-grained reconfigurable architectures are suited properly for integration of algorithms which process data at the bit level.

Basically, it is possible to integrate every logic and algorithmic function on fine-grained reconfigurable architectures. However, it has to be considered that

Table 6.2 Classification of reconfigurable hardware in terms of bitwidth

Bitwidth	Classification to architecture class
$<$4 Bit	Fine-grained
$\leq$8 Bit	Middle-grained
$>$8 Bit	Coarse-grained

fine-grained reconfigurable architecture might have a high amount of hardware overhead and therefore costs in term or area and power consumption. Operations whose bitwidth exceeds the size of the selected architecture must be realized by connecting more than one logic block, which leads to an increased complexity for wiring. This increased complexity causes an inefficient exploitation of the chip area and an increased delay for communication and power consumption.

In comparison, coarse-grained reconfigurable architectures can handle a higher number of bits, which leads in the best case to an avoidance for the necessity of connecting other logic blocks. However, operations with a bitwidth smaller than the provided size need to be processed with the full size of available bits, which leads to an inefficient exploitation of the architecture and an increased power consumption. The described set of problems shows that the choice of the granularity in relation to the operation, which needs to be implemented, has a strong impact on the efficiency of the complete realization. Further details and examples for reconfigurable hardware with different granularity can be found in Hauck and DeHon [20].

6.3 Related Work

The resources on reconfigurable architectures such as the FPGAs from Xilinx and Altera have been increased a lot during the last years. Nowadays, not only logic blocks, but also DSP (Digital Signal Processing) cores, on-chip memory blocks and even some hard-IP processor cores are available. This way, complex systems, such as a MPSoC consisting of 20 or more cores on a single FPGA, are possible. A famous example for such a system is the research accelerator for multiple processors (RAMP) [6], which consists of several Berkeley emulation engine 2 (BEE 2) Boards [8] with each five big Xilinx Virtex-5 FPGAs. On each FPGA several 32-bit RISC processors, called Xilinx MicroBlaze,[1] are implemented to build a homogeneous manycore system. This system is used to investigate different application mapping strategies for future manycore systems. Due to the power consumption of these multiple FPGA boards, it cannot be used for embedded high-performance systems. Also, a runtime adaptation of the system is not supported so far.

Several research labs investigate reconfigurable MPSoCs. For example, Paulsson et al. [24] presented a system consisting of several Xilinx MicroBlazes, which supports the reconfiguration of the instruction memories.

[1] "Xilinx MicroBlaze Reference Guide"; Available at http://www.xilinx.com.

Claus et al. [9] and Bobda et al. [4] also developed FPGA-based multiprocessor systems. In both works, the processors are fixed, but the accelerators can be reconfigured using dynamic and partial reconfiguration. An additional example for such a multiprocessor system is the MORPHEUS chip [26], which consists of an ARM processor and three reconfigurable accelerators. Each accelerator has a different reconfigurable granularity, this means one is fine-, the other medium- and the third one is a coarse-grained reconfigurable device.

A different approach is the XiRisc reconfigurable processor [7], which consists of a VLIW (Very Large Instruction Word) RISC core with a runtime reconfigurable data path, called Pipelined Configurable Gate-Array (PiCoGA). By reconfiguring the PiCoGA, which is within the data path of the processor, the instruction set of the processor and therefore the instruction stream are reconfigurable. An additional example for such a VLIW processor with a runtime reconfigurable data path is the ADRES architecture [2].

In summary, the here mentioned multiprocessor systems are either completely static or only support the reconfiguration of either the instruction memory or the accelerators, while the processors themselves and the communication infrastructure are fixed and cannot be modified at runtime. Similarly, the here presented reconfigurable VLIW processors are single processors with a reconfigurable datapath, but additional processors cannot be added.

To the best of our knowledge, no other such holistic approach as RAMPSoC exists, which provides a higher degree of freedom by supporting the design-time and runtime adaptation of the communication infrastructure, the number and types of processors, the instruction memory of the processors, and the accelerators. This way, requirements such as performance and power consumption can be fulfilled more efficiently. Furthermore, the RAMPSoC approach provides the user with a design methodology and a runtime operating system, which both hide the complexity of the underlying hardware from the user.

6.4 The RAMPSoC Approach

The RAMPSoC approach combines the benefits of multiprocessor systems and reconfigurable architectures. Figure 6.6 shows the resulting Meet-in-the-Middle approach of RAMPSoC. This Meet-in-the-Middle approach extends the typical top-down tool supported application partitioning of state-of-the-art MPSoCs with a bottom-up hardware adaptation approach supported by state-of-the-art reconfigurable architectures such as Xilinx FPGAs. This is the first approach, which combines the simpler programming paradigm of MPSoCs with the runtime hardware adaptivity of reconfigurable architectures. This way the application partitioning and the definition of a starting MPSoC architecture are developed at design-time. Due to the configurability of FPGAs, an optimized MPSoC architecture for a given application can be defined at design time. Furthermore, by exploiting runtime reconfiguration, which is supported, e.g., by Xilinx FPGAs, the hardware of the MPSoC can

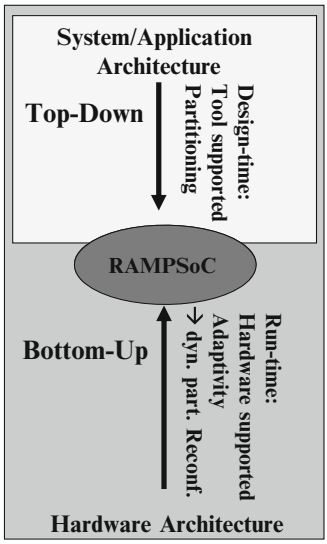


Fig. 6.6 The meet-in-the-middle approach of RAMPSoC

be adapted at runtime to achieve a good performance per Watt ratio for the application.
The RAMPSoC approach supports the runtime adaptation of the following hardware components:

- Number and type of processors
- Number and kind of accelerators
- Communication infrastructure

Of course, also the software executable files can be exchanged at runtime either by sending the new software tasks over the communication infrastructure or by using dynamic and partial reconfiguration to override the instruction and the data memory of a given processor.

Defining such an optimal multiprocessor system for a given set of applications is a multi-dimensional optimization problem. Even though the programming paradigm of multiprocessor systems is simpler than the one of reconfigurable architectures the programming of such a runtime adaptive MPSoC is very complex. Therefore, the complexity of the underlying hardware has to be abstracted using a novel design methodology, which guides the application programmer. Therefore, the following four abstraction layers shown in Fig. 6.7 have been introduced:

- MPSoC-Level
- Communication-Level
- Processor-Level
- Physical-Level

The abstraction layers are used for both the hardware system architecture and the design methodology of RAMPSoC.

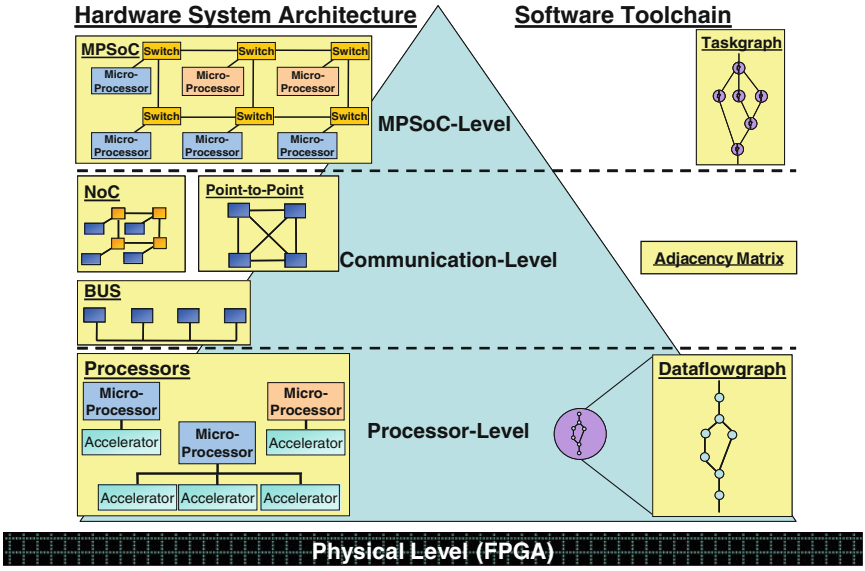


Fig. 6.7 The four abstraction layers of RAMPSoC, which are used to hide the complexity of the underlying hardware from the user

The MPSoC-Level is the highest level of abstraction. It is used by the user, who knows that his/her architecture will be mapped on an FPGA-based multiprocessor system, but details such as the processor types, the type of communication infrastructure, or the number and types of accelerators are hidden from the user. On the side of the design methodology, this layer represents the user application, which is given in C, C++, or C with MPI and which is transformed automatically into a task graph by the design methodology.

The Communication-Level represents from the hardware point of view a library of communication infrastructure such as buses, NoCs, Point-to-Point connections, or a combination of those. The design methodology selects on this abstraction level the required set of communication infrastructures for the given application based on the adjacency matrix. The adjacency matrix is the result of the partitioning of the task graph and shows the communication requirements of tasks, which are mapped on different processors.

The Processor-Level is a library of supported processors and accelerators. At this level, the design methodology explores the tasks of each processor and suggests appropriate processors and accelerators.

The processor-, accelerator-, and the communication infrastructure libraries can be extended, if needed. Currently, the following processors are supported: Xilinx MicroBlaze, IBM PowerPC405, Leon Sparc, and Xilinx PicoBlaze. For the communication infrastructure, it can be chosen between the Fast Simplex Links (FSLs) from Xilinx, Peripheral Local Bus (PLB), On-Chip Peripheral Bus (OPB), Circuit-Switched Runtime Adaptive Network-on-Chip (CSRA-NOC) [5], and the novel Star-Wheels

Network-on-Chip [16]. Several image processing accelerators such as Gauss, Sobel, Median, Sum of Absolute Differences (SAD), Normalized Squared Cross Correlation, Hotspot and Coldspot are supported. Furthermore, the design methodology supports the use of commercial C-to-FPGA tools (e.g., ImpulseC or CatapultC) to generate hardware accelerators for the computation intensive functions or loops of a task.

6.5 Hardware Architecture of RAMPSoC

In Fig. 6.8, a RAMPSoC system at one point in time connected over an incomplete version of the Star-Wheels Network-on-Chip is shown. As can be seen, the RAMPSoC is a heterogeneous MPSoC supporting different types of processors. Each processor can have several closely coupled accelerators. Instead of a complete processor, a Finite State Machine (FSM) combined with a hardware function can be used. For connecting sensors and actors, such as cameras and displays and also for a PCI connection with a hostPC with the communication infrastructure of the MPSoC, a component called Virtual-IO was developed. Virtual-IO further supports the splitting of incoming data, e.g., an image, into several tiles for different processors and also collecting the results from several processors before forwarding them to the display or the hostPC via PCI.

RAMPSoC uses a distributed memory approach to achieve a maximum speedup. Therefore, the software executables for each processor have to be small enough to fit in local on-chip memory, which can be accessed within a single cycle. If this is

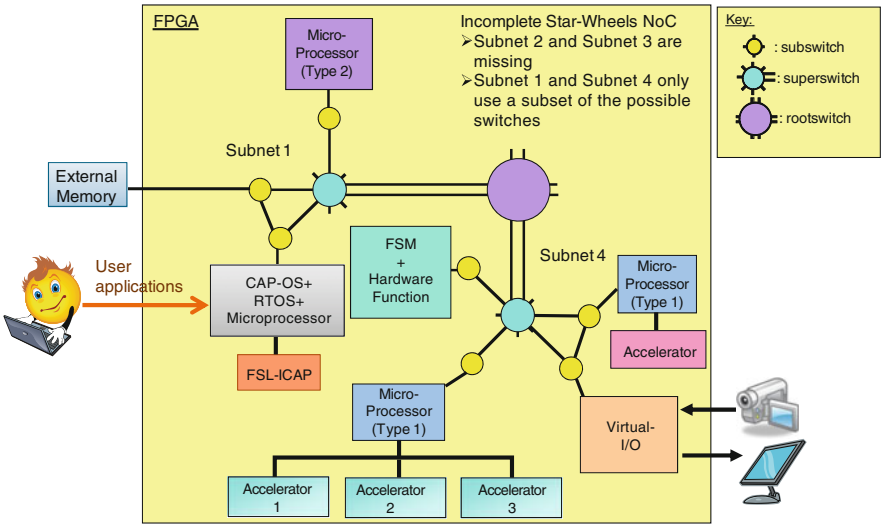


Fig. 6.8 A RAMPSoC system connected over an incomplete Star-Wheels Network-on-Chip at one point in time

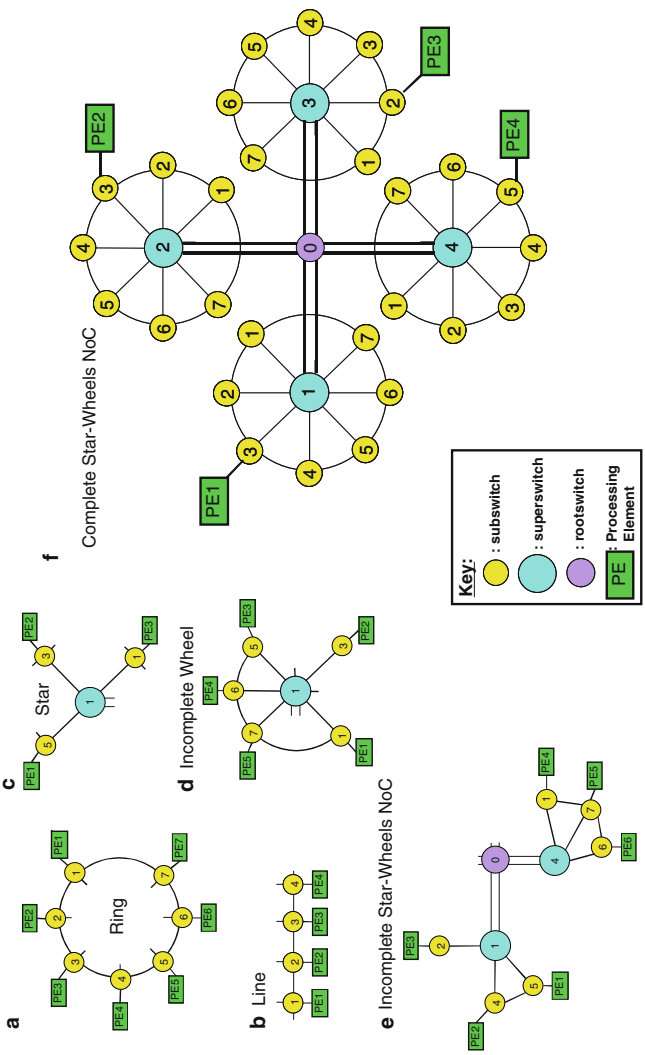


Fig. 6.9 Different topologies supported by the Star-Wheels Network-on-Chip. (**a**) Ring, (**b**) Line (**c**) Star, (**d**) Incomplete Wheel, (**e**) Incomplete Star-Wheels NoC, and (**f**) Complete Star-Wheels NoC topology

not feasible, a restricted number of processors can have access to external memory. Due to the longer latency and the limited external memory on commercial FPGA boards, this should be an exception, because it will reduce the achievable speedup. As already mentioned, different communication infrastructures are supported.

Figure 6.9 gives an overview of the different topologies supported by the Star-Wheels Network-on-Chip. The Star-Wheels NoC supports three different types of switches to achieve a good performance per area tradeoff. The smallest switch is the subswitch. It is used to connect the processing elements of the MPSoC to the NoC. The neighboring subswitches can directly communicate with each other. Non-neighboring subswitches use the superswitch to communicate with each other. Seven subswitches and one superswitch form a subnet, in which the communicating partners are closely connected with each other. To communicate between different subnets the rootswitch is required, which supports a limited number of communication links between different subnets. The Star-Wheels NoC is named after its heterogeneous topology consisting of up to four subnets using the novel Wheel topology, which are connected over a rootswitch using a Star topology as shown in Fig. 6.9f).

To provide a low latency and therefore a high throughput, as it is required for high performance computing applications such as image processing, the Star-Wheels NoC uses a synergy of circuit- and packet-switching communication protocol. Each switch has separate ports for circuit- and packet-switching. Packet-switching is used for control purposes and for establishing and freeing a communication channel between two processing elements. For exchanging data over the communication channels, the circuit-switching protocol is used. The network is self-adaptive and each switch recognizes at runtime, if a neighboring switch or PE has been removed, added, or exchanged. The Star-Wheels NoC is scalable and area efficient for FPGAs. Different clock domains are supported, which is an important feature for MPSoCs, due to the possibility of Dynamic Frequency Scaling. In [16], it was proven that the Star-Wheels NoC is deadlock free. Like the Virtual-IO and other components of RAMPSoC, the Star-Wheels NoC has been integrated into the high level design tool from Xilinx called Embedded Development Kit (EDK).

6.6 Design Methodology of RAMPSoC

To program such flexible hardware architecture, a novel design methodology is needed, which hides the complexity of the underlying hardware from the user by following the four abstraction layers of RAMPSoC. Figure 6.10 gives an overview of the RAMPSoC design methodology consisting of three phases:

- Phase 1: Software/Software Partitioning
- Phase 2: Hardware/Software Partitioning
- Phase 3: Implementation

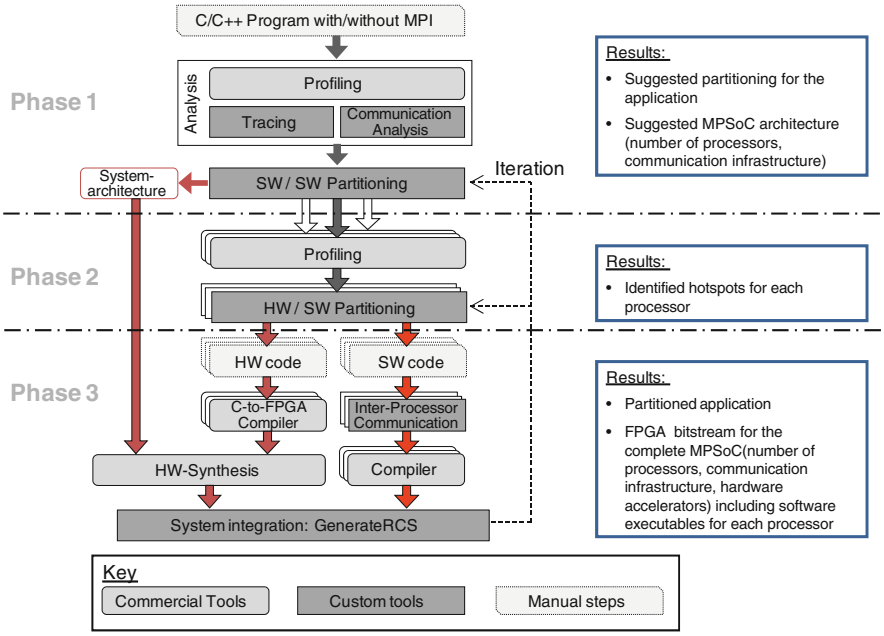


Fig. 6.10 Design methodology of RAMPSoC

In its current version, the design methodology is semiautomatic and uses a combination of commercial and custom tools as well as some minor manual steps. It can be used for user applications written in C, C++ without MPI or C with MPI.

In phase 1, the user applications are analyzed. Using a commercial profiling tool such as the AMD CodeAnalyst, the runtime of each function is measured. The call graph is generated and the communication between the different functions is analyzed. For this, two custom tools have been developed. The communication analysis tool currently supports C programs with/without MPI and is described in detail in [14]. The support of C++ programs is currently under development. The benefit of this tool is that it generates the call graph and automatically analyzes the communication requirements for both function calls and MPI commands. For these applications, the tracing library tool is not required. The tracing library is only needed for C++ applications, as these are not yet supported by the communication analysis tool. To analyze the communication requirements of C++ applications, either the novel neighborhood relation heuristic [13] can be used, or the communication requirements have to be analyzed manually. The timing results of the profiling, the call graph and the communication requirements are then used as parameters for the cost function of the Software/Software Partitioning tool. The Software/Software Partitioning is based on a heuristic approach using the hierarchical clustering algorithm. As shown in Fig. 6.11, hierarchical clustering is a multistep algorithm and generates a partitioning hierarchy. Each level of the hierarchy can be mapped to a specific number of processor, which is equal to the

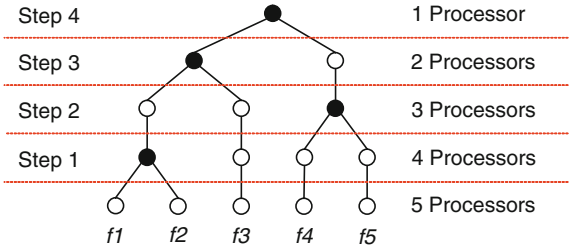


Fig. 6.11 Partitioning hierarchy of the hierarchical clustering algorithm and how this maps to MPSoC architectures

Closeness Function:

$$C(x,y) = \begin{cases} \omega_{MPI}\dfrac{MPI\_COM(x,\,y)}{T(x,\,y)} + \omega_{Call}\dfrac{Call\_COM(x,\,y)}{T(x,\,y)} \\ \dfrac{NH(x,\,y)}{T(x,\,y)}, \text{ if } MPI\_COM,\ Call\_COM \text{ unknown} \end{cases}$$

T(x, y):	Sum of the profiled runtimes of the two tasks to be clustered
MPI_COM (x, y):	Communication costs between two tasks communicating via MPI
Call_COM(x, y):	Communication costs between two tasks in the call graph
NH(x, y):	Proximity of two tasks based on the call Graph
ω_{MPI}:	Weighting factor for MPI communication
ω_{Call}:	Weighting factor for call graph communication

Fig. 6.12 Closeness function used to decide at each step of the hierarchical clustering about the clustering of two functions

number of clusters. If the desired number of processors is known, the hierarchical clustering algorithm can be stopped at that specific step.

Figure 6.12 shows the closeness function, which is used to evaluate the clustering of two functions/clusters for the hierarchical clustering. For the clustering, it is differentiated between MPI-based communication (MPI_COM) and communication costs of function calls (Call_COM). Both types are weighted differently using $\omega_{\text{MPI}} \cdot \omega_{\text{Call}}$, where $\omega_{\text{MPI}} + \omega_{\text{Call}} = 1$ and $\omega_{\text{MPI}} \leq \omega_{\text{Call}}$.

ω_{Call} is greater or equal ω_{MPI}, because two functions communicating over the call graph should be more likely clustered. The reason for this lies in the basic principles of MPI, which is a programming model that is used to exchange data between different processors. If the programmer therefore uses MPI to exchange information between two functions, this indicates that these two functions should be placed on different processors. The application programmer can adapt these weights depending on the needs of his/her application. If a normal C, C++ application without MPI is used, then ω_{MPI} should be set to 0 and ω_{Call} should be set to 1.

If MPI_COM and Call_COM are unknown, the clustering can still be done by using our custom proximity heuristic NH(x, y) [13].

As a result, the SW/SW Partitioning, based on the hierarchical clustering, suggests both an application partitioning and a system architecture for the user.

In phase 2, each cluster for each processor of the final system is profiled on a line-by-line basis, using a commercial profiling tool, such as AMD CodeAnalyst, to identify possible computation intensive code segments. The report generated by the profiler is then sent to a custom developed tool called Profile Analyzer. The Profile Analyzer calculates the timing for each function and loop and also the relation between the required execution time of the loops and their corresponding functions. It graphically shows the user this relationship and generates a text file with hotspots that could be outsourced into hardware accelerators. Figure 6.13 shows the different views and results of the Profile Analyzer tool. In Fig. 6.13a, a screenshot of the Profile Analyzer and the extracted timing analysis values for the loops and the functions are shown. One exemplary timing diagram showing the runtime of the loops relative to the total runtime of the corresponding function is shown in Fig. 6.13b. Finally, Fig. 6.13c shows the summary file listing all extracted hotspots.

Phase 3 is the integration and implementation phase of the design methodology. Based on the results of the previous phases, each cluster is separated into software and hardware code for the found hotspots. The code for the found hotspots has to be manually modified depending on the requirements/restrictions of the chosen C-to-FPGA tool such as ImpulseC. These modifications are minor and only at the C/C++

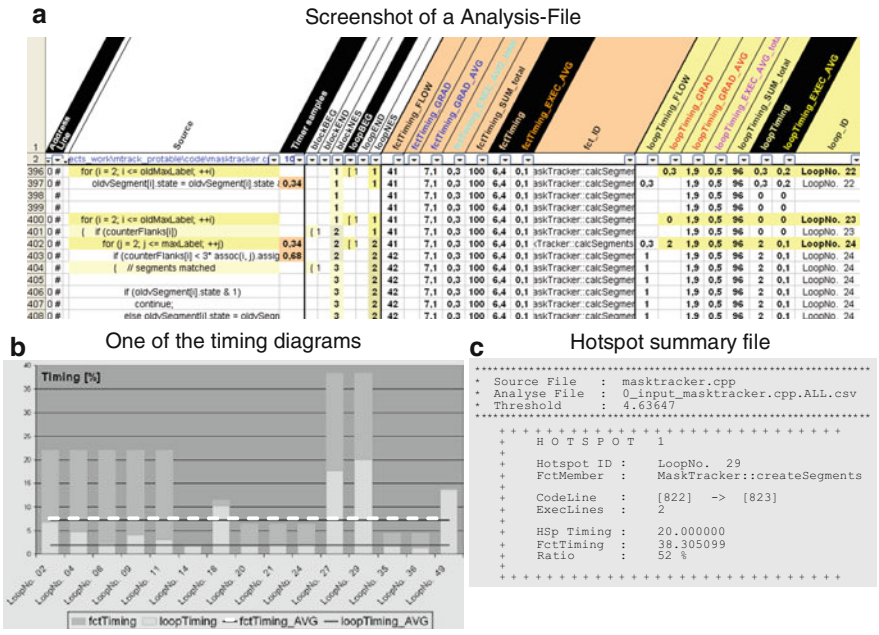


Fig. 6.13 Different screenshots and results of the Profile Analyzer tool

level, e.g., specific pragmas have to be inserted, etc. Of course, if the chosen accelerator is already available in an existing hardware IP library, a C-to-FPGA tool is not required and the existing IP could be used directly. After this the commercial tools of the FPGA vendor, e.g., Xilinx, are used to synthesize the hardware. On the software side, the inter-processor communication is handled automatically for MPI-based applications by including the RAMPSoC MPI implementation library (see [14] for details). If the chosen software application does not use MPI, the interprocessor communication has to be added manually, which can be done easily, due to available API-libraries for the different communication infrastructures. The software is then compiled using an off-the-shelf compiler such as GNU gcc. The software executables and the netlists of the hardware architecture are then given to the custom system integration tool called GenerateRCS [17], which provides the user with a graphical interface and eases the generation of the full and partial configuration bitstreams for the FPGA by calling the appropriate Xilinx tools.

Now the design methodology is fully executed and the system can be tested on the FPGA. If the achieved results are not sufficient, the user can return to phase 1 or 2 to stop, for example, at a different hierarchy in the hierarchical clustering or to select a different/additional hotspot, iteratively to fulfill his/her envisioned design goals. Another reason to go back to the previous phases would be if in phase 2 most of the code fragment is mapped to the processor's accelerators, which is then most of the time idle. So the designer could go back to phase 1 to initiate a different partitioning and to move parts of the tasks, which were originally mapped to other processors now to this processor. The result could be in this case a reduced number of utilized processor cores.

The modular structure of the design methodology makes it very flexible and independent from the target architecture as well as from the available commercial tools. It therefore does not matter to the overall design methodology, which commercial tool is used for profiling or for generating the hardware accelerators. The difference may only be in the manual effort, e.g., some C-to-FPGA tools have more restrictions on the input C, C++ language as others. Furthermore, each of the three phases is independent of the others and can be reused also for other target architectures. Phase 1 can be used to analyze the applications and to suggest a partitioning for other MPSoC architectures, while phase 2 can be reused to analyze applications for the generation of single processors with FPGA-based accelerators. Finally, the GenerateRCS tool can be used stand-alone to graphically display VHDL files and to generate partial and full bitstreams for Xilinx FPGAs by calling the Early Access Partial Reconfiguration Flow from Xilinx [22].

6.7 CAP-OS: Configuration Access Port-Operating System for RAMPSoC

To schedule the applications, to allocate the application tasks and to manage the hardware resources and the access to the single internal configuration access

port (ICAP), a special purpose operating system called Configuration Access Port-Operating System (CAP-OS) [15] was developed. CAP-OS has to assure that the different applications running on RAMPSoC meet their real-time constraints. At the same time, because RAMPSoC is an embedded system, CAP-OS needs to assure that the utilization of hardware resources and therefore the overall power consumption is kept low.

CAP-OS is running on one of the RAMPSoC processors, as can be seen in Fig. 6.8. In its current version, CAP-OS is running on top of the Xilkernel real-time operating system (RTOS).[2] Xilkernel is managing the resources on this single RAMPSoC processor, while CAP-OS is managing the resources for the complete RAMPSoC. CAP-OS is programmed as a multithreading application and therefore uses the multithreading capabilities of the Xilkernel RTOS. This processor will be called CAP-OS processor in the following.

Like the design methodology, CAP-OS also hides the complexity of the underlying RAMPSoC hardware from the user, as can be seen in Fig. 6.14.

The highest layer of abstraction is the Applications layer. Here, CAP-OS receives the output of the design methodology. This means the task graph description of the applications, the software executables and the partial configuration bitstreams. Furthermore, it can receive task requests from other RAMPSoC processors.

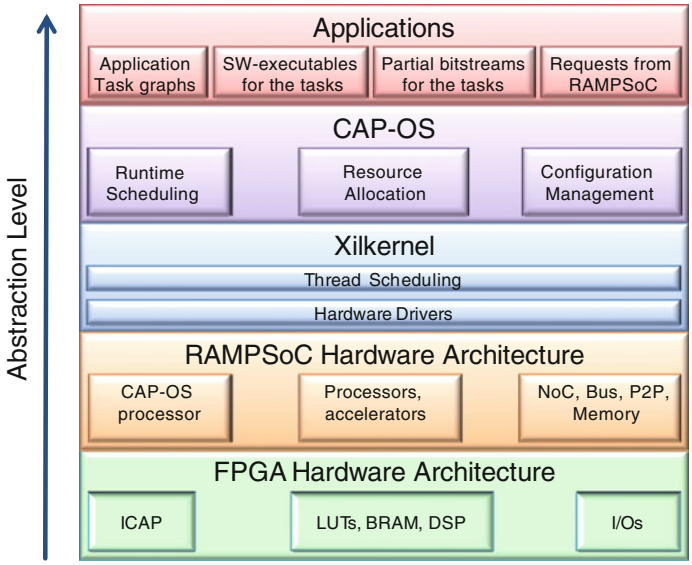


Fig. 6.14 Abstraction levels of the CAP-OS

[2] "Xilkernel v3_00_a"; EDK 9.1i, December 12, 2006. Available at http://www.xilinx.com.

The second level represents the three major tasks of CAP-OS:

- Runtime scheduling of the applications
- Allocation of the application tasks to resources by trying to reuse already existing resources
- Managing the configuration of the device and therefore the access to the ICAP interface

Below CAP-OS, the Xilkernel RTOS is running, which schedules the execution of the CAP-OS threads. Moreover, it provides the hardware drivers, so that the CAP-OS can access peripherals, such as external memory, the FSL-ICAP, the UART, and the communication interfaces to the other processors.

The fourth level is the RAMPSoC Hardware Architecture, which represents the CAP-OS processor, all other processors, accelerators, the communication infrastructure, and the memories of the system.

Finally, the lowest layer of abstraction is the FPGA Hardware Architecture, e.g., ICAP, LUTs (Look-Up Tables), BRAMs (Block Random Access Memories), DSPs and I/Os (Input/Outputs).

The user only sees the highest level of abstraction: the Applications level. All other layers are hidden from the user. The CAP-OS level is hidden by the design methodology, which configures the CAP-OS parameters and the hardware architecture. The CAP-OS itself then hides at runtime the lower three level from the user by automatically managing the hardware architecture.

For the runtime scheduling, CAP-OS uses a preemptive scheduling approach allowing the termination of a configuration. For the scheduling, a combination of a static list scheduling and a novel dynamic scheduling approach are used.

The static list scheduling algorithm has the benefits that it is a priority-based algorithm and that it considers resource constraints. Therefore, it is used to roughly assign priorities to the application tasks using the information of the task graph description, which has been received from the design methodology of RAMPSoC. For calculating the priority with the list scheduling algorithm, first the As Soon As Possible (ASAP) and the As Late As Possible (ALAP) start time for each task are calculated. By subtracting the ASAP start time from the ALAP start time, a mobility can be calculated for each task, which is used to assign a priority to each task, where tasks with a small mobility get a higher priority. For the scheduling resource constraints, e.g., single ICAP, maximal number of possible processing elements, as well as other constraints, e.g., reconfiguration time and communication costs of the tasks are considered.

Based on the results of the static list scheduling, a novel dynamic scheduling approach is used, which evaluates the current ready tasks. Ready tasks, are tasks, which either do not have any predecessors or whose predecessors have already been reconfigured. Here, a differentiation happens between tasks of an application task graph with hard real-time constraints and such with soft real-time constraints. The tasks with soft real-time constraints become a lower priority and will be reconfigured after the ones with hard real-time constraints, even if they had a higher priority based on the static list scheduling. Additionally, the novel scheduling algorithm tries to reuse

existing resources, if they are already reconfigured. This way, the time needed for reconfiguring the task can be saved. An additional feature of this novel scheduling approach is that the clock frequency of processing elements can be increased/decreased on demand at runtime to speed up the execution of the current task to free a processing element faster so that the next task can be mapped onto this processing element. This way, the processing element could be reused and no new processing element needs to be configured onto the device, which results in saving reconfiguration time and also in keeping the power consumption low. If on the other hand the task can finish easily, the clock rate could be decreased to reduce the dynamic power consumption and to achieve a good tradeoff between power consumption and performance. Hereby, it is assumed that the execution time stays in strong relation to the clock frequency. Another reason for increasing the clock frequency occurs, if a task otherwise cannot finish its execution before its ALAP time. Furthermore, the termination of a current reconfiguration is possible if a task with a higher priority urgently needs to be reconfigured.

The current version of CAP-OS is implemented using the following six threads:

1. Test_main: Initial thread that launches the following five threads
2. Init_proc: Generates a list with all possible processors and their attributes. This thread is only executed once at the startup.
3. Task_graph: Based on the information obtained by the design methodology, this thread initializes all tasks and generates the task graphs. It further calculates the start time and the mobility of each task using the ALAP and the ASAP algorithms. Finally, it analyzes the requirements of each task, e.g., specific hardware constraints or type of algorithm, and searches for tasks with similar requirements. Tasks with equal requirements are marked, because this allows reusing the existing resources by several tasks.
4. Schedule: This thread schedules the currently ready tasks. Ready tasks are tasks which predecessors have been already configured on the device. Furthermore, this thread also searches for available processing elements for these tasks.
5. Configure: This thread is responsible for managing the configuration of new or existing processing elements. Furthermore, it is responsible for transferring the software executables to existing processing elements, either via the ICAP or via the communication infrastructure. It also sends the new configured task the required information about the location of its predecessors and successors tasks.
6. Contr_Exit_Task: It controls the currently executing tasks and if one task finishes its execution it frees the corresponding processor element.

The last three threads: Schedule, Configure, and Contr_Exit_Task have the same priority and continue with their execution until the complete task graph is executed. They share the CAP-OS processor, while threads 1–3 are only executed once at the beginning. Currently, an extension of CAP-OS is under development, which supports the processing of requests from processing elements and also further runtime requests from the user to load additional applications at runtime. Requests from processing elements can be, for example, the addition or exchange of a hardware accelerator depending on the data to be processed.

6.8 Conclusions and Outlook

After introducing reconfigurable architectures and several reconfigurable VLIW and MPSoC architectures, the holistic approach of RAMPSoC was presented. RAMPSoC combines the benefits of MPSoCs and FPGAs, resulting in a very flexible hardware architecture, which achieves a good performance per Watt ratio, because it can adapt to the requirements of the application. The RAMPSoC hardware architecture supports the design-time and runtime adaptation of the type and number of processors, the communication infrastructure and the closely coupled hardware accelerators. To hide the complexity of the RAMPSoC hardware, a novel design methodology is provided that aids the user in partitioning his/her application and generating an appropriate RAMPSoC hardware architecture. The design methodology can be used without any knowledge of hardware description languages such as VHDL or Verilog. The configuration files and the software executables together with a task graph descriptions are also generated by the design methodology. These files are then passed on to CAP-OS, which is a special purpose operating system that hides the complexity of the runtime adaptation of RAMPSoC from the user. CAP-OS is responsible for scheduling the applications based on the task graph descriptions, for allocating tasks to processing elements, and for managing the hardware resources and for configuring the overall system using dynamic and partial reconfiguration.

Future steps will be the evaluation of the complete RAMPSoC approach with several high-performance computing applications from the image processing domain. Based on this evaluation, the hardware architecture, the design methodology, and CAP-OS will be improved to further ease the programming of the RAMPSoC architecture and to support a greater library of processing elements, communication infrastructures and hardware accelerators.

References

1. J. Becker, R. Hartenstein, Configware and Morphware going Mainstream; Elsevier Journal of Systems Architecture JSA (Special Issue on Reconfigurable Systems), October 2003
2. M. Berekovic, A. Kanstein, B. Mei, Mapping MPEG Video Decoders on the ADRES Reconfigurable Array Processor for Next Generation Multi-Mode Mobile Terminals; In Proc. of Global Signal Processing Conferences & Expos for the Industry: TV to Mobile (GSPX 2006), Amsterdam, Netherlands, March 29–30, 2006
3. P. Bertin, D. Roncin, J. Vuillemin, Introduction to Programmable Active Memories; Systolic Array Processor, Prentice Hall, pp. 300–309, 1989
4. C. Bobda, T. Haller, F. Mühlbauer, D. Rech, S. Jung, Design of Adaptive Multiprocessor on Chip Systems; In Proc. of the 20th Annual Conference on Integrated Circuits and Systems Design (SBCCI 2007), Copacabana, Rio de Janeiro, pp. 177–183, Sept. 3–6, 2007
5. L. Braun, D. Göhringer, T. Perschke, V. Schatz, M. Hübner, J. Becker, Adaptive real time image processing exploiting two dimensional reconfigurable architecture; Journal of Real-Time Image Processing, Springer, vol. 4, no. 2, pp.109–125, 2009

6. D. Burke, J. Wawrzynek, K. Asanovic, A. Krasnov, A. Schultz, G. Gibeling, P.-Y. Droz, RAMP Blue: Implementation of a Manycore 1008 Processor System; In Proc of RSSI 2008, July 2008
7. A. Cappelli, A. Lodi, C. Mucci, M. Toma, F. Campi, A Dataflow Control Unit for C-to-Configurable Pipelines Compilation Flow; In Proc. of IEEE 12th Int'l. Symposium on Field-Programmable Customs Computing Machines (FCCM 2004), Napa Valley, CA, USA, pp. 332–333, April 20–23, 2004
8. C. Chang, J. Wawrzynek, R. W. Broderson, BEE2: A High-End Reconfigurable Computing System; IEEE Design and Test of Computers, vol. 22, no. 2, pp. 114–125, 2005
9. C. Claus, W. Stechele, A. Herkersdorf, Autovision - A Run-time Reconfigurable MPSoC Architecture for future Driver Assistance Systems; Information Technology Journal, vol. 49, no. 3, pp. 181–187, June 20, 2007
10. K. Compton, S. Hauck, Reconfigurable Computing: A Survey of Systems and Software; ACM Computing Surveys, vol. 23, no. 2, pp. 171–210, 2002
11. G. Estrin, Organization of Computer Systems-The Fixed Plus Variable Sructure Computer; In Proc. of Western Joint Computer Conference, pp. 33–40, 1960
12. D. Göhringer, J. Becker, High Performance Reconfigurable Multi-Processor-Based Computing on FPGAs; In Proc. of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, USA, April, 2010
13. D. Göhringer, M. Hübner, M. Benz, J. Becker, A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip; In Proc. of the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010), Charlotte, USA, May, 2010
14. D. Göhringer, M. Hübner, L. Hugot-Derville, J. Becker, Message Passing Interface Support for the Runtime Adaptive Multi-Processor System-on-Chip RAMPSoC; In Proc. of the 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS X), Samos, Greece, July 2010
15. D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, J. Becker, CAP-OS: Operating System for Runtime Scheduling, Task Mapping and Resource Management on Reconfigurable Multiprocessor Architectures; In Proc. of Reconfigurable Architectures Workshop (RAW 2010), Atlanta, USA, April, 2010
16. D. Göhringer, B. Liu, M. Hübner, J. Becker, Star-Wheels Network-on-Chip featuring a self-adaptive mixed topology and a synergy of a circuit- and a packet-switching communication protocol; In Proc. of the International Conference on Field Programmable Logic and Applications (FPL2009), Praha, Czech Republic, August/September, 2009
17. D. Göhringer, J. Luhmann, J. Becker, GenerateRCS: A High-Level Design Tool for Generating Reconfigurable Computing Systems, In Proc. of the IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2009), Florianopolis, Brazil, October, 2009
18. R. Hartenstein, A Decade of Reconfigurable Computing: A Visionary Retrospective; In Proc. of Design, Automation and Test in Europe (DATE 2001), Munich, Germany, pp.642–649, March 12–16, 2001
19. R. Hartenstein, Why We Need Reconfigurable Computing Education; RC-Education Workshop, Karlsruhe, Germany, 2006
20 S. Hauck, A. DeHon, Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation; Morgan Kaufmann Series in Systems on Silicon, 2007
21. J. Howard, S. Dighe, Y. Hoskote et al., A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS; In Proc. of IEEE International Solid-State Circuits Conference (ISSCC 2010), San Francisco, CA, USA, Feb. 2010
22. P. Lysaght, B. Blodget, J. Mason, J. Young, B. Bridgford, Invited paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs; In Proc. of the International Conference on Field Programmable Logic and Applications (FPL 2006), Madrid, Spain, pp. 1–6, August 2006
23. nVIDIA® Tesla™, GPU Computing Technical Brief, Version 1.0.0, May 2007

24. K. Paulsson, M. Hübner, H. Zou, J. Becker, Realization of Real-Time Control Flow Oriented Automotive Applications on a Soft-core Multiprocessor System based on Xilinx Virtex-II FPGAs; In Proc. of International Workshop on Applied Reconfigurable Computing (ARC 2005), Algarve, Portugal, pp. 103–110, Feb. 22–23, 2005
25. B. Radunovic, An Overview of Advances in Reconfigurable Computing Systems; In Proc. of the 32nd Hawaii International Conference on System Science, January 1999
26. N. Voros, A. Rosti, M. Hübner, Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach; Springer, Netherlands, 2009
27. A. Thomas, J. Becker, Dynamic Adaptive Routing Techniques In Multigrain Dynamic Reconfigurable Hardware Architectures; In Proc. of the International Conference on Field Programmable Logic and Applications (FPL 2004), Antwerp, August 2004
28. M. Ullmann, B. Grimm, M. Huebner, J. Becker, An FPGA Run-Time System for Dynamical On-Demand Reconfiguration; In Proc. of Reconfigurable Architectures Workshop (RAW 2004), Santa Fé, USA, 2004
29. W. Wolf, The Future of Multiprocessor Systems-on-Chips; In Proceedings of the Design Automation Conference (DAC 2004), San Diego, USA, pp. 681–685, June 7–11, 2004

Part III
"Physical Design of Multiprocessor Systems"

Chapter 7
Design Tools and Methods for Chip Physical Design

Ricardo Reis

Abstract This chapter presents a new approach for the physical design of integrated systems, as the MPSoCs, where all logic cells are designed on the fly, without the limitations that a physical designer faces when using a cell library (number of functions, number of transistors, transistor sizing, area, and power consumption). Many functional blocks compose MPSoCs and several of them are composed by random logic. So, it is important to optimize these random logic blocks by using only the needed transistors and using the right sizing. Then there is demand of EDA to cope with this goal. A basic tool to minimize the number of transistors is the one that provides degeneration of any logical function using the optimal number of transistors. A cell generator allows the automatic design of cells composed by any transistor network (using simple gates or static CMOS complex gates – SCCG) and having any transistor sizing. When the size of the transistor should be bigger than the cell height, the tool is able to do transistor folding. As the designer is free from the limitations of a cell library, it is possible to do a deep logic minimization where all needed logic cells will be generated on the fly. This allows a reduction in the number of needed transistors to implement a circuit. As a consequence, the static power consumption will also be reduced. The cell generator provides cells with a compacted layout, allowing a significant transistor density. It is presented some physical design automation strategies related to transistor topologies, management of routing in all layers, VCC and Ground distribution, clock distribution, contacts and vias management, body ties management, transistor sizing and folding, and the how these strategies can improve the layout optimization. Some results are compared with the ones obtained with traditional standard cells tools (vendor's tools), showing the gain in area, delay, and power consumption. The flexibility of the approach can also let the designers to define the layout parameters to cope with problems such as tolerance to transient effects, yield improvement, printability, etc. The designer can also manage the sizing of transistors to reduce power consumption, without compromising the clock frequency.

R. Reis (✉)
Instituto de Informática – Universidade Federal do Rio Grande do Sul,
Porto Alegre, Brazil
e-mail: reis@inf.ufrgs.br

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_7,

Keywords Physical Design · Low Power · Transistor Networks · Layout · EDA

7.1 Introduction

The research and development of the design automation of integrated circuits started at the layout level and evolved to higher levels of abstraction. The physical design of integrated circuits was done by hand till the end of years 1970s. The computer was used as a tool for drawing the layout, but there was no tool for the physical design automation of ICs. As the number of components was increasing, it appeared a strong need for physical design automation. The first solution was the use of some regular blocks that were more viable to automate their design. The Motorola 68000 was an example of the starting of using regular blocks such as ROMs and PLAs [1]. It is possible to see that the control part of the original 68000 uses two levels of ROMs and many PLAs. To improve performance and area, in years 90, many circuits such as the Intel 486 start to use the standard cell (SC) approach, mainly in their control part. The traditional SC approach is used till today, and it is still okay for many designs. So, if the SC approach is okay in many cases, do we need to search for a new approach? Yes, if we want to optimize the physical design for power reduction, area reduction and performance increase. Is the SC approach an automated one? It is a partial automated approach, because at the same time that placement and routing are automated steps, the design of each cell is not. The cells are already designed and taken from a cell library. As the design of a cell library represents an important cost, the number of different functions that we can find in a typical cell library is limited (in general we can find something like 150 different logic functions) and the sizing of the cells is also limited. In general, we can find about three different sizing for each function, one for power, one for performance, and one for area. These limitations do not allow reaching an optimization of the circuit at the physical design level. One of the big advantages to use SC in the past was that the cell characterization was sufficient to estimate the delay. This is anymore the case if we consider recent technologies, where the delay is mainly due to the connections. So, we should find a way to reduce the wire lengths, as the connections are the central problem to reduce the delay of a circuit. We can claim that the SC approach is far from minimization of power consumption, number of transistors, delay, wirelength and area. So, if we want to do a physical design optimization we need to do a change on the design paradigm and to search for new physical design approaches. We are proposing a new approach where the cells are designed on the fly, during the physical design step, considering fan-in and fan-out, and an optimization of the number of transistors. This new approach also means a change in the level of abstraction of the physical design step, because it is not anymore just a placement and routing of logic cells, but a placement and routing of a network of transistors.

7.2 Use of MOS Complex Gates

One way to reduce the amount of transistors is to use static CMOS complex gates (SCCGs), where functions with several inputs can be implemented using only one gate. In Fig. 7.1, it is shown an example where a same function can be implemented using basic gates or using just a complex gate (SCCG). It is clear in the example that the option using complex gates uses a minimum number of transistors.

Table 7.1 [2] shows the number of functions that are possible to realize using a maximum number of stacked P and stacked N transistors. It is possible to see, for example, that if it is chosen a limit of maximum four stacked P and four stacked N transistors, the possible number of logical functions is 3,503. So, this number of functions is much higher than the number that we can find in a typical cell library. As it is not feasible to implement cell libraries with all possible functions, a solution is to use an approach where the cells are designed on the fly, during the physical design.

An automatic cell generation approach gives much more freedom for the logical minimization step, because there is no more need to bias the logical minimization by the functions available in the cell library. Any function provided at the logical

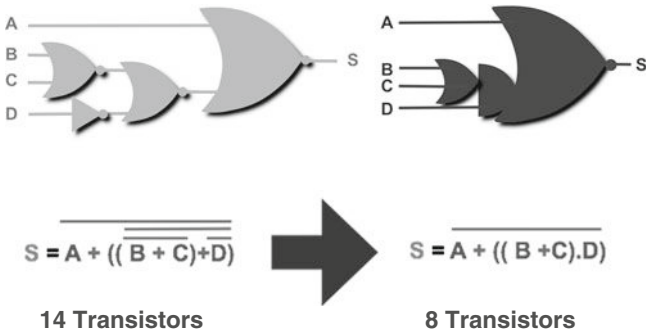


Fig. 7.1 Two different options for the design of a same function

Table 7.1 Number of possible different functions using a limited number of stacked P and stacked N transistors

		NUMBER OF STACKED PMOS TRANSISTORS				
		1	2	3	4	5
NUMBER OF STACKED NMOS TRANSISTORS	1	1	2	3	4	5
	2	2	7	18	42	90
	3	3	18	87	396	1677
	4	4	42	396	3503	28435
	5	5	90	1677	28435	125803

synthesis step can be implemented during the layout synthesis. It is also important to remark that when doing the generation of cells on the fly, we can also generate transistors with any size, by using folding of transistors.

7.3 Wirelength Reduction

One challenge in modern integrated circuit design is to reduce the wire length. The possibility to use any logic function gives the opportunity to reduce significantly the number of transistors. As the reduction in the number of transistors will reduce the number of wires, this also will reduce the wire length, contributing to reduce the delay of a circuit. The wire length is also reduced by the area reduction of the circuit due to the reduction in the number of transistors. An important point is that there is still a space to improve the placement and routing algorithms. This claim is presented in [3], where the authors showed that placement algorithms are 1.43–2.38 times the optimal solutions considering wire length.

7.4 Power Reduction

The reduction in the number of transistors is becoming more and more important in recent technologies. The main reason for that is the static power consumption. Leakage current is increasing more and more as the features of the transistors are being reduced. As a consequence, the static power is in some cases bigger than the dynamic power. This static power consumption is due to the leakage current that is significant in recent technologies. So, a way to reduce the static power consumption is to reduce the amount of transistors, as it is also proportional to the amount of transistors.

7.5 Layout Strategies

The layout of a cell depends on decisions related to the different issues present in the design of a cell layout, such as:

- Transistor topologies,
- VCC and Ground distribution,
- Clock distribution,
- Management of contacts and vias,
- Management of routing layers,
- Body ties management, and
- Transistor sizing.

The decision taken in each issue has effects on the other ones. For example, a decision related to the VCC and Ground distribution defined some restrictions to the placement of some vias. If it is used too many parameters to do each decision, the algorithms will be too complex and the runtime will be prohibitive. Basic possible transistor topologies are, for example, horizontal, vertical, transistor with doglegs, transistor folding. But if we consider doglegs, they could be done in many different locations, but depending on the position of contacts and vias. If we consider folding, the number of transistor segments can vary a lot, especially when they are drives.

Figure 7.2 shows a folding of a transistor in two segments. So, the use of transistor folding also increases the layout choices to be done when we have to do the design of a logic cell. The routing management has to define the priority for routing in each layer, the priority for routing in each track of a strip (a strip is the region between two power lines, VCC and ground), and the routing directions in each layer. The management of routing priorities in a strip can avoid the need to route a signal using several tracks.

The VCC and ground can be implemented in the borders of a strip, in the middle of a strip (between P and N planes), or over the transistors. Figure 7.3 shows a layout where the power lines run over the transistors.

The contact and vias management is becoming more and more important, as the tower of vias crossing several layers can impose important restrictions to the routing step. In [4], it is described an approach where the vias are placed in one or two tracks

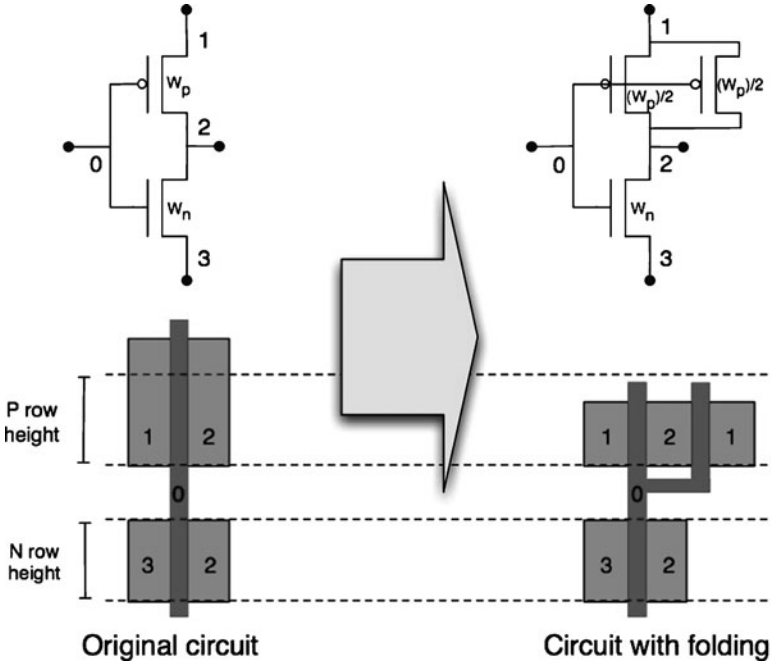


Fig. 7.2 Transistor folding in two segments

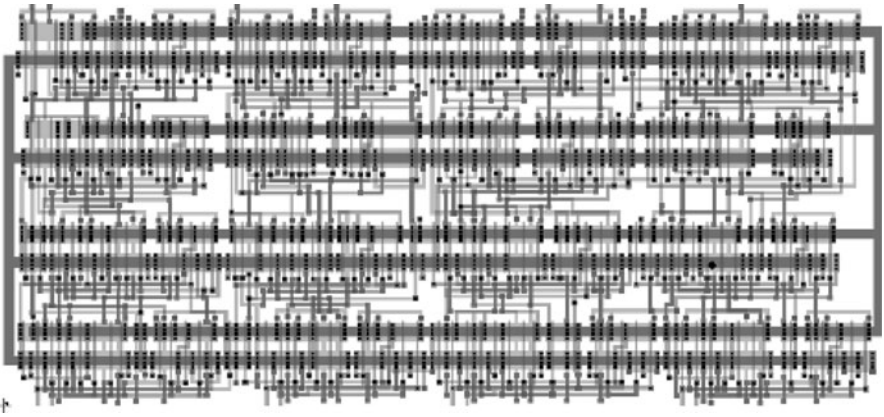


Fig. 7.3 Layout with power lines over the transistors

between P and N planes. Then the routing in the upper layer can be done using a channel router approach. As a consequence, the routing step will be speeded up.

So, the challenge is to develop algorithms that can provide quality results but with a short runtime.

7.6 Layout as a Network of Transistors

In the new proposed approach, a circuit is viewed as a network of transistors. The tools are designed to generate automatically any network of transistors, even with a different number of P and N transistors. In this approach, there is a change in the level of abstraction of the layout synthesis in relation to the traditional SC approach. In place of the synthesis of a layout using a set of predesigned cells, it is done a placement and routing of transistors. The transistors can have any size. If the W of the transistor is bigger than the height of the respective diffusion region, it is done a transistor folding. This is very useful when doing the design of a driver. Figure 7.4 shows the layout of a circuit done using the Parrot tool set. The circuit layout was generated automatically and all the transistors were completely designed on the fly.

Figure 7.5 shows the evolution of the approach showing the results for an ISCAS C1355 benchmark using Tropic tool set [5] and Parrot tool set [6]. It is possible to see a nice evolution in density and an important reduction in the delay. Both tools have the power to generate in one shot the full layout a functional block with thousands of gates (the number of gates depends on the machine where the tool is running).

One alternate approach we are working on is to do first the automatic generation of the cells that a specific circuit will need, using the right sizing. This approach is used in a new tool named automatic synthesis of transistor networks (ASTRAN) that is able to generate the layout of cells with a set of tens of transistors. The first results were published in [7].

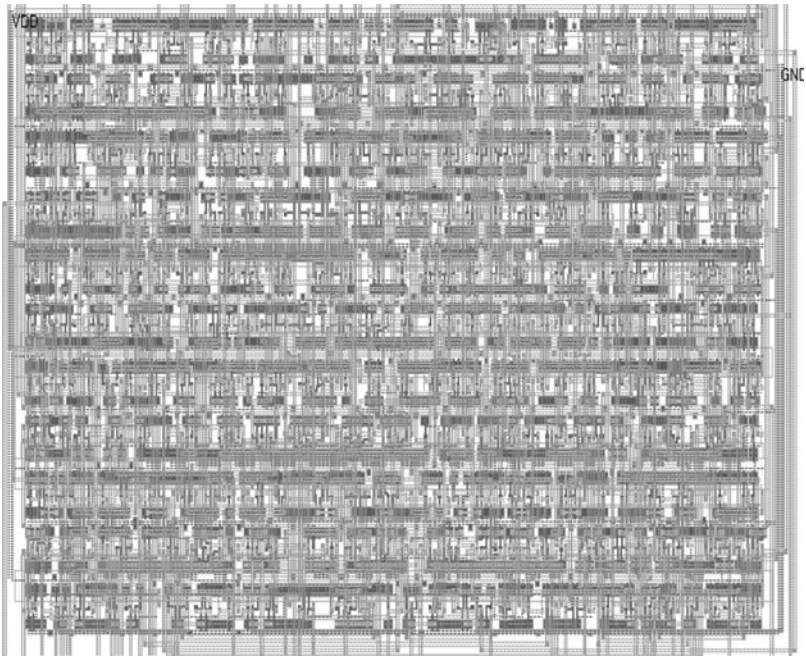


Fig. 7.4 Layout generated with the Parrot Suite

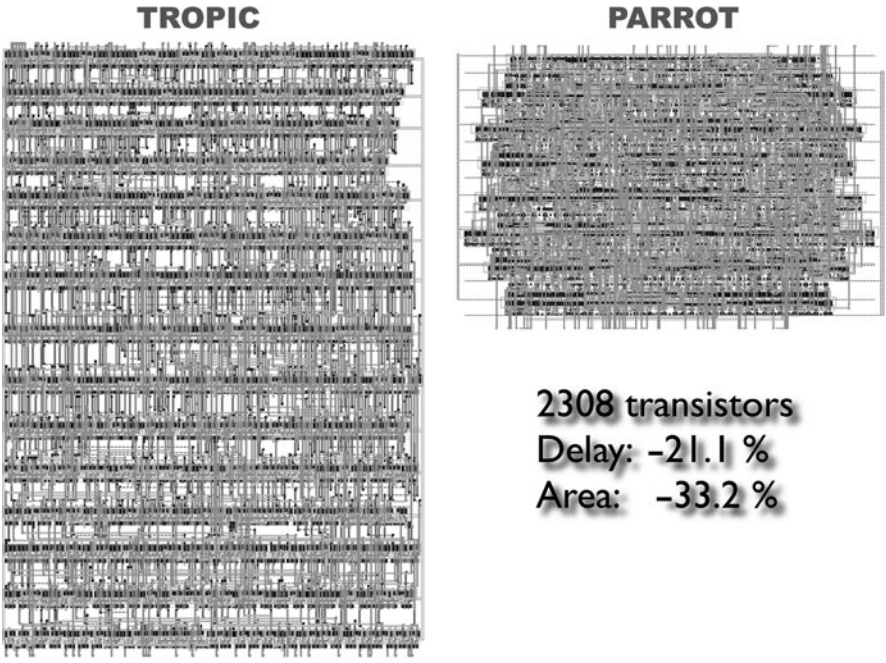


Fig. 7.5 Comparison between layouts generated with Tropic and Parrot tools for an ISCAS C1355 benchmark

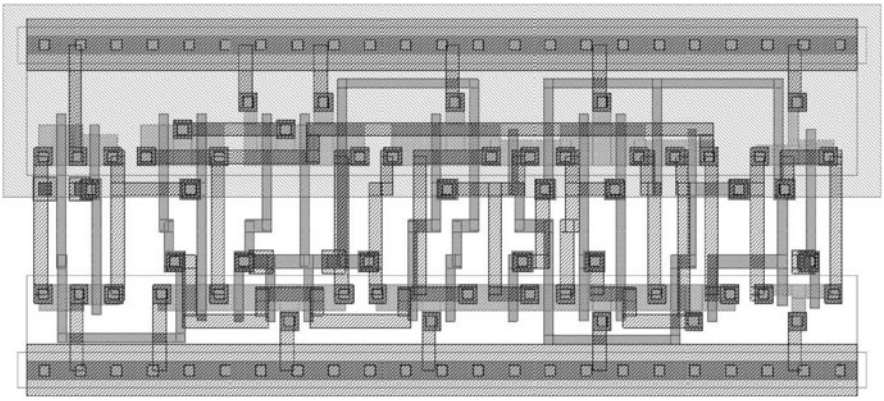


Fig. 7.6 A JK FF using 34 transistors generated automatically

Figure 7.6 shows the layout of a JK FF with 34 transistors generated automatically. It is possible to see that the tool can manage to route polysilicon and metal using doglegs. It is also possible to see that it can generate transistors with different sizes.

Some other cells generated with ASTRAN are presented in Fig 7.7. It is possible to verify that the cell density is quite good. The density is mainly improved because the tool is able to provide automatically an internal routing using polysilicon and metal wires with doglegs.

Figure 7.8 shows the layout of a 4×4 multiplier generated using two approaches. The first one was generated using a traditional SC approach provided by a vendor tool set. The second one was generated using ASTRAN and a Cell Assembly Tool using a Data Path approach. It is evident that the second solution presents a smaller layout. Observing Table 7.2 that show the data of both implementations, it is clear that the second solution presents an important reduction in the number of transistors (634 against 376) due mainly to the use of complex gates. The second solution using ASTRAN presents also a nice reduction on delay and an important reduction on power consumption (almost 40% power reduction).

As ASTRAN tool can support transistor folding, transistor sizing, and other layout parameters, it can be used to experiment different layout solutions that can cope with many issues, such as tolerance to radiation effects and variability. Here, there is a lot of experiments and research to explore these possibilities.

7.7 Using ASTRAN to Help in the Synthesis of Analog Modules

The MPSoCs can also include analog modules. The ASTRAN tool can also help in the synthesis of some kinds of analog modules. There is an interesting space to work in a version of ASTRAN, dedicate to generate some kinds of analog circuits.

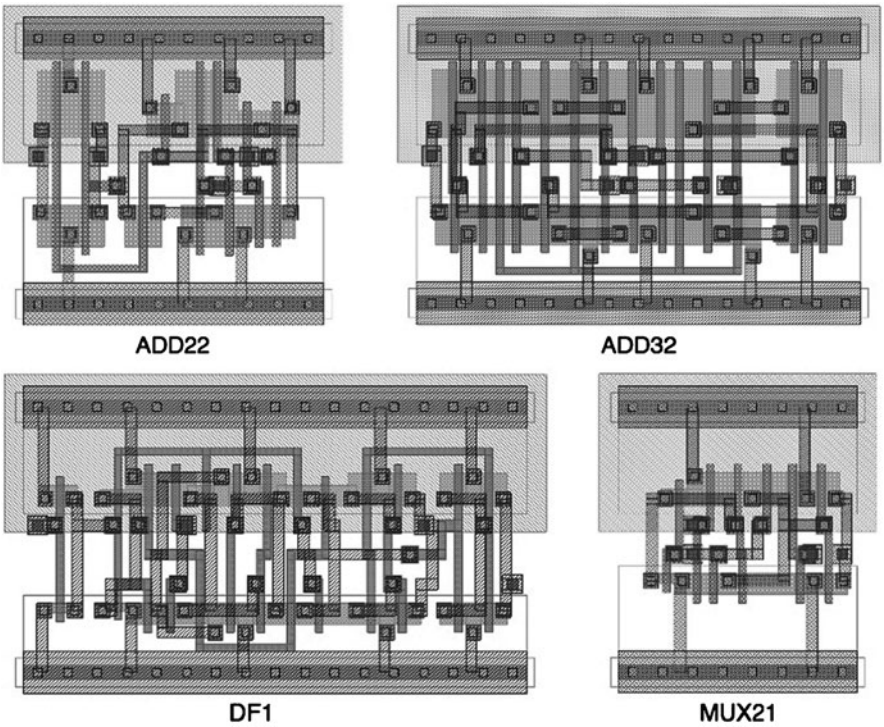


Fig. 7.7 A set of cells generated with ASTRAN

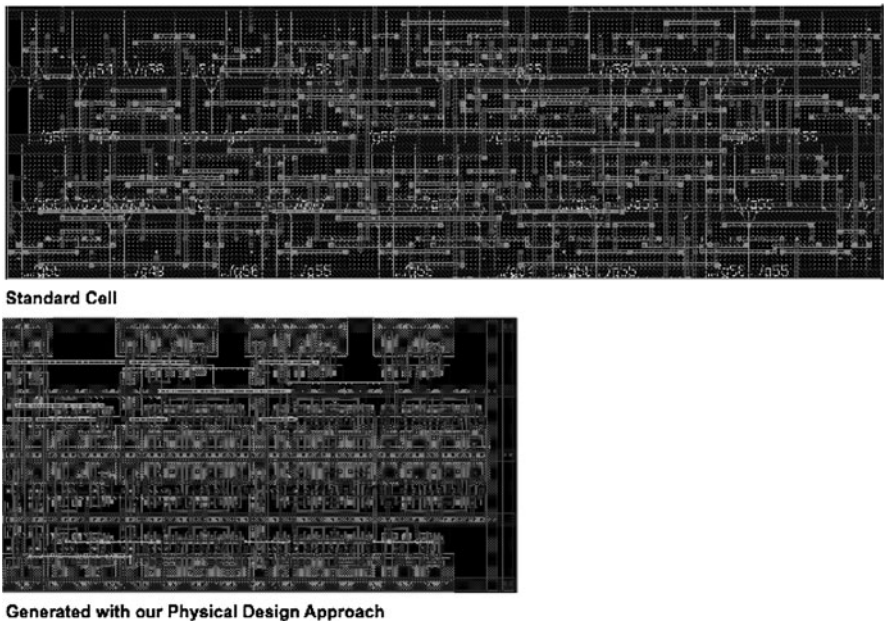


Fig. 7.8 A 4×4 multiplier generated using ASTRAN and a cell assembly tool

Table 7.2 Comparison between results using a standard cell approach and ASTRAN, our automatic layout tool

	Standard cell	Cell compiler	Gain (%)
Number of cells	52	28	46
Number of transistors	634	376	59.3
Area (μm^2)	6,716	5,070	24.50
Delay (ps)	2,174	1,896	12.8
Power (mW)	6.45	3.97	61.55

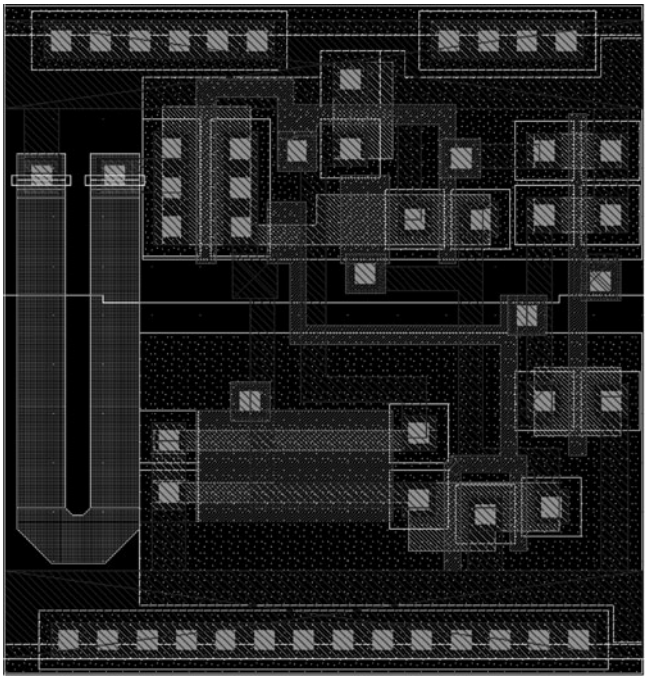


Fig. 7.9 Current generator layout using a 65 nm technology

Figure 7.9 [8] shows an example of an analog circuit generated with ASTRAN that is a current generator using a 350 nm technology. Another example is presented in Fig. 7.10 [9, 10], which shows the layout of an aging sensor using an industrial 65 nm technology and mixing analog and digital components.

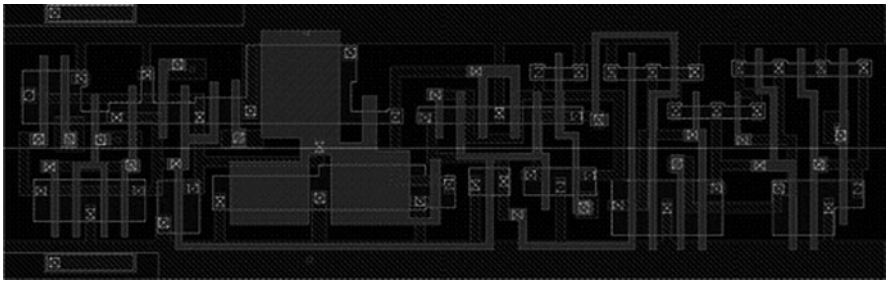


Fig. 7.10 Aging sensor layout using a 65 nm technology

7.8 Conclusions

It was presented a new approach for the physical synthesis of integrated circuits willing to optimize area, power consumption, and performance. The tool can also be used to generate modules of an MPSoC. The methodology is based on the synthesis of a transistor network where the layout is designed automatically on the fly during the design of the circuit. The possibility of implementing any logic function provides a reduction in the number of transistors. As a consequence, there is a reduction in area, wire length, delay, and power consumption. As the cells are designed on the fly and the transistors can have any size, it lets the designer to explore several layout strategies to improve delay, power consumption, tolerance to radiation effects, and variability.

Acknowledgments We thank all the students and colleagues who worked in the Physical Design Project and implemented several tools which some of the results are reported in this chapter. We also thank CNPq and CAPES that sponsored some of our works and some of our students.

References

1. Reis, R., Lubaszewski, M., Jess, J., Design of Systems on a Chip: Design and Test, Springer, p. 297, 2007. ISBN 978-0-387-32499-2
2. Detjens, E. et al., Technology Mapping in MIS, In: IEEE ICCAD, 1987. Proceedings, Los Alamitos, California: IEEE Computer Society Press, pp. 116–119, 1987
3. Chang, C.-C., Cong, J., Xie, M., Optimality and Scalability of Existing Placement Algorithms. ASPDAC, 2003
4. Santos, G., Johann, M., Reis, R., Channel Based Routing in Channel-less Circuits. IEEE International Symposium on Circuits and Systems. ISCAS2006, Kos, Greece, May 21–24 2006, IEEE Press. ISBN:0-7803-9389-9
5. Moraes, F., Reis, A., Robert, M., Auvergne, D., Reis R. Towards Optimal Use of CMOS Complex Gates in Automatic Layout Synthesis. In: Proccedings of the 10th Congress of the Brazilian Microelectronics Society, Canela, July 31–August 4, Canela, SBMicro, pp. 11–20, 1995

6. Lazzari, C., Santos, C., Reis, R., A New Transistor-Level Layout Generation Strategy for Static CMOS Circuits, 13th IEEE International Conference on Electronics, Circuits and Systems – ICECS2006, Nice, France, December 10–13, pp. 660–663, 2006. ISBN: 1-4244-0395-2
7. Ziesemer, A., Lazzari, C., Reis, R., Transistor Level Automatic Layout Generator for non-Complementary CMOS Cells, In: IFIP/CEDA VLSI-SoC2007, International Conference on Very Large Scale Integration, Atlanta, USA, pp. 116–121, October 15–17 2007. ISBN: 978-1-4244-1710-0
8. Vazquez, J., Champac, V., Ziesemer A., Reis, R., Teixeira, I., Santos, M., Teixeira, J., Built-In Aging Monitoring for Safety-Critical Applications, IEEE International On-Line Test Symposium, IOLTS 2009, Sesimbra, Portugal, pp. 29–34, June 24–27 2009. ISBN 978-1-4244-4822-7
9. Vazquez, J., Champac, V., Ziesemer A., Reis, R., Teixeira, I., Santos, M., Teixeira, J., Delay Sensing for Parametric Variations and Defects Monitoring in Safety-Critical Applications. In: First IEEE Latin American Symposium on Circuits and Systems – LASCAS 2010, Iguaçu Falls, February 24–26 2010
10. Vazquez, J., Champac, V., Ziesemer A., Reis, R., Teixeira, I., Santos, M., Teixeira, J., Low-sensitivity to Process Variations Aging Sensor for Automotive Safety-Critical Applications, IEEE VLSI Test Symposium, VTS 2010, Santa Cruz, USA, June 18–21 2010

Chapter 8
Power-Aware Multicore SoC and NoC Design

Miltos D. Grammatikakis, George Kornaros, and Marcello Coppola

Abstract This chapter examines system-level design of power-efficient systems-on-chip. It starts by examining the sources of power consumption, considering high-level techniques for power-efficient processing, storage and on-chip communication. It also discusses algorithmic- and architecture-driven software transformations and application embedding for power-efficient embedded software. Then, it provides a glimpse at research and development of computer-aided design tools for effective multicore SoC power estimation, analysis and optimization at different abstraction levels and especially system-level modeling, including efforts towards standardization of power formats to enable tool interoperability. Finally, it considers state-of-the-art runtime power management and optimization techniques, including dynamic voltage scaling (DVS), frequency scaling (DFS) and other NoC-based power saving mechanisms. This chapter concludes by briefly outlining future trends towards true system-level power-aware design, providing a large list of references for further study.

8.1 Introduction

As CMOS technology is continuously scaling, single chip systems integrating a large number of processors, on-chip memories, and custom intellectual property (IP) cores have become a reality [1]. Most major chip manufacturers have already delivered or announced plans for large-scale chip multiprocessors (CMPs). Multi-threaded workloads that execute on such processors experience high on-chip communication latencies and dissipate significant power; hence, the need for adaptive chip architectures and corresponding design techniques that can dynamically accommodate the full range of workloads, from intensive computation- to memory- or communication-bound applications.

M. Coppola (✉)
ST Microelectronics, J. Horowitz 12, 38019 Grenoble, France
e-mail: marcello.coppola@st.com

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_8,

Low-power design techniques have been employed for more than 25 years, particularly in mobile consumer devices, such as watch circuits, cellular phone, gaming and calculators. Nowadays, small time-to-market, low cost, high performance, and low energy are mandated by the increasingly complex software applications, e.g. new power-efficiency MIPS/mW (or similar) metrics start to appear in communication and control systems in everyday life or portable battery-powered consumer electronic appliances. Major applications targeted by this new power-aware design methodology include not only mobile multimedia terminals and digital cellular telephony, but also next generation networks, laptop, set-top-box/digital TV design, and devices for emerging applications, e.g. ambient intelligence and sensor networks in surveillance systems.

The immense interest in today's energy-efficient microelectronics systems stems from increase of clock frequency, design complexity (in number of transistors), and leakage power in SoCs for deep submicron (DSM) technology (below 65nm), environmental laws, ethical reasons, energy delivery cost for stationary systems, extended battery life for portable systems with data intensive applications and thermal heat dissipation management to avoid heating that could make chip operation unfeasible or impractical [2].

Power profiling is needed to identify opportunities for optimizing power-critical parts for different applications, e.g., through built-in device-specific power management based on different operating voltages and reduced-power states that may be offered by the technology. For power profiling, analysis, estimation, and optimization of entire SoCs, five different abstraction levels have been considered (from the most abstract down to the most concrete level): application software (including software design, high level synthesis, and compilation), transaction, behavioral, gate (RTL synthesis), and transistor (or layout).

The choice of an abstraction level affects power estimation accuracy, execution speed of the model, and the time required to develop it. Until recently, RTL has been considered as the entry point in the design flow. For gate and transistor levels, many EDA vendors provide accurate probabilistic or simulation-based power estimation and optimization tools that often take prohibitively large time. Although modeling at this level is accurate, the scope for design exploration is reduced and significant changes are very costly to implement. In contrast, high-level power estimation based on component models with power state abstractions, analytical equations or lookup tables is at its infancy; it is difficult and inaccurate, since design characteristics and typical switching activity and application test-benches of hardware resources are unavailable, unknown, not well documented or partially known during the early design phases.

Nevertheless, over the last few years, high-level power estimation has been gaining momentum in the industry, as the new entry point in the design flow. System-level power estimation methodology, e.g., based on a bit- and cycle-accurate SystemC2.0 + transaction- or behavioral-level models separating communication and computation functionality, can be orders of magnitude faster than RTL simulation, yet sufficiently (near RTL) accurate to achieve considerable power savings during design space exploration, HW-SW partitioning, and component selection.

During early phases of design space exploration, absolute and accurate power estimation is not as important as relative accuracy, since it is enough that qualitative power metrics correlate with the final implementation ensuring that early design decisions are appropriate.

System power is computed by summing the power consumed by all resources, interactions, and the environment. Consumed power can be classified into four components: short-circuit, dynamic, static and leakage.

Short circuit power dissipation is caused by a short-circuit current that flows directly to ground during the nonzero interval (idle time) between fall and rise time in CMOS circuits when both the p- and n-device of a gate are conducting, i.e. during switching activity [3]. This power is wasted and never collected by output capacitances.

Dynamic power can be partitioned into internal consumption by the cell and energy for driving the load, including wiring and fan-out capacitance. Dynamic power consumption of CMOS circuits can be described as: $P=0.5\alpha fCV^2$, where f is the clock frequency, α is the switching activity that refers to bit transitions in gates per clock cycle, i.e. charging and discharging of output capacitance, C is the total load capacitance which is roughly proportional to the chip area, and V is the supply voltage level. The total capacitance is the sum of the input capacitance of fan-out gates, wiring capacitance and parasitic capacitance. While f and V are directly defined by the designer, C is determined by the system architecture, and α depends on data representation, application, mapping and architecture.

Static power is proportional to switching activity and several technology-related parameters. Since its contribution is usually below 20%, it is either ignored at the higher levels or captured as part of dynamic power estimation.

With traditional technologies, approximately 80–90% of all dissipated power in a circuit is due to switching activity and 90% of this is due to dynamic power. However, for deep submicron (below 65nm), wiring capacitance is the dominant component, but difficult to estimate and control at system-level. The standby power consumption due to subthreshold leakage current can be represented in the form: $P_{leak}=V_{dd}\ I_{off}\ K$, where I_{off} is the current that flows between the supply rails in the absence of switching and K is a factor that accounts for the distribution/sizing of P and N devices, the stacking effect, the idleness of the device, and the design style used.

Dynamic power management (DPM) consists of various runtime techniques employed to achieve energy-efficient processing at requested Q_oS constraints by minimizing the number of active system components, and therefore, total power consumption. When the workload of a power-manageable component (e.g. processor, memory or peripheral) is low, certain circuits can be turned off. In addition, a number of effective techniques are available for minimizing dynamic power consumption, by reducing f, α, C or V at transistor-, gate-, RT- or system-level:

- For example, we can influence dynamic power using technology-driven low power design, e.g., by employing architecture-driven scale reduction in ultra-low power CMOS technologies which also affects short-circuit power, reducing

load capacitance through silicon on insulator junctions and low-k dielectrics constants and routing high-frequency signals on the least capacitive upper layers.

- Similarly, at architecture-level, we can increase data parallelism or employ globally asynchronous locally synchronous channel design (GALS) which allows independent clocking and reduce clock frequency and/or the supply voltage level to provide a promising quadratic reduction on consumption [4]. Reducing the clock frequency, shortening the logical depth, or adding pipeline registers also reduces power due to glitches, i.e. delay-dependent transitions at gates due to signal delay imbalances in combinational logic (before the correct logical value becomes stable); this is very effective for data-path components, such as multipliers and parity trees. Glitches cause spurious, resulting in dynamic power dissipation. However, reducing frequency extends program execution time. Since energy consumption is a product of execution latency and power consumption, scaling down frequency saves dynamic power, but is ineffective in providing energy savings [5]; moreover, due to long delays, connected peripheral SoC devices, e.g. display, may consume more energy.
- Different design methodologies are available for reducing dynamic power at system/application software, mostly by decreasing switching activity. For example, in order to reduce overall signal switching activity, we can invoke improved routing techniques and smart data representations or modify our data or resource allocation scheme or the scheduling algorithm to minimize the number of basic data flow operations or increase correlation between successive input patterns of functional blocks. We can even change our programming paradigm, e.g. the object-oriented paradigm is known to introduce a significant performance and power penalty due to increased instruction count, larger code size and increased number of memory accesses [6]. Latency hiding techniques, e.g. using cache to explore data locality, multithreading or prefetching are also helpful in reducing capacitance by minimizing global communication over long wires with high capacitance load [7].

Static power management applies at design time. Static power as a fraction of total power increases as clock frequency drops. Static power can be controlled using a number of techniques:

- Scaling down transistor size through lowering the threshold voltage affects leakage power;
- Different choices of gate oxide thickness affect performance and change the balance between dynamic vs. static power consumption.
- Power gating (also called multi-threshold CMOS design) scales down the supply and scales up the threshold voltage to provide different power vs. performance tradeoffs for each macro-level block or standard cell. Since scaling down threshold voltage, exponentially increases subthreshold leakage, so called sleep transistors with high threshold voltage are inserted to function units or gates. Sleep transistors are turned off during the sleep mode, which can significantly reduce the leakage.

Next, we examine power-efficient design methodologies, including advanced application-driven adaptive strategies for energy-aware computing. We concentrate more on system-level design techniques for computation, storage, and communication, while only briefly considering energy-efficient gate-level, logic and physical chip design [8] Notice that system-level power estimation models use low-level simulation data on instruction execution, e.g. transistor state-transitions or technology data from an RT-synthesizable HDL for soft cores, netlist for firm cores and layout for hard cores to better estimate system metrics.

More specifically, in Section 2, we address the sources of power consumption, i.e. computation, communication and storage. We consider power estimation models, considering processor, memory and on-chip interconnect components, including system-level approaches towards NoC power estimation. We also discuss algorithmic- and architecture-driven software transformations and application embedding for power-efficient embedded software, and provide a glimpse at research efforts towards computer-aided design tools developed for effective multicore SoC power estimation, analysis and optimization at different abstraction levels from modeling to implementation, focusing on system-level. Finally, we identify current standardization efforts on power formats that enable tool interoperability.

Then, in Section 3, we explain state-of-the-art runtime power management and optimization techniques, including dynamic voltage scaling (DVS) and frequency scaling (DFS) and other Network-on-Chip-based power saving mechanisms. Finally, in Section 4, we briefly outline future trends towards power-aware systems. We conclude this chapter with a list of references and bibliography.

8.2 Power Estimation Models: From Spreadsheets to Power State Machines

Total energy dissipation for the execution of a system or application task on the target architecture is obtained by summing respective energies of all system components. Total energy consumption per component is further analyzed as the accumulation of the power dissipation for all transitions in the state machine(s) of the component during application execution. The dynamic and static power dissipation for each possible transition of a hardware component constitutes its power state model; notice that a number of basic operations may be executed during each state transition, e.g., read, write, and wait operations.

The power state model can be abstracted using equations involving entropy, computed using low-level technological (structural and functional) parameters of hardware blocks (e.g., accelerators and interconnect) or approximately evaluated using cycle-accurate system-level simulation combined with a macro-modeling (C/C++ or SystemC) library containing datasheet information providing an energy view, e.g., for memory or general purpose processor.

For different target architectures and application-specific workloads, system-level power macro-modeling does not provide a high degree of absolute accuracy or fine granularity, e.g., exact instantaneous amplitude of power peaks, since design details are unavailable at the behavior or algorithm level. However, it features generality and flexibility, efficient and relatively accurate analysis of system power management, required cooling and performance, or quality of service requirements and a much shorter modeling time, therefore leading to significantly improved design and time-to-market. It also enables cycle-accurate SystemC models for virtual platforms including finite states machines (control), data path components, memory, and processors.

For systems with few data dependencies, e.g. in mathematical software with regular or fixed activity patterns, we can estimate static activity and use a static spreadsheet model to estimate power consumption of memory accesses, data path, control path, and interconnect functions. This model is simple and common, especially for early phase "back-of-the-envelope" calculations, resolving bottlenecks and facilitating rapid exploration of design partitioning. For example, Powerplay has a web-based spreadsheet interface based on a library of power models at several accuracy levels to model constant to complex activity-sensitive systems [9].

For complex data-dependent conditionals, branches and loops, dynamic spreadsheet models using regression-based approximations or dynamic profiling techniques are slower but more accurate than static spreadsheets. These models work by computing transition probabilities using theoretical analysis or gathering switching activity statistics using experimental behavioural-level simulation. As input, they also require typical variations in user-supplied input vectors and operating modes; semiconductor technology parameters, e.g. basic energy costs: per gate transition, instruction or bus transaction and circuit complexity metrics. A few examples are given next.

An RTL cycle-based power analysis tool called SPA is based on activity profiling of design entities and signals in the data path, control path (FSM), memory and interconnect for typical instruction and data inputs [10–12].

The power estimation tool ESP [13] which targets a RISC processor using a fixed-activity power model which has a part proportional to the number of bit transitions in the input vector.

Similarly, instruction-level power estimation models for embedded, general-purpose and DSP processors execute an instruction loop on the target processor [14]. Average power consumption for different instructions can be stored in a lookup table with axes specifying averages of input signal probability and number of (zero delay) input/ouput transitions per cycle. The model can consider pipeline stalls, cache misses and inter-instruction effects due to additional power consumption during state changes between executing pairs of instructions, but not hazards or glitches.

For arithmetic operations, the dual bit type model (DBT), used in Sente's commercial tool WattWatcher/Architect [15], provides an analytical structure-based activity model for the data path. This technique is based on the observation that fixed-point, two's-complement data streams are characterized by two distinct activity regions, resulting in two distinct effective capacitance coefficients; least-significant data bits exhibit activity similar to uniformly distributed white

noise, while most-significant sign bits depend on the sign transition probability, which is related to the temporal correlation of data streams.

The Power Factor Approximation (or PFA) method experimentally models power of RTL functional blocks based on word length, hardware complexity and activation frequency parameters [16].

Power state machines (PSMs) are fine grain FSM-like graph structures. For each block, they contain abstract states representing different operating modes with power dissipation annotations, and edges representing state transitions through a sequence of operational states also annotated with power costs and transition delays. State transitions are driven by external stimuli (events) coming from the environment. PSMs capture system power consumption through executable specifications describing components, interaction among components and workload behavior. They can be specified in HDLs or SystemC and avoid limitations of spreadsheets in estimating system power [17]. They are now parts of evolving standards in power models and power management, e.g. Advanced Configuration and Power Interface (ACPI) for PC_s [18] and are extensively used for DPM.

8.2.1 Power Models of Processors

Maximum power consumption for a processor tends to increase by a factor of a little more than two every four years. Application-specific CISC or DSP processors offer substantial energy savings and performance gains using energy-aware compiler optimization and code generation based on a clean instruction set architecture offering enhanced parallelism with several different functional units and register files. Power savings are obtained through instruction reordering, smart data allocation to memory banks or register files and packing instructions [19–21].

RISC processors can be optimized for low power using different techniques.

- Static or dynamic supply voltage downscaling that adapts its voltage or frequency to the workload [22]. Many processor families have low power versions with reduced voltage.
- Clock gating and operand isolation to avoid useless switching activity in idle units. CPU energy values can be deduced from datasheets or measured at physical-level per groups of instructions and operating mode, e.g., normal, voltage/frequency scaling or halt, which turns off on-chip components [23].
- Specialized instruction sets for a specific workload, e.g., parallel SSE3 instructions, special addressing modes, or native multiply-accumulate instructions. Execution of program instructions translates to switching activity at the processor circuitry causing charging or discharging of node capacitances, resulting in dynamic power dissipation [24–25]. The two basic components of an instruction power model therefore are:
  - Base energy costs associated with instruction execution. This cost is estimated by executing a loop with several instances of this instruction, [26] for

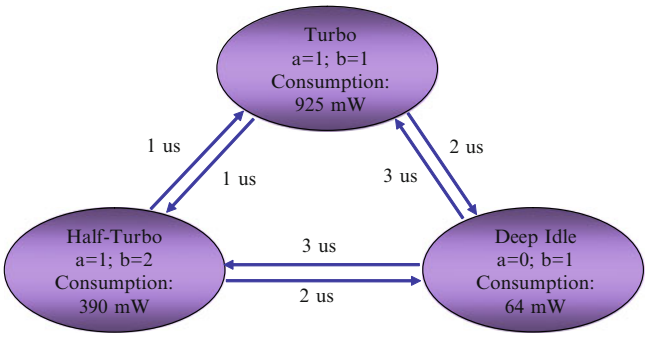


Fig. 8.1 FSM modeling DVFS and DPM modes for the MIPS32 processor

different input signal probabilities and number of (zero delay) input/ouput transitions per cycle.

– Energy consumption overhead costs due to switching activity in the processor circuitry caused by the execution of adjacent instructions. This cost is estimated by executing a loop based on a sequence of alternating instructions, but not hazards or glitches. This cost is estimated by executing a loop based on a sequence of alternating instructions. Pipeline stalls or cache misses can be treated in a similar way.

As an example, Fig. 8.1 illustrates an FSM example of PXA270-based power-aware resource management of the MIPS32. The power state machine of Marvell's (formerly Intel) PXA270 application processor family [27] includes several DVFS modes, e.g. Turbo and Half-turbo, and several dynamic power management (DPM) modes with different switching latencies, e.g. Deep Idle. The best configuration is normally selected based on task execution time.

8.2.2 Power Models of Memory

Storage is required to support computation. Accurate prediction of memory performance (access time, bandwidth) and power consumption are key challenges in system design, especially for emerging data intensive applications. Memory systems (including cache) in multicore SoCs often consume most of the total energy. This claim is even more substantiated with multicore SoC, where memory occupies most of the chip area.

Memory power dissipation is associated with the energy cost of accessing instructions or data and depends on memory size, organization, and access patterns or replacement policies. A memory system consists of the cell array, the decoders, and the control circuit, i.e., sense amplifiers. Memory power corresponds to static and dynamic power consumed when interacting with these components. For write

operations, power consumption depends on the Hamming distance between the old and the new value, while for read access, it depends on the Hamming distance between the current value and literal 0×00. Notice that DRAM power is also consumed during idle periods and read operations at the cell array, due to refreshing. Power dissipation at the decoders can be expressed in terms of the Hamming distance between the current address and the previous one. Power dissipation at the control circuitry is represented by a constant value, unique for each kind of operation (Write, Read, Idle).

Designers have considered dynamic tuning, circuit design and automated synthesis of power-efficient application-specific hierarchical multi-bank memory systems each with its own clock and refresh signals [28]. They have also considered implementing DPM techniques, e.g. clock and power gating (or no refreshes) based on different operating modes, dynamic voltage or frequency scaling, efficient data representation through encoding or compression and software transformations that optimize data allocation and reduce the code size, the number of memory accesses, or data switching activity in intensive array processing applications. For example, refer to Data Transfer and Storage Exploration (DTSE) methodology[29].

8.2.3 Power Models of On-Chip Interconnects

On-chip communication design methodology must adapt to the challenging performance, power consumption, and reliability demands placed by DSM technology. In this context, since communication, rather than computation, emerges as a more significant power and performance constraint, it is necessary to leverage on-chip interconnect properties at system-level, shifting existing power reduction techniques from the era of single microprocessors to modern multicore architectures, where joint optimization across multiple design variables at multiple abstraction levels is necessary.

One of the greatest performance bottlenecks of future Systems-on-Chip is the high cost of on-chip communication through global wires. Innovative multicore architectures that allow different cores to communicate to each other via the Network-on-Chip (NoC) paradigm have emerged as a promising alternative to traditional bus-based approaches. By eliminating global wires, NoC-based multicore systems provide scalability and predictability, while facilitating design reuse.

Nowadays, power consumption has also emerged as a first-order design metric and wires contribute up to 50% of total chip power in some processors. Together with power consumption, thermal constraints appear to dominate other physical constraints, such as pin-bandwidth and silicon area.

The relative impact of communication performance and power consumption on multicore SoC design has steadily increased with new technology. Communication layering based on distributed network technology, data and address encoding, packetization multiple clock domains with self-timed circuits, and dynamic voltage and frequency scaling for efficient task mapping, routing path allocation or link

speed assignment are fundamental techniques that reduce power. Energy-efficient channel design has focused on all layers, especially the physical and data link layer.

At the **physical layer**, technology characteristics of interconnect wires, transmitters and receivers, as well as modulation and communication channel coding and signalling are important. At this level, performance and energy optimizations can be based on low swing differential signaling [30], distributed synchronization and self-timed asynchronous protocols which resolve system clock-generation/distribution and provide modularity and robustness. We can also reduce power consumption by decreasing frequency and especially (quadratically by decreasing voltage (below 1V) on high-capacitance, shared long wires; the effective capacitance for such wires is much larger than that of local wires, since several computations and storage communicate over the same channel. In fact, technology parameter variations in future power-efficient multicore systems-on-chip compose a multi-disciplinary research area, requiring also careful placement and routing [31] and hierarchical and segmented bus architectures [32].

At the **data link and network layer**, or so called **flit level for on-chip networks**, the physical characteristics of the communication channel and the transmitter/receiver architecture are abstracted. This layer heavily influences communication energy.

At this level, clock gating is a popular power saving technique used in many synchronous circuits. It inserts extra control logic into the design (usually manually) to control enable conditions attached to registers and prune the clock tree, thus disabling portions of the on-chip network components, such as network interface or router sub-blocks. Then, corresponding flip-flops do not change state, their dynamic power consumption is reduced to zero and essentially only leakage dissipation is incurred. Moreover, chip area is reduced, since a large number of multiplexers is replaced with clock gating logic. As the granularity of clock gating approaches zero, power consumption of the synchronous circuit approaches that of an asynchronous circuit, i.e. the circuit would generate logic transitions only when it is actively computing. Although asynchronous circuits by definition do not have a "clock", the term perfect clock gating is used to illustrate how different clock gating techniques are simply approximations of the data-dependent behavior exhibited by asynchronous circuitry.

Moreover, redundant (signal) data encodings schemes for error detection and retransmission or power-hungry error correction protocols also help minimize switching activity by exploiting temporal data correlation through extra network control wires that specify how data has been encoded and improve compatibility and communication reliability [33]. Notice that a reduced switching activity does not immediately translate into power savings, since the power consumed by encoder and decoder circuits has to be taken into account. A large number of variations combining data bus invert techniques with address bus encodings have been recently proposed, including applications to partitioned bus architectures and adaptive encoding in special-purpose application data streams.

The **transport layer** optimizes network resources and control flow for providing end-to-end quality of service. Connection-oriented protocols can be energy

inefficient under heavy traffic conditions due to retransmissions and additional computation energy spent at destination nodes. In addition, network flow control techniques reduce throughput to avoid network contention and congestion by regulating the amount of data that enters the network; this has been observed in wireless networks [34–35].

By focusing on channel design, buses are generally considered non scalable interconnects, since when bus size increases, contention and arbitration time increase superlinearly [36]. Buses are also less energy-efficient than distributed on-chip communication architectures, since data is either broadcast from one initiator to one target or multicast to a small subset of targets [37–38]. Moreover, packet-switching is better than circuit-switching for irregular and non-stationary communication patterns; notice that hybrid solutions are also possible.

Power consumption of on-chip interconnect can dominate total chip power consumption for large packet sizes [39]. Moreover, for a large number of nodes, power consumption for on-chip communication networks is estimated to be smaller than that of a similar performance non-partitioned on-chip bus or segmented bus design [40]. A fair comparison between bus and NoC approach requires actual layout experiments on a real application, as well as a consideration of power management techniques for different subsystems, including clock or power gating and layer-by-layer GALS paradigm that allows a subsystem to run at the lowest frequency compatible with application requirements. Nevertheless, a NoC allows for distributed arbitration and improved floorplanning by accommodating smaller wire length and associated capacitance load compared to a bus that fans outs its wires to all targets. A high-level dynamic NoC power estimation model defines the total energy consumed for transmitting a single bit within the NoC, from ingress to egress point, to energy consumed in intermediate routers, internal buffers, and interconnect wires [41].

8.2.4 Power Models for Embedded Software

System and application software play a major role in managing component service levels and corresponding power consumption. Although low-power software libraries and primitives are usually application specific (e.g., for graphics, visualization, multimedia or arithmetic), certain general principles for power reduction are common.

Storage of application or system programs and data consumes power due to memory refresh operations in DRAM or static power consumption in SRAM. Power consumption due to instruction and data storage can be reduced by shortening the instruction length, e.g., through recoding, or compressing the object code with automatic on-the-fly decompression and execution.

Moreover, profiling techniques for several architectures indicate that OS, system software, batch or interactive application software, and compiler can all affect energy consumption due to different computation, communication and

synchronization tasks that cause different power consumption on the underlying hardware structures. While OS-based power reduction focuses on power hungry tasks, such as context switching, system and application software examine empirical source code rewriting by exploring regular activity patterns and locality or compiler-based code transformation based on automatic code generation technology and high-level synthesis. Although traditional compilers attempt to optimize code for high performance and short compilation time (e.g., using speculative execution), multicore SoC compilers must focus on high performance and energy-efficiency at the expense of longer compilation time.

Algorithmic- and architecture-specific source-level compiler transformations (called specializations) optimize control and data flow (including memory access) in data-intensive applications, such as multi-dimensional array processing in multimedia and signal processing. Several efforts try to quantify the effects of hardware-software partitioning and advanced compiler transformations on power and energy, often using processor simulation and power analysis tools operating at RTL or instruction-level. These methods include (see [17, 42, 43]):

- analysis of source code or manipulation of an equivalent symbolic control-data flow graph model of the application for sending clock gating, and voltage or frequency downscaling directives at the entry points of non critical delay paths; notice that dynamic scaling sets the voltage level for each individual loop using linear programming techniques.
- instruction selection using dynamic programming, e.g. in register vs. memory access;
- multiple instruction packing or out-of-order instruction scheduling to reduce inter-instruction effects causing high data transition activity,
- data allocation in memory banks for scheduling efficiently array operations using proper allocation of SoC registers, buffers or multi-banked hierarchical memory arrays based on common patterns or collected traces (called memory access dependency graphs). This includes techniques that raise the concurrency level, such as pre-fetching, software or hardware caching to improve instruction or program locality, replacing storage with redundant computation, and performing data flow transformations, such as conditional branch pre-computation for reducing the number of loads or stores and enabling clock gating in idle functional units and rewriting arithmetic operations using shifts, data reordering based on associative/distributive properties, and small integer calculation. In respect to control flow, loop transformations and parallel memory array access have been examined performance-wise in detail in high-performance parallelizing compilers. These techniques move loop invariants, loop unrolling, loop distribution and tiling, loop fusion and nested loop permutation to reduce cache miss ratio, affecting also power-efficiency.

Previous research efforts have also considered application embedding in symmetric NoC topologies, studying different routing algorithms, buffer size allocation, and switch arbitration policies. Hu and Marculescu examined power-efficient mapping of a heterogeneous 16-core task graph representing a multimedia

application into a Mesh NoC topology [44–46], while Murali and De Micheli used a custom tool (called Sunmap) to map a heterogeneous 12-core task graph representing a video object plane decoder and a 6-core DSP filter application into a Mesh or torus NoC topology using different routing algorithms [47–48]. The proprietary Sunmap tool, proposed by Stanford and Bologna University, performs NoC topology exploration by minimizing area and power consumption requirements and maximizing performance characteristics for different routing algorithms. The Xpipes compiler can eventually extract efficient synthesizable SystemC code for all network components, i.e. routers, links, network interfaces and interconnect, at the cycle- and bit accurate level.

Another study focuses on extending and parameterizing existing open-source partitioning tools from parallel processing, as well as evaluating the embedding quality through bit- and cycle-accurate OMNeT++ (C++-like) simulation models. By considering mapping common tree-like synthetic task graphs (representing master-slave communication) and an mpeg4 decoder application on conventional NoC topologies, e.g. mesh and torus, as well as low-cost circulant graphs, such as the Spidergon STNoC, the study concludes that for realistic network sizes (below 64 nodes), Spidergon is more cost-efficient than traditional topologies [49].

Finally, another important publication focuses on application traffic. Communication weighted application models consider communication aspects (CWM), while communication dependence and computation models (CDCM) simultaneously consider both application aspects. For current technologies, CDCM model embedding into regular NoCs results in average reductions of 40% in NoC execution delay and 20% in dynamic energy consumption [50].

8.2.5 Power Estimation, Analysis, and Optimization Tools

Electronic design automation tools for accurate and efficient power modeling, analysis, and optimization of multicore systems are fundamental for low-power and power-aware design. Increasingly aggressive power optimization is especially helpful for portable consumer electronic devices with long battery life and small weight, and systems with environmental concerns, e.g. energy star compliant systems.

Power analysis and verification tools are needed at every abstraction level of the design flow (from transistor-, to gate-, RT-and behavior-level) to ensure that power specifications along with cost, size, performance, and time to market constraints are never violated. In fact, this is known as feed-forward design: a design does not progress to lower abstraction levels (e.g. synthesis or subsequent layout) until the architecture satisfies all specifications. Thus, power analysis tools focus on addressing large consumers of power dissipation, such as improving the SoC architecture, optimizing the mapping, modifying the data representation or rewriting the application software through low power and power-aware algorithm design and implementation.

Traditional power estimation tools concentrate on power-driven synthesis. Low-level power estimation tools, such as SPICE derivatives or Mentor Graphics's Lsim at transistor-level and Synopsys Power Compiler or Cadence's InCyte Chip Estimator at gate-level, take advantage of floor-planning information to provide increased accuracy. Moreover, a variety of power reduction techniques (e.g. clock, voltage and frequency scaling on less critical paths of the circuit) and semiconductor fabrication technology variations are able to support low voltage.

While low-level power estimation is useful for design validation or late optimization, there is now an increasing trend towards applying abstract power estimation methods based on system-level modeling using C/C++ or SystemC. Numerous emerging tools from Academic research, start-up companies and SMEs perform system-level power estimation using behavioral synthesis of C/C++ or SystemC executable specifications of hardware modules, thus enabling early design decisions without relying on hardware synthesis. Some tools provide not only the total energy consumption at the end of the simulation, but also evolution of power consumption with time. Since a direct top-down translation from C/C++ or SystemC TLM to silicon design flow methodology is not currently possible, power estimation macro-models use spreadsheets or back annotation from structural or behavioral RTL which may not always be available.

- ChipVision's Orinoco is a system-level design space exploration tool chain estimating performance and power for running different algorithms (specified in ANSI-C or SystemC) on different architectures [51–52]. The algorithm compiles to a hierarchical control data flow graph (CDFG) describing the expected circuit architecture without resorting to complete synthesis. CDFG nodes represent power-characterized operations, edges represent control and data dependencies among operations, and nested procedure calls correspond to transitions between successive hierarchy levels. Compositional rules compute the total cost of a complex CDFG depending on the implementation. Components are instrumented with area, dataflow and switching activity using a standard power library for the target technology, consisting of functional units, such as adders, subtractors, multipliers, and registers.
- Synopsys Innovator is a SystemC-based integrated development environment for virtual platform developers to efficiently integrate, analyze and verify transaction-level models [53]. Early estimates from RTL simulation can be back annotated through a graphical user interface into system-level virtual platform models created in the recently announced Synopsys Innovator environment to estimate power consumption and develop power management software.
- HyPE is a high-level simulation tool that uses analytical power macromodels for fast and accurate power estimation of programmable systems consisting of datapath and memory components [54].
- Web-based JouleTrack estimates power of an instruction-level model specified in C for commercial StrogARM SA 1100 and Hitachi SH-4 processors [55]. SoftExplorer is similar to Jouletrack, but focuses on commercial DSP processors [56]. Other similar tools providing power models for processor with an

Instruction Set Simulator (ISS) are Simunic [57] and Avalanche [58], while Lajolo [59] uses RTL simulation and can be linked to Avalanche for hardware-software co-simulation.

- Alike Lajolo, Powerchecker [60] uses slower RTL simulation incorporating power estimation model for hardware components. BullDast's PowerChecker works on a mixed RT-gate level description obtained through source HDL analysis, elaboration and hardware inferencing [61]. Design objects are annotated with real switching activities obtained through RTL simulation.
- BlueSpec [62]. PowerSC [63] and Power-Kernel [64] are frameworks built by adding C++ classes on top of SystemC for power-aware characterization, modeling and estimation in multiple levels of abstraction. Unlike other tools, Power-Kernel is open source. It provides an efficient object-oriented library for SystemC 2.0, which allows simple introduction of a SystemC power macro model at RT-level of a complex design [64]. PK achieves much higher simulation speed than lower-level power analysis tools. Power instrumentation is based on a SystemC class that uses advanced dynamic monitoring and storage of I/O signal activity of SoC blocks through put_activity and get_activity functions [65]. Both constant power models and more accurate regression-based models with a linear dependence on clock frequency, gate and flip-flop switching activity are used. As an example, dynamic energy estimation of the AMBA AHB bus is decomposed into arbiter, decoder and multiplexing logic for read and write operations (master to/from slave). The latter operations are estimated to control over 84% of the total dynamic power consumption. Similar power instrumentation techniques for synthesizable SystemC code at RTL level are described in [66].

8.2.6 Standardization and Power Formats

The influence of power formats to SoC design opens the possibility to thoroughly examine and propose new power modeling, analysis and optimization requirements within future power format standardization efforts. Moreover, it is beneficial for the industry to provide a common definition, estimation, analysis and optimization methodology for static and dynamic power reduction, and set (as much as possible) common conditions (and computational interfaces) for power labeling in different embedded SoC platforms.

While clock gating can be expressed adequately in HDLs today, the same is not true for power distribution, power modeling and power gating through switches and supply nets. Existing EDA industry initiatives toward standardizing power formats addresses rising concerns on low power and power-aware systems in a holistic approach toward power management. EDA vendors have already created two initiatives, Common Power Format (CPF) and Unified Power Format (UPF), that try to express and communicate power intent in a consistent way throughout the RTL-to-GDSII design flow, verification and implementation. CPF, initially developed by Cadence, is now being taken forward by the Silicon Integration Initiative

(Si2), while UPF, initially supported by Synopsys, Mentor Graphics and Magma Design Automation, is now IEEE standard P1801 (UPF2.0) [67]. Both formats are being successfully deployed, easing low-power and power-aware design and verification challenges, although designers are often caught in the middle of a standards war between two competing formats. Despite that the two formats have not yet converged into a single standard, designers expect them to do so soon, due to the need for flexibility, portability, and interoperability of tools. However, it is not clear yet how soon these two camps will find common ground for agreement.

Power tool designers and multicore SoC architects must equally contribute their requirements to standardization of power methodology and data, especially in terms of higher levels of automation, such as analog and mixed signal design, system-level multicore SoC design and verification, extended power domains, and new power state models for embedded software development for a full-fledged system-level support of power management. These contributions can realize significant improvements in productivity and quality of results by having a single, open, and portable file format with which a designer can easily and consistently specify, modify, extend, and maintain complex power model behavior and design data in different EDA tools. This effort could enable end-user support from leading EDA vendors and customers for industry-wide adoption of interoperable low-power and power-aware methodology.

8.3 Power Management

Dynamic power management (DPM) is a widely used strategy for reducing system energy consumption while a chip is powered and tasks are running [68]. The key idea underlying all DPM-based approaches is to put part of a system into a low-power (and low performance) state to save energy when that subsystem is not working during a suitably long time-period determined by the shutdown and wakeup overhead of the subsystem.

Recent research efforts usually focus on power or energy reduction during execution using dynamic voltage/frequency scaling (DVFS) techniques, which control the supply voltage and clock frequency depending on each task's computational requirements. Since dynamic power consumption in a CMOS circuit scales quadratically with the supply voltage and linearly with the frequency, significant power gains are expected by using DVFS techniques.

Nowadays, power-efficient techniques need to shift from the era of single microprocessors to modern multicore SoC architectures. As CMOS technology is continuously scaling, single chip systems integrating a large number of processors, on-chip memories and custom intellectual property cores (IP cores) have become a reality [69–70]. In fact, most major chip manufacturers have already announced plans for large-scale chip multiprocessors (CMPs) [71–72]. Innovative architectures that allow different cores to communicate to each other via the NoC paradigm have

emerged as a promising alternative to traditional bus-based approaches [73]. By eliminating global wires, the NoC approach provides the needed scalability and predictability, while facilitating design reuse (refer also to the end of Section 8.2.3).

With communication and synchronization posing critical and complex power and performance constraints than computation, it is necessary to understand and leverage the properties of the interconnect fabric at a higher level. More specifically, one of the greatest bottlenecks to performance in future systems-on-chip is the high cost of on-chip communication through global wires [74]. Moreover, power consumption has emerged as a first order design metric, with wires contributing up to 50% of total chip power for certain processors [75]. Therefore, multithreaded workloads executing on multicore processors dissipate significant power in the on-chip interconnect and experience high on-chip communication latencies, as well as network bandwidth and scalability problems.

In addition, thermal constraints appear to dominate other physical constraints like pin-bandwidth, bisection bandwidth and silicon area. Together with power consumption, clock distribution and (technology) parameter variation problems in future multiprocessor systems-on-chip, they compose a multi-disciplinary equation, whereas joint optimization across multiple design metrics from different areas is necessary. For this reason, the design of efficient adaptive multicore SoC architectures that can dynamically accommodate computation-, memory- and communication-bound workloads is envisioned.

Since application traffic has varying characteristics and is often unpredictable at the time of SoC development, researchers argue that online management is necessary. Ideally, power estimation, analysis and optimization tools and models should be relatively accurate and cost-effective in respect to hardware and software architecture requirements. Low level power tools operating on the circuit and RTL level, such as Synopsys PowerMill and Mentor Graphics' QuickPower, provide excellent accuracy, but are not practical for design space exploration and corresponding system architecture decisions. In addition, as discussed in Section 2, power models are used to evaluate the energy-efficiency of proposed architectures, both during development (offline), and more aggressively during system execution (online). Online power models used in dynamic power management policies, for which speed is a first-class constraint, cannot rely on detailed simulation. Thus, instead of simulated activity counts or complex analytically calculated energy functions, real-time system events are used to resolve this drawback.

At the task level, since a modern chip is divided into regular tiles, whereas each tile can be a general-purpose processor, a DSP, a memory subsystem, etc, the application is divided into a graph of concurrent tasks and the system designer must decide on which task must be mapped on each core, so as to optimize an objective function. By considering different NoC topologies and generic design methodologies mostly based on using simulated annealing, we can achieve power-efficient mappings for regular architectures and different static routing schemes [45, 76] (see also Section 2.3). However, non-deterministic workloads require dynamic adaptation of system parameters, such as voltage and frequency or throttling and thread migration. Thus, tasks with inherent dynamic behavior may incur situations

which are not always observable with offline profiling. Moreover, these cases must be treated by specialized architectural system attributes e.g. dynamic routing protocols and shared IP cores for processors with different time-varying behavior.

8.3.1 Categorization of Management Techniques

Different online management techniques have been explored acting either proactively or reactively, i.e. only when emergency situations appear. These methodologies can be categorized as follows.

- DPM techniques focus on mechanisms to scale the voltage or frequency level or even shut-down links. Network statistics, such as buffer utilization, can be used to drive the communication link on/off decision policy depending on the current traffic [77]. However, at the same time, path diversity is reduced, potentially harming NoC connectivity. Thus, based on a power-performance connectivity graph, a deadlock-free routing (or alternatively, complex deadlock recovery) algorithm is needed to ensure packet delivery irrespective of the total number of link candidates that are off during network operation. Such techniques can work in conjunction with power-aware buffer mechanisms, proposed for supplementary power savings in interconnection networks when links are on and operational [78].
- Dynamic throttling of the workload reduces power consumption under specific energy or thermal constraints. In particular, reducing bandwidth (e.g. for memory accesses), or throttling communication link traffic are complementary policies to achieve dynamic system power management.
- Dynamic management of data transmission via encoding techniques reduces switching activity and/or tackles signal integrity (crosstalk) effects. By detecting bit transition patterns on a communication link, encoding hardware converts data into a low-transition form before transmission. Use of special coding to reduce crosstalk between wires and avoid adversarial switching patterns on the bus has been examined [79–82]. Alternative techniques, such as serialized low energy transmission coding for on-chip interconnect networks (SILENT), aim at reducing the switching activity in the serial link by employing differential encoding [83]. Combinations of both serialization of data and encoding schemes have also been proposed to deal with energy-efficient link transmission [84].
- Dynamic activation of recovery policies can tackle the effect of voltage or frequency scaling for different fault rates. For instance, Razor describes support for adaptive failure rate monitoring for timing faults [85].

8.3.2 Dynamic Monitoring for Power and Thermal Management

Monitoring techniques are increasingly employed at the integration level of modern SoCs. Dynamic management for temperature, power, clock jitter, supply noise,

process variation and performance behavior are becoming an integral part of today's SoCs. For instance, the IBM Power6 processor employs 24 critical path monitors (CPMs) distributed across the chip which guarantee correct circuit operation under different process, voltage and temperature conditions [86]. These monitors not only identify perturbations of process variation, supply noise effects, aging effects, clock instability, but also provide corrective actions in order to prevent circuit failures.

As system operating frequencies increase and power supply voltages are reduced, transient faults start to occur causing increasing device soft error rates at the macro-architecture level. Dynamic management has been employed for the detection of soft errors in processor core logic. Backward recovery through checkpoint and rollback is a popular approach used in modern processors to recover from these types of transient faults. Also, with the dual modular redundancy technique, two redundant processors execute simultaneously, with an execution error in one processor manifesting itself as a deviation in the behavior of the two processors. This deviation is evaluated using a "fingerprint" comparison of the states of the two processors at regular checkpoint intervals. A checkpoint of a program state consists of a snapshot of the registers and memory at a specific point of time. A checkpoint interval is the time between two successive checkpoints. A fingerprint is a hash value that summarizes the states of the processors after every instruction in the checkpoint interval. If the fingerprints for both processors agree at the end of the checkpoint interval, all instructions executed during the interval are known to be correct. If the fingerprints disagree, then, the processor must be rolled back to the last correct state of execution, which is the checkpoint at the beginning of the current interval.

Industrial examples of on-chip dynamic monitoring, such as Intel's Itanium processor, utilize voltage, thermal and power sensors [87]. The integrated feedback control monitor, referred to as Foxton technology, utilizes on-chip sensors to measure power and temperature. To optimize performance under power and temperature constraints a microcontroller modulates both voltage and frequency, so that processor cores perform computations at optimal power efficiency. OMAP2420 is another industrial processor from TI demonstrating a SoC partitioned into several power managed IPs [88]. Each IP's power control interface is connected to a global power manager which is controlled by software. Different power saving modes are implemented, including idle (clock stopped), retention for low leakage and fast re-start, and power-off mode for ultra low leakage. Power switches are used to connect each local IP to a global power plane. If a particular domain is off the local plane drifts to a potential near ground. As reported in [88] the design exhibits a 2 to 2.4 leakage reduction for voltage scaling and 3.4 to 4 reduction for SRAMs in retention compared to the active mode. When all power domains are in off-mode a 40 leakage reduction is achieved versus active leakage at room temperature.

On-chip thermal sensor implementations that exploit the temperature coefficient of a forward biased diode voltage have been proposed, while also ring oscillator based temperature sensors are widely employed; these exploit the linear

dependence on junction temperature to achieve a controlled oscillation frequency which is indicative of the temperature inside the chip. Researchers have also presented cascade current mirror-based frequency output thermal sensors and low area overhead differential temperature sensors [89] or process variation tolerant thermal sensor designs with active compensation circuitry [90].

The joint optimization of performance with controlled temperature and power requires intelligent policies, especially as multicore SoCs are emerging. Such a monitoring system that allows collaboration between a processing and thermal monitor is demonstrated in [91]. The processing monitor evaluates whether the processor is operating within expected parameters, by comparing the results of an offline analysis of the system binary to runtime information obtained from the processor core. Temperature information from the ring-based oscillator thermal monitor is correlated with monitoring graphs representing the application running on the processing monitor to allow for more robust evaluation.

More recently, as NoC gains importance as a viable alternative to on-chip buses due to better scalability and power-performance, monitors also emerge as a service layer, but pose several additional challenges that must be addressed:

- The number and location of the monitors. Due to the increased number of cores, the number of monitors must also follow this trend.
- The type of the monitors. General purpose monitors are either too complicated or cost-ineffective because of the diversity of the monitoring process.
- Throughput requirements and circuit resources. Differences in monitoring for functions present diverse requirements which are also affected by NoC size.
- Interface and interactions with existing NoC. Monitoring functions can be implemented as services over the existing links of a NoC, or as separate monitoring cores using a private, secondary, network on chip.

Monitoring mechanisms in power and thermal management are generally necessary to measure NoC parameters at runtime and improve traffic attributes, enhance quality-of-service, predict deadlock or livelock, and avoid congestion or unfair use of resources. Monitoring of NoC communication parameters can be performed at any of the communication layers of NoC protocol stacks. However, since monitoring mechanisms need to function at least at the speed of each considered NoC layer, in order to capture accurate statistics of the NoC traffic, they are often supported by hardware monitoring agents. On the contrary, operating conditions are relaxed when monitoring temperature fluctuations which are modeled at wire speed.

DVFS algorithms are typically implemented in the operating system. Thus, the operating system scheduler is enhanced in order to monitor the application phases, and request for core power mode transitions occurring (when necessary) at the millisecond time scale. Isci et.al. have recognized the importance of monitoring application phase activity at finer time scales and have proposed using a global power manager framework to reevaluate DVFS decisions at intervals on the order of hundreds of microseconds [92]. However, most proposed state-of-the-art DVFS-based power management schemes incur a large transition delay for voltage and

frequency in order to achieve the target power mode. The voltage transition delay, which is on the order of tens of microseconds, is due to off-chip voltage regulators that limit how quickly voltage can change; the frequency transition delay results from PLL relock times. These transition delays fundamentally limit re-evaluation of application behavior and re-mapping core voltages and frequencies at finer time scales. In contrast, micro-architectural events, such as cache misses introduce application variability at nanosecond granularities. In addition, micro-architectural reactive techniques in the form of clock gating or power saving pipelines and throttling provide for dynamic management in the order of nanoseconds.

The concept of NoC-based voltage island architecture focuses on minimizing power consumption using fixed supply assignment based on application traffic patterns available at design time [93–94].

A DVFS-enabled island organization is depicted in Fig. 8.2. Local network conditions in each island can be adaptively adjusted by a DVFS monitor, which collects network information on separate, narrow links from each of the switches in the island. The island voltage is the output of a voltage regulator, and the frequency is determined by one PLL. The configuration of the voltage regulator and PLL is set by the DVFS monitor from a discrete number of voltage and frequency pairs. Between the islands, FIFOs are needed to interface different frequency domains.

Constant increase of the size of NoC-based architectures raises the need for a scalable approach when designing run-time monitoring services, supported controllers and communication among them. Physically separate networks significantly reduce switching and arbitration complexity in the communication fabric which provides energy efficiency but costs more wiring overhead. It allows the maximal flexibility in configuring the networks adaptively based on the monitoring traffics on different architectural levels. Virtual channels are another alternative to decouple monitoring information from data traffic, while reservation of bandwidth is an effective method to achieve predictable and guaranteed average latency.

Communication monitoring is most of the times a priority class and thus needs to be treated with guaranteed services, decoupled from the data traffic. Emergencies

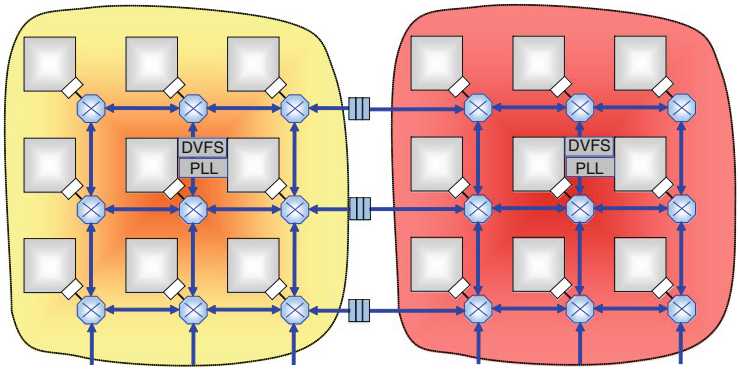


Fig. 8.2 A NoC-based voltage/frequency island architecture

must be immediately identified raising the need for monitor packets which should have high priority and routed over very fast connection paths. Distributed monitor architectures and filtering of information are often required. However, scaling to large NoCs in order to reduce the bandwidth requirements of monitoring communication and avoid energy overheads is also important.

8.4 Future Trends

Power-efficiency is an important concern in emerging complex, energy-aware applications in deep submicron multicore SoC designs due to increased power dissipation and thermal heat dissipation which increase packaging and cooling costs and reduce reliability.

A low-power and power-aware future of electronic devices is currently driven by advances in device manufacturing, multicore SoC architecture, system and application software, algorithm design and EDA methodology and tools focusing on monitoring, estimation, analysis, and optimization is forthcoming. For example, new technological breakthroughs low power true-single-phase clocked (TSPC) flip-flops and latches, circuits that can return excess energy to the supply, limited-swing circuits, optical interconnects based on wavelength division multiplexing and parallel or asynchronous systems-on-chip are examined.

Moreover, new system-level tools, frameworks and methodologies based on power macro-models or instruction-level models can support efficient low power or power-aware design space exploration and easy technology migration of existing IPs beyond the RT level. As shown in Fig. 8.3, as we progress from high to low level design, tools become more accurate, but also about an order of magnitude slower, thus being able to handle much smaller circuits. Since performance and software power are addressed very early in the design, benefits for early software development and product differentiation are expected to be large, while equivalent

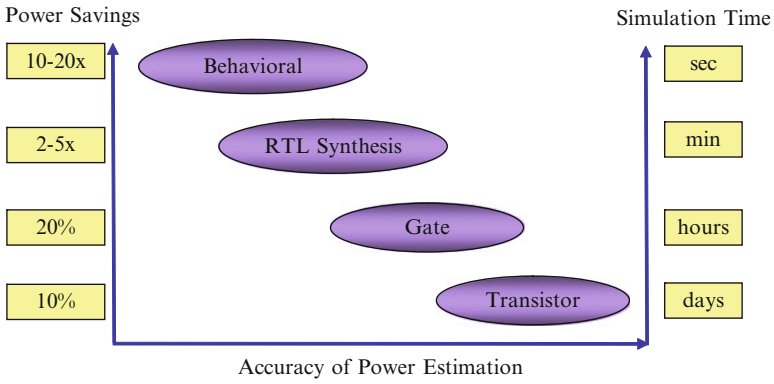


Fig. 8.3 Cost, simulation efficiency and accuracy of power estimation at various abstraction levels

RTL models are not necessary. Moreover, by exploring innovative algorithmic, architectural and technology-related features we can perform:

- accurate and efficient variation-aware power analysis by focusing on the distribution of several metrics of key system components rather than a single deterministic and absolute metric, and
- promising performance and power prediction during technology migration.

Thus, system-level power methodology and tools must exploit several concepts, such as code-rewriting, lightweight system monitoring threads and power instrumentation, targeting power estimation, analysis and optimization at sufficient accuracy and improved performance, power consumption, and productivity compared to RTL design flow. Despite a sacrifice in power estimation accuracy due to unavailable low-level (physical and structural) information, efficient, entry-level dynamic power estimation models can be based on bit- and cycle-accurate transaction-level SystemC macro-models. For example, switching activity is usually computed by multiplying transactions and/or bit transitions for all gate signals in all components with appropriate bit energy coefficients. Notice that for absolute power estimation results, calibration against current technologies can be based on statistical experiments and linear regression. The computations can also be grouped, e.g. as shown below:

- at input/output interface ports, including registers and local signal drivers,
- memory, by capturing read, write and idle transactions and bit transitions at row/column decoders and cell array, and
- FSM and data path components represented as binary decision diagrams (nodes corresponding to gates), by evaluating bit transitions at input and output signals, as well as at the output of each individual gate.

Acknowledgements Work of the first author towards this project has been indirectly funded by ISD S.A. and in particular, EU sources: a) ARTEMIS/SCALOPES "SCAlable LOw Power Embedded platformS" Joint Undertaking under grant agreement n° 100029 (duration: 2009–2010), and b) ENIAC MODERN "MOdeling and DEsign of Reliable, process variation-aware Nanoelectronic devices, circuits and systems" under reference n° ENIAC-120003 MODERN (duration 2009–2011), and corresponding Greek funding authorities.

References

1. M. Coppola, M.D. Grammatikakis, R. Locatelli, G. Maruccia, and L. Pieralisi, "Design of cost-efficient interconnect processing units: Spidergon STNoC", CRC Press, Inc., (2008).
2. T.Cohen, N. Sriram, D. Leland, Moyer, et al., "Soft Error Considerations for Deep-Submicron CMOS Circuit Applications", in Proc. IEEE International Electron Device Meeting (IEDM), pp. 315–318, (1999).
3. H.J. Veendrick, "Short-Circuit Dissipation of Static CMOS Circuitry and its Impact on the Design of Buffer Circuits", J. Solid-State Circ., SC-19 (4), pp. 468–473, (1984).
4. T. Burd et. al., "A dynamic voltage scaled Microprocessor System", in Proc. Int. Solid State Circ. Conf., (2000).

5. A. Chandrakasan and R. Brodersen, “Low power digital CMOS design”, Kluwer Academic Publisher, (1995).
6. A. Chatzigeorgiou and G. Stephanides, “Evaluating performance and power of object-oriented vs. procedural programming in embedded processors”, LNCS 2361, J. Blieberger and A. Strohmeier (Eds.), Springer-Verlag, pp. 65–75, (2002).
7. H. Mehta, R.M. Owens, and M.J. Irwin, “Some issues in Gray code addressing”, in Proc. Great Lakes Symposium on VLSI, pp. 178–180, (1996).
8. E. Macii, M. Pedram, and F. Somenzi, “High level power modeling, estimation and optimzation”, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 17, pp. 1061–1079, (1998).
9. D. Lidsky and J. Rabaey, “Early power exploration: A world wide web application”, in Proc. Design Automation Conf., (1996).
10. P. Landman, “Low-Power architectural design methodologies”, Ph.D. Dissertation, UC Berkeley, (1994).
11. P. Landman and J. Rabaey, “Architectural power analysis: The dual bit type method”, IEEE Transactions on VLSI Systems, 3(2), pp. 173-187, (1995).
12. P. Landman and J. Rabaey, “Activity-sensitive architectural power analysis”, IEEE Trans. on CAD, 15(6), pp. 571–587, (1996).
13. T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago, “Evaluation of architecture-level power estimation for CMOS RISC processors”, in Proc. Symp. Low-Power Electr., pp. 44–45, (1995).
14. V. Tiwari, S. Malik, and A Wolfe, “Power analysis of embedded software: a first step towards software power minimization”, IEEE Trans. VLSI, 2(4), pp. 437–445, (1994).
15. WattWatcher Product Sheet, Sente Corp., Chelmsford, MA, (1995).
16. S. Powell and P. Chau, “Estimating power dissipation of VLSI signal processing chips: The PFA technique”, J. VLSI Signal Proc., Vol. IV, pp. 250–259, (1990).
17. L. Benini, and G. De Micheli, “System-level power optimization: techniques and tools”, ACM Transactions on Design Automation of Electronic Systems, 5(2), pp. 115–192, (2000).
18. ACPI,http://www.teleport.com/~acpi/
19. V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez, “Dynamic power management for microprocessors: A case study”, VLSI Design, 185–192, (1997).
20. V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee, “Instruction level power analysis and optimization of software,” VLSI Design, 13(2), pp. 223–238, (1996).
21. M. Caldari, M. Conti, M. Coppola, P. Crippa, et al., “System-level power analysis methodology applied to the AMBA AHB bus”, in Proc. Design Automation and Test in Europe Conf., pp. 32–37, (2003).
22. T.D. Burd and R.W. Brodersen, “Design issues for dynamic voltage scaling”, in Proc. ISLPED, pp. 9–14, (2000).
23. C. Kulkarni, F. Catthoor, and H. De Man, “Advanced data layout organization for multimedia applications”, in Proc. IPDPS - Workshop on Parallel, Distributed Computing in Image Processing, Video Processing and Multimedia, (2000).
24. V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez, “Dynamic Power Management for Microprocessors: A case study”, VLSI Design, pp. 185–192, (1997).
25. V. Tiwari, S. Malik, A. Wolfe, and M. T-C. Lee, “Instruction level power analysis and optimization of software”, J. VLSI Signal Proc., 13(2), pp. 223–238, (1996).
26. V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: a first step towards software power minimization”. IEEE Trans. VLSI, 2(4), pp. 437–445, (1994).
27. Intel PXA27x Processor Family, Electrical, Mechanical, and Thermal Specification, Technical Report, (2005).
28. M. Farrahi, G. E. Tellez, and M. Sarrafzadeh, “Memory segmentation to exploit sleep mode operation”, in Proc. Design Automation Conf., pp. 36–41, (1995).
29. F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, et al., “Data access and storage management for embedded programmable processors”, Kluwer Acad. Publ., (2002).

30. H. Zhang and J.M. Rabaey, “Low-swing interconnect interface circuits”, in Proc. Int. Symp. Low Power Electr. and Design, pp. 161–166, (1998).
31. M. Pedram, and H. Vaishnav, “Power optimization in VLSI layout: A survey”, J. VLSI Signal Processing, 15(3), pp. 221–232, (1997).
32. L. Xie and P. Qiu, and Q. Qiu, “Partitioned bus coding for energy reduction”, in Proc. Asia South Pacific Design Automation Conf., pp. 1280–1283, (2005).
33. D.Bertozzi, L. Benini, and G. De Micheli, “Low-Power Error-Resilient Encoding for On-Chip Data Busses”, in Proc. Design Automation and Test in Europe Conf., pp. 102–109, (2002).
34. J. Walrand, P. Varaiya, High-Performance Communication Networks. Morgan Kaufman, (2000).
35. I. Papadimitriou, M. Paterakis, “Energy-conserving access protocols for transmitting data in unicast and broadcast mode”, in Proc. Int. Symp. Personal, Indoor and Mobile Radio Communication, pp. 416–420, (2000).
36. A. Tanenbaum, “Computer networks”. Prentice-Hall, Englewood Cliffs, NJ, (1999).
37. C. Patel, S. Chai, S. Yalamanchili, D. Shimmel, “Power constrained design of multiprocessor interconnection networks”, IEEE Int. Conf. on Computer Design, pp. 408–416, (1997).
38. H. Zhang, M. Wan, V. George, J. Rabaey, “Interconnect architecture exploration for low-energy configurable single-chip DSPs”, IEEE Computer Society Workshop on VLSI, pp. 2–8, (1999).
39. T.T. Ye, L. Benini , and G. De Micheli . “Packetization and routing analysis of on-chip multiprocessor networks”, J. Syst. Arch. - Special Issue on Networks on Chip, 50 (2-3), pp. 81–104, (2004).
40. P.T. Wolkotte, G. J.M. Smit, N. Kavaldjiev, Jens E. Beckerand J. Becker, “Energy model of networks-on-chip and a bus”,. in Proc. Int. Symp. System-on-Chip, pp. 82–85, (2005).
41. M.R. Stan and W.P. Burleson, “Low-power encodings for global communication in CMOS VLSI”, IEEE Trans. VLSI Syst., 5, pp. 444–455, (1997).
42. D. Brooks, V. Tiwari and M. Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations”, in Proc. ProcInt. Symp. Comp. Arch., (2000).
43. M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye, “Influence of compiler optimizations on system power”, in Proc. 37th Design Automation Conf., (2000).
44. J. Hu and R. Marculescu. “Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures”, in Proc. Design, Automation and Test in Europe Conf., (2003).
45. J. Hu and R. Marculescu. “Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints”, in Proc. Design, Automation and Test in Europe Conf., (2004).
46. J. Hu and R. Marculescu. “Energy- and performance-aware mapping for regular noc architectures”. IEEE Trans. Computer-Aided Design of Integr. Circ. and Syst., 24(4), pp. 551–562, (2005).
47. S. Murali and G. De Micheli. “Bandwidth-constrained mapping of cores onto NoC architectures”, in Proc. Design, Automation and Test in Europe Conf., (2004).
48. S. Murali and G. De Micheli. “SUNMAP: a tool for automatic topology selection and generation for NoCs”, in Proc. Design Automation Conf., (2004).
49. L. Bononi, N. Concer, and M. Grammatikakis, “System-level tools for NoC-based multicore design” in Embedded Multicore Architectures. Ed. G. Kornaros, Chapter 6, CRC Press, Taylor and Francis Group, (2009).
50. C. Marcon, N. Calazans, F. Moraes, and A. Susin. “Exploring NoC mapping strategies: an energy and timing aware technique”, in Proc. Design, Automation and Test in Europe, (2005).
51. S. Rosinger, K. Schroder, and W. Nebel, “Power management aware low leakage behavioural synthesis”, Int. Conf. Digital Syst. Design, pp. 149–156, (2009).
52. ChipVision, “Orinoco: A high-level power estimation and optimization tool suite”, see http://www.chipvision.com

53. Synopsys Innovator, Datasheet. Available from http://www.synopsys.com/virtualplatform
54. X. Liu and M.C. Papaefthymiou, "HyPE: hybrid power estimation for ip-based programmable systems", in Proc. Asia and South Pacific Design Automation Conf., pp. 606–609, (2003).
55. A. Sinha and A.Chandrakasan, "JouleTrack – A web-based tool for software energy profiling", in Proc. Design Automation Conf., pp. 220–225, (2001).
56. E. Senn, J. Laurent, N. Julien, and E. Martin, "Softexplorer: estimating and optimizing the power and energy consumption of a C program for DSP applications", EURASIP J. Appl. Signal Proc., Vol 1, pp. 2641–2654, (2005).
57. T. Simunic, L. Benini, and G. D. Micheli, "Cycle-accurate simulation of energy consumption in embedded systems", in Proc. Design Automation Conf., pp. 867–872, (1999).
58. J. Henkel and Y. Li, "Avalanche: an environment for design space exploration and optimization of low-power embedded systems", Transactions on VLSI Systems, 10, pp. 454–468, (2002).
59. T. M. Lajolo, A. Raghunathan, S. Dey, and L. Lavagno, "Efficient power co-estimation techniques for system-on- chip design," in Proc. Design Automation and Test in Europe Conf., (2000).
60. BullDast, Powerchecker: An integrated environment for rtl power estimation and optimization, Version 4.0, available from http://www.bulldast.com
61. PowerChecker by BullDAST, see http://www.bulldast.com/powerchecker.html.
62. Bluespec, see http://bluespec.com
63. F. Klein, G. Araujo, R. Azevedo, R. Leao, et al., "PowerSC: An efficient framework for high-level power exploration", in Proc. Midwest Symp. Circ. and Syst., pp. 1046–1049, (2007)
64. L. Pieralisi, M. Caldari, G.B. Vece, M. Conti, et al., "Power-Kernel: Power analysis methodology and library in SystemC", in Proc. VLSI Circ. and Syst., Vol. II, (2005).
65. M. Caldari, M. Conti, M. Coppola, P. Crippa, et al. "System-level power analysis methodology applied to the AMBA AHB Bus", in Proc. Design Automation and Test in Europe Conf., (2003).
66. S. Xanthos, A. Chatzigeorgiou, and G. Stephanides, "Energy estimation with systemC: A programmer's perspective", in Proc. WSEAS Int. Conf. on Systems, Computational Methods in Circuits and Systems Applications, pp.1–6, (2003).
67. IEEE P1801, available from http://ieeexplore.ieee.org
68. L. Benini, A. Bogliolo, and G. D. Micheli, "A survey of design techniques for system-level dynamic power management", IEEE Trans. VLSI, 8(3), pp. 299–316, (2000).
69. ITRS, 2009, http://www.itrs.net
70. The European Design Automation Roadmap, available from http://www.medeaplus.org
71. P. Hofstee. Power efficient processor architecture and the Cell processor, in Proc. HPCA-11, (2005).
72. P. Kongetira, K. Aingaran, and K. Olukotun, "A 32-way multithreaded SPARC processor, IEEE Micro, 25, pp. 21–29, (2005).
73. W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks", in Proc. Design Automation Conf., (2001).
74. R. Ho, K. Mai, and M. Horowitz, "The future of wires", in Proc. IEEE, 89(4), (2001).
75. N. Magen, A. Kolodny, U. Weiser, and N. Shamir, "Interconnect power dissipation in a microprocessor", in Proc. System Level Interconnect Prediction, (2004).
76. Y. Hu, Y. Zhu, H. Chen, R. Graham, and C.-K Cheng, "Communication latency aware low power NoC synthesis", in Proc. Design Automation Conf., pp. 574–579, (2006).
77. V. Soteriou and L.-S. Peh, "Exploring the design space of self-regulating power-aware on/off interconnection networks", IEEE Trans. Parallel Distrib. Syst., 18(3), 393–408, (2007).
78. X. Chen and L.-S. Peh, "Leakage power modeling and optimization in interconnection networks", in Proc. Symp. Low Power Electr. and Design, (2003).
79. P.-P. Sotiriadis and A. Chandrakasan, "Bus energy minimization by transition pattern coding (TPC) in deep submicron technologies", in Proc. Int. Conf., pp. 322–327, (2000).

80. T. Lv, J. Henkel, H. Lekatsas, and W. Wolf, "A dictionary-based en/decoding scheme for low-power data buses", IEEE Trans. VLSI Syst., 11 (5), pp. 943–951, (2003).
81. M.R. Stan and W.P. Burleson, "Bus-invert coding for low power I/O", IEEE Trans. VLSI Syst., 3 (1), pp. 49–58, (1995).
82. V. Wen, M. Whitney, Y. Patel, and J. Kubiatowicz, "Exploiting prediction to reduce power on buses.", in Proc. Symp. High Proc. Comp. Arch., pp. 2–13, (2004).
83. K. Lee, S.-J. Lee, and H.-J. You, "SILENT: serialized low energy transmission coding for on-chip interconnection networks", in Proc. Int. Conf. CAD, pp. 448–451, (2004).
84. G. Kornaros, "Temporal coding schemes for energy efficient data transmission in Systems-on-Chip", in Proc. Workshop Intelligent Solutions in Embedded Systems, pp. 111–118, (2009).
85. D. Ernst, N.S. Kim, S. Das, S. Pant, et al., "A low-power pipeline based on circuit-level timing speculation", in Proc. Int. Symp. Micro-architecture, pp. 7–18, (2003).
86. A. Drake, R. Senger, H. Deogun, G. Carpenter, et al., "A distributed critical-path timing monitor for a 65nm high-performance microprocessor ", in Proc. Solid-State Circuits Conf., (2007).
87. R. McGowen, C. A. Poirier, C. Bostak, J. Ignowski, et al., "Power and Temperature Control on a 90nm Itanium Family Processor", IEEE Journal on Solid State circuits, 41 (1), pp. 229–237, (2006).
88. P. Royannez, et. al., "90nm low leakage SoC design techniques for wireless applications", in Proc. Solid State Circuits Conf., (2005).
89. C. Qikai, M. Meterelliyoz, and K. Roy, "A CMOS thermal sensor and its applications in temperature adaptive design", in Proc. Int. Symp. Quality Electronic Design, (2006).
90. S. Remarsu and S. Kundu, "On process variation tolerant low cost thermal sensor design in 32nm CMOS technology", in Proc. Great Lakes symposium on VLSI, pp. 487–492, (2009).
91. T. Wolf, S. Mao, D. Kumar, B. Datta, W. Burleson, and G. Gogniat, "Collaborative monitors for embedded system security", in Proc. Workshop on Embedded Syst. Security, (2006).
92. C Isci, G Contreras, and M Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management", in Proc. Int. Symposium on Microarchitecture, (2006).
93. L.F. Leung and C.Y. Tsui, "Energy-aware synthesis of networks-on-chip implemented with voltage islands", in Proc. Design Automation Conf., pp. 128–131, (2007).
94. U.Y. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu, "Voltage-frequency island partitioning for gals-based networks-on-chip", in Proc Design Automation Conf., pp. 110–115, (2007).

Part IV
Trends and Challenges for Multiprocessor Systems

Chapter 9
Embedded Multicore Systems: Design Challenges and Opportunities

Dac Pham, Jim Holt, and Sanjay Deshpande

Abstract Embedded systems have evolved into sophisticated on-chip collections of processor cores, on-demand acceleration, and input/output interfaces. These systems enable increased performance in terms of system throughput and better overall efficiency than ever before. Yet, this power comes at the cost of increased complexity for system designers as well as for system programmers. This chapter explores in depth the opportunities that multicore systems provide for the embedded application space, and the challenges associated with multicore systems design as well as several innovative approaches to dealing with those challenges.

Keywords Embedded Multicore Systems · Multicore Systems Design · Multicore Systems Performance · Multicore Interconnect · Multicore Software Standards

9.1 Introduction

Over the last decade, technology scaling has dramatically increased leakage power in CMOS circuits. With gate dielectrics and other device features fast approaching fundamental limits, a continuation of historical trends would see passive power surpassing active power within the next few years. Furthermore, the conventional techniques for improving single thread performance (e.g., increased frequency and deeper/wider processor pipelines) have reached the point of diminishing returns when power is taken into consideration [1–3]. In the face of this power/performance wall, increased system efficiency becomes essential.

With each technology node system designers have significantly more transistors at their disposal. This opens new avenues for innovation to extend system integration and achieve performance and efficiency improvements. Thus, the advent of multicore SoC created great opportunities to increase overall system performance

D. Pham (✉)
Freescale Semiconductor, Inc., Austin, TX, USA
e-mail: B06187@freescale.com

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_9,

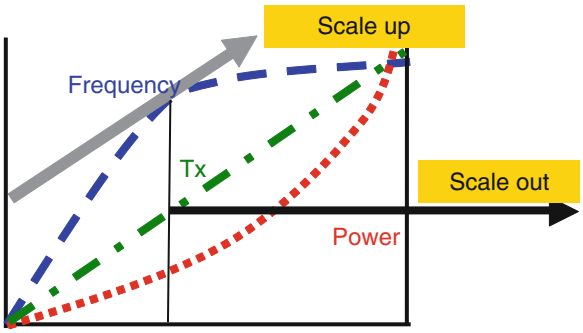


Fig. 9.1 Processor design challenges

while keeping power in check. As shown in Fig. 9.1, designers continue to leverage system integration through advancement in technology and are starting to take a right hand turn for performance "scale out" instead of frequency "scale up."

While this high degree of system integration (e.g., utilizing multiple processor cores, specialized hardware acceleration units, and numerous I/O interfaces) provides continuing opportunity for performance improvement, it also creates new design challenges that must be overcome. We explore the "real world" requirements that are driving current and future multicore SoC and discuss the challenges and opportunities associated with designing and using these multicore SoC chips.

9.2 "Real World" Requirements

Multicore SoC chips have a number of intrinsic characteristics which simultaneously distinguish them from previous generations of chips and enable them to provide new levels of system efficiency. These characteristics arise as a result of requirements from a world that is rapidly changing. At the heart of this rapid change are two important technology trends that span application domains: (a) the demand for higher performance at constant power and (b) the demand for higher levels of system integration. Combined with these technology trends, a few important historical and future market trends are fueling massive industry growth. We examine the most important of these in more detail below.

9.2.1 Continuing Demand for Higher Performance at Constant Power Envelope

The challenge of doubling performance every 2 years used to drive superscalar processor design with more functional units running concurrently or deeper pipelines racing to achieve the highest possible frequency at the cost of higher power. However, this ever-increasing application performance requirement can no longer

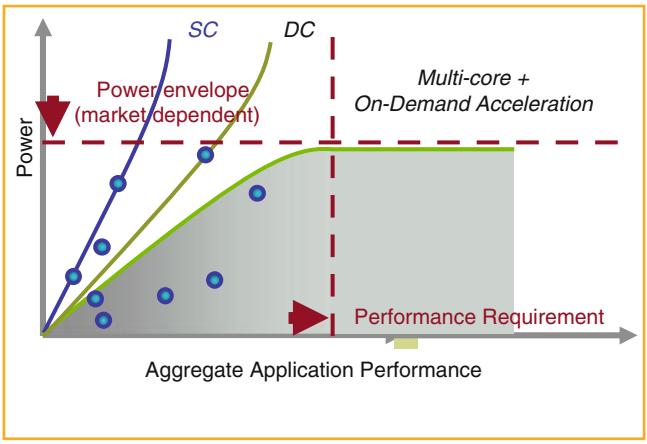


Fig. 9.2 The ever-increasing demand for performance

be sustained without leveraging multicore and on-demand acceleration (Fig. 9.2). For example, the expanding demand for richer, higher definition, higher fidelity content will require networks that are more responsive, more interactive, and lower cost [4]. Furthermore, the requirement of connectivity anywhere and anytime with stringent security will create monumental performance demands on infrastructure and access systems. An answer to these increasing performance demands is a multicore processing approach.

9.2.2 Demand for Higher System Integration

Market pressures coupled with the availability of billions of transistors in today's 45 nm SoCs are driving Multicore SoC to take on many of the functions typically associated with larger systems [5–8]. These pressures span a range of applications from high-performance down to deeply embedded.

Network infrastructure providers, for example, are driven to reduce the cost of running their operations by integrating functionality from several rack blades into a single blade. This in turn drives silicon providers to achieve higher levels of system integration on a single chip. A typical SoC in the networking domain now encompasses functions of the management processor, control plane processors, data plane processors, and offload and acceleration.

Similarly, today's automotive Multicore SoC combines adaptive engine control to meet emissions and fuel-economy standards, advanced diagnostics for repair, new safety features, and new comfort and convenience features. This higher system integration not only increases system performance and throughput but also reduces overall system cost.

9.3 Industry Growth Drivers and Sustainable Megatrends

Across the industry, multicore is being driven by ever-increasing demands for computational power; these computational demands come especially from emerging application domains that (a) exploit interactivity and connectivity and (b) make the world a safer place by augmenting our ability to manage complexity in a fast-paced environment. These trends unfortunately expose end-users to security and privacy risks, ultimately requiring additional bandwidth and computational power to mitigate exposure. Thus, the shape of future multicore SoC will be determined not only by *historical trends* (Fig. 9.3) but also by a few *sustainable megatrends* encompassing many emerging application domains. In the following sections, we discuss important historical trends and identify three important megatrends: the *Interactive World*, the *Connected World*, and the *Safer World*.

9.3.1 Interactive World

As our world becomes more interactive, the demand for systems capable of creating "virtual immersion" through sensory computing also increases, and the user becomes part of this virtual world. Gamers, and filmmakers are a few pioneers in this area, see Fig. 9.4.

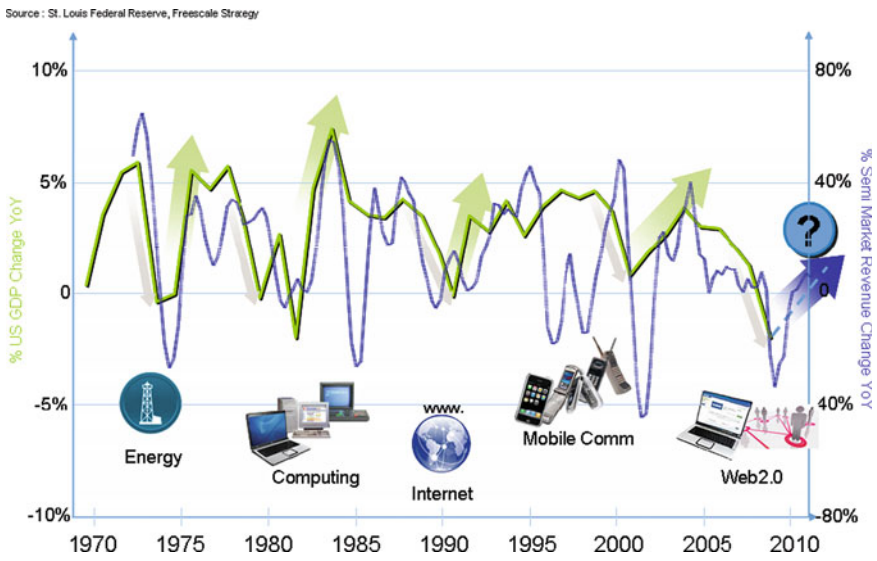


Fig. 9.3 Historical growth drivers

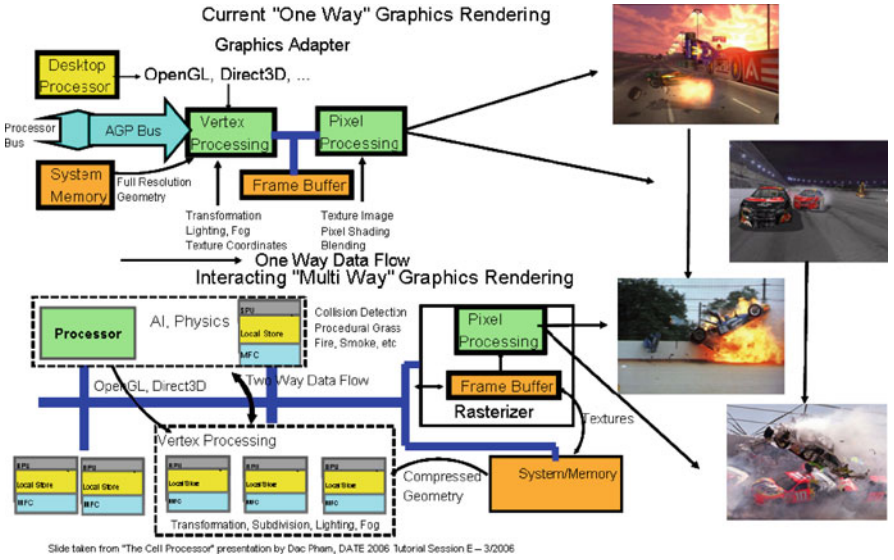


Fig. 9.4 Computing demand from an ever-increasing interactive world

9.3.2 Connected World

With billions of smart connected devices, the network is converging from multiple formats to packet processing. This unified network, with wireless leading the way, is driving the exponential demand for computing power. Social networks on-line merchants, etc., are, etc driving the need for a trusted and secure network (Fig. 9.5).

9.3.3 Safer World

Automotive and health care are other examples of drivers for embedded multi-core systems. In automotive, more and more active and predictive safety measures will be standard in tomorrow's vehicles (Fig. 9.6). The ability to do lane detection warning and obstacle detection are examples of applications that will drive computing power from image processing, pattern recognition, etc.

9.4 Distinguishing Multicore SoC Features

Multicore systems have evolved under pressures from two forces: (1) CMOS technology scaling is providing the ability to produce ever higher levels of integration of cores, hardware accelerators, and input/output devices into a single cost-effective

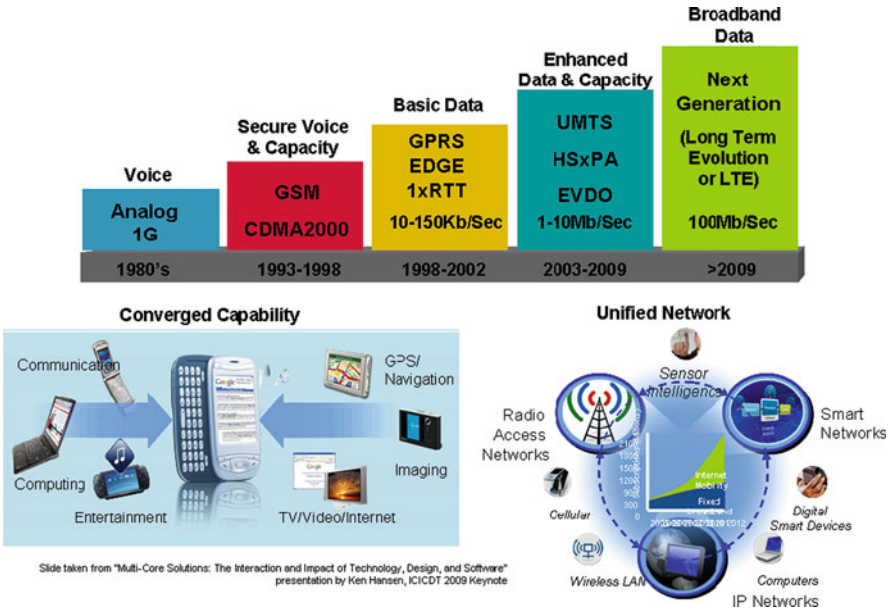


Fig. 9.5 Computing demand from an ever-increasing connected world

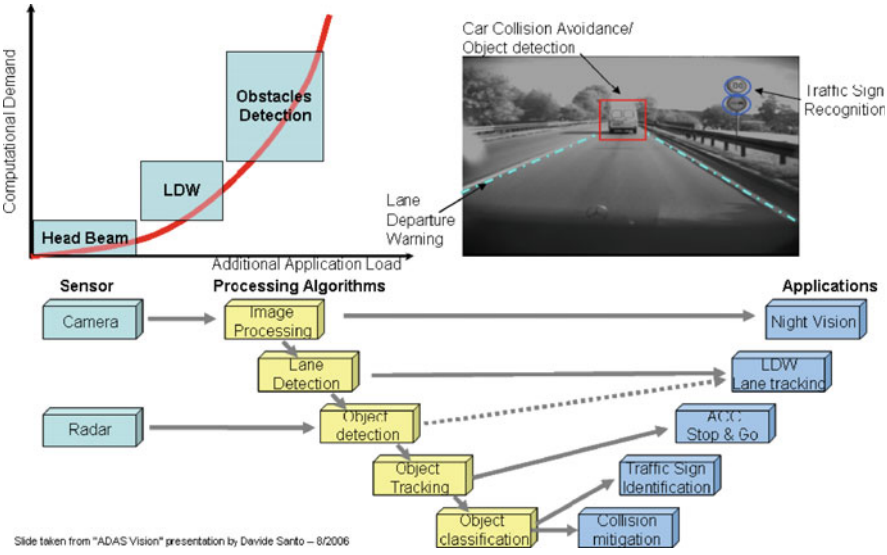


Fig. 9.6 Computing demand from an ever increasing safer world

package, and (2) customers want to save component costs and power consumption while achieving increased system performance. Two important features of the systems designed to meet these demands are support for virtualization and the heterogeneous system that this allows.

9.4.1 *Virtualization*

Problems can occur when no single operating system image is in control of all the resources in a Multicore SoC. This can be solved with the introduction of virtualization. A virtualized Multicore SoC is divided into a set of partitions, with each partition being considered a virtual machine (VM). Software running within each virtual machine appears to be running on its own hardware machine. A VM may not be aware that there are other VMs running on the same device.

Virtualization is enabled by an additional layer of software called a hypervisor which is inserted between the hardware and the VMs (see Fig. 9.7). The hypervisor software has the responsibility to ensure that each virtual machine has access to required resources without any contention or security issues from other virtual machines. Successfully enabling virtualization requires support in the processor core as well as system-level support. Processor cores must have an additional privilege state (the hypervisor state) which supercedes the system level that operating systems run in. Interrupts and timers must also be virtualized to be delivered to the right virtual

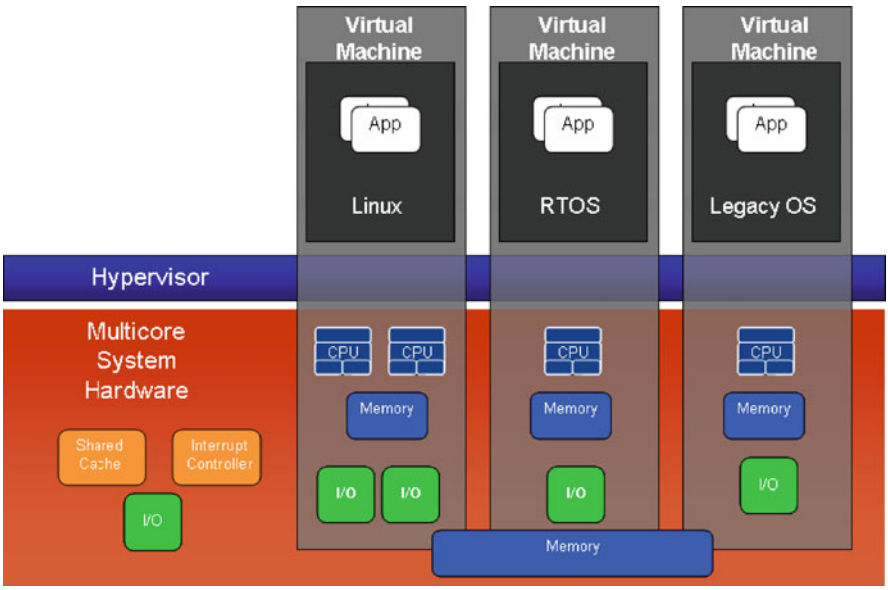


Fig. 9.7 Virtualization – abstraction of underlying hardware

machine. At the system level, virtualized memory protection mechanisms must be added to ensure that memory accesses from input/output devices obey the partitioning of memory that the hypervisor software has done.

The addition of virtualization solves not only the problem of allowing multiple operating systems to share resources effectively, it also enables other advanced capabilities such as rolling upgrades of software (by allowing an older version of an operating system to run alongside a newer version), or partitioning of the system for high security, availability, or quality of service. Such features will be a mainstay of future embedded Multicore SoC.

9.4.2 Heterogeneous Multicore System

Today's Multicore SoC are highly integrated dices that integrate processor cores, memory controllers, input/output devices and on-demand acceleration engines (see Fig. 9.8). These features are complemented with virtualization technologies to allow much flexibility in software configurations chosen for the system. The selection of which components to integrate is a complex systems engineering task that requires expertise in many hardware protocols, broad application knowledge, performance engineering methods, and sophisticated verification and validation methodologies. Each generation of such a Multicore SoC will strive to incorporate more features, enhance performance, and maintain power envelopes. These things are required in order to respond to the rapidly increasing demands for computational power and bandwidth in modern system applications.

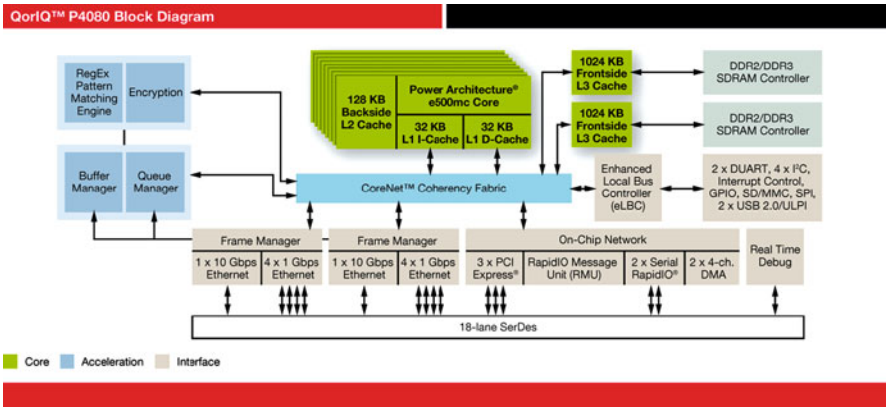


Fig. 9.8 A highly integrated multicore communication platform

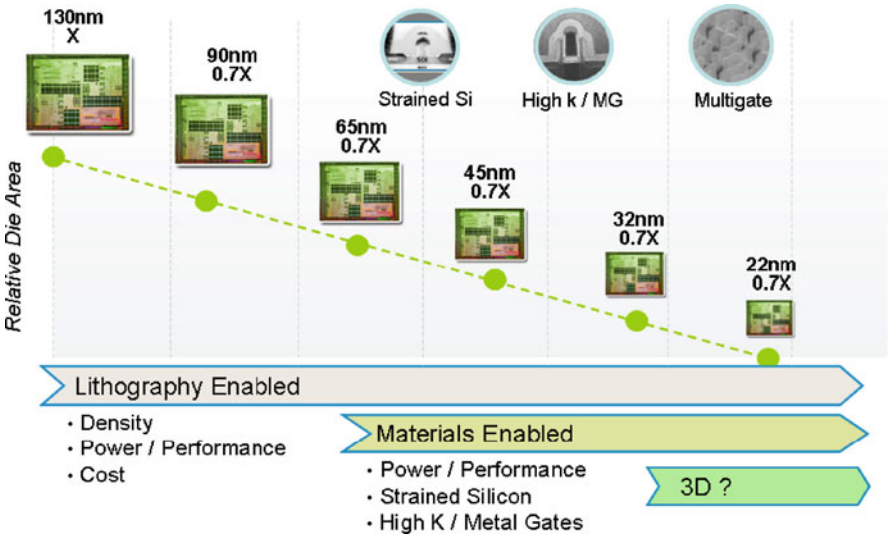


Fig. 9.9 Technologyscaling

9.5 Multicore Design: Key Considerations

Multicore SoC design is an incredibly complex undertaking. A successful design effort must look beyond merely creating a functional device. Competition among silicon providers is fierce and the costs associated with masksets and manufacturing are considerable. Therefore, designers must ensure that system performance, area, and power goals are met.

But multicore also presents new challenges to programmers, including finding and implementing parallelism, debugging deadlocks and race conditions, and eliminating performance bottlenecks [9]. It will take time for most programmers and enabling software technologies to catch up. This is because concurrent analysis, programming, debugging, and optimization are significantly different in concept from their sequential counterparts, and because heterogeneous multicore programming is impractical using standards defined for Symmetric Multi-Processing (SMP) system contexts or for networked collections of computers. To remedy the situation, multicore SoC creators must also address programmer's needs with a multicore programming model and debugging and optimization paradigms that support that programming model.

Below we discuss key aspects of technology scaling, performance, power and area, interconnect, and software in multicore SoC design. This is followed by a brief discussion of the trend toward heterogeneous manycore SoC in the near future and how it might impact these key considerations for multicore design.

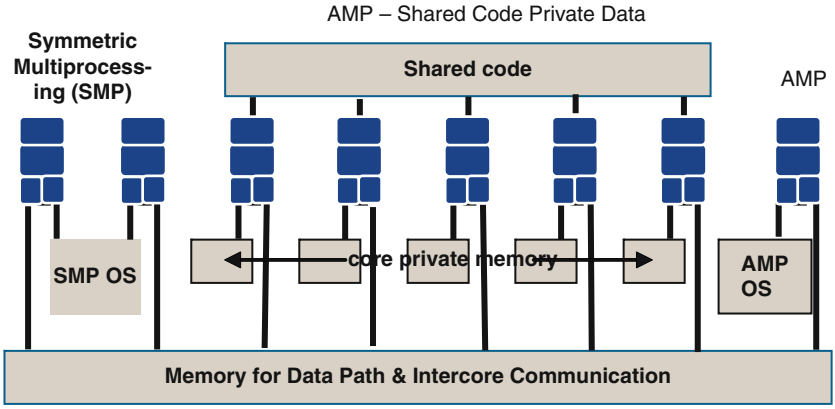


Fig. 9.10 Memory sharing/access Control Example

9.6 Performance

Even in multicore SoC the processor core matters. For example, a core with high single thread performance is well suited for a mixed control and data plane processor. This is because the processor core is able to offload repetitive and compute intensive data plane operations to on-demand acceleration blocks, opening up processing cycles for control plane work, or headroom for new services and applications. As shown in Fig. 9.10, another key consideration is the continuing trend to move larger and faster memories closer to the processing core. Given this trend, and after careful consideration of application memory demands, a new three-level memory hierarchy is introduced in this Multicore Communications Platform (MCP): 32KB of Level 1 (L1) instruction and data caches are included in the processor core, while a private Level 2 (L2) cache is attached to the core in a back-side implementation. This backside L2 cache is connected directly to the CPU, enabling extremely high application performance for most workloads. This technique allows the cache to match the full speed of the CPU, resulting in significant latency improvements compared to a typical shared front side L2 cache.

There are some tasks for which a shared cache is desirable, such as interprocessor communication and operating on shared data structures. For those instances, a Level 3 (L3), multiple-way shared front side cache is used. This shared cache maximizes hit-rates while providing low-latency memory for I/O and accelerator blocks.

9.7 System Bandwidth

The next-generation MCP requires a highly scalable and modular coherent fabric rather than an intercore bus as the interconnecting medium among the cores, memory, and on-chip peripherals (see Fig. 9.8). This fabric eliminates the bus contention issues that other multicore architectures face as more traffic is introduced into the system.

Inherently scalable, the coherent fabric, such as CoreNet in Fig. 9.8, enables coherent, concurrent, low-latency connectivity among cores – easily expanding to accommodate more cores. CoreNet also provides the option of heterogeneous clustering. Therefore, to support this rich scalability and flexibility, the CoreNet fabric works in concert with the caching hierarchy to enable coherent and concurrent accesses through a highly scalable, low-latency implementation.

9.8 Software Complexity

While multicore architectures hold promise for new performance levels, multicore applications software and enablement development is still in the early stages. Clearly, multicore systems will only be as effective as the software's ability to take advantage of concurrency, and the pure processing potential of multicore platforms today is not yet being fully tapped. Another significant challenge is the fact that the majority of the installed base is still operating on a large body of legacy software traditionally deployed on single-core systems. Software developers have the enormous challenge of migrating this large installed base of software code to multicore architectures to take advantage of all the benefits that a true concurrent system can provide. As shown in Fig. 9.11, any combination of SMP and Asymmetric Multi-Processing (AMP) with flexible choice of operating system (OS) can be supported on this MCP.

9.9 SoC Integration

One of the key design challenges for integrating large multicore SoC is controlling the variation between cores in terms of timing and power due to variation in device critical dimension, threshold voltage, doping fluctuation, layout

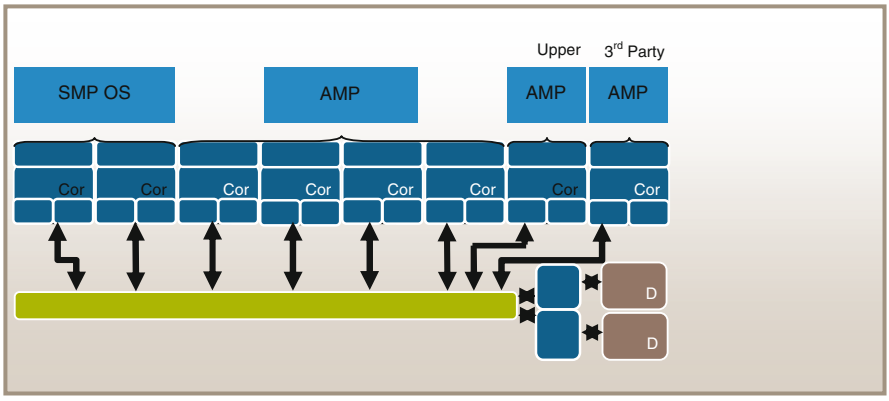


Fig. 9.11 Flexible multicore OS model – any combination of SMP/AMP

matching for strained Si, etc. To minimize these within-chip variations, designers need to design with appropriate margins, using on die sensors to monitor variations, and using asynchronous interfaces to reduce dependencies on the slowest core(s).

Next, to design such a complex system, a concurrent analysis and optimization design methodology is needed for converging functional, power, and timing aspects of the design. A common design database, where all tools can be integrated on a single platform and can be optimized concurrently, is essential to improve design quality, reduce overall effort, and improve schedule predictability.

Other key technology challenges include yield improvement for arrays with separate array supplies, array repairs, Error Correcting Code (ECC), etc. On-demand performance while controlling power can be accomplished with Dynamic Voltage Frequency Scaling (DVFS). In some applications, it is crucial to shut down leakage when not in operation. Multi-voltage domains, sleep device, etc. can be used to manage this. Finally, overall Power Frequency Limited Yield (PFLY) can by accomplished through personalizing operating voltage per device with the usage of Voltage ID (VID).

9.9.1 Area and Power

Most of the opportunities for application of embedded SoCs are accompanied by strict power dissipation limits based on enclosures in which these chips find themselves. As processing power demand has increased, the number of cores and other hardware devices incorporated in these SoCs has gone up to match the demand. This has led to a steady increase in transistor count, which has led to a potential of higher power dissipation in the chips. Chip designers employ multiple techniques to overcome the challenge of growing power ranging from architectural to silicon device design. As discussed earlier, performance is achieved by adding parallelism of multiple cores instead of running fewer cores at higher frequency. These larger number of cores can be run at lower frequency to achieve the same performance. Maximum switching speed of a transistor is inversely proportional to its threshold voltage (V_T). Thus core that can be run at a lower frequency, can be built using more higher V_T transistors where timing is not critical. However, the higher a transistor's threshold voltage, the lower its static leakage current. Thus a slower core can be built to dissipate lower static leakage power using a higher percentage of higher V_T transistors.

Dynamic power associated with switching of the clock is a large portion of dynamic power dissipated by flip-flops in the system. If this switching can be turned off, it can lead to significant reduction in SoC power dissipation. A clock edge is necessary to change the value held by a flip-flop. If the value of a flip-flop is not going to change, the clock edge is unnecessary. Logic techniques that detect if the value of the flip-flop will change and gate off the clock if it will not are now commonly used in the design to limit an SoC's power profile.

Circuits that are supplied with power dissipate at least static power. As device geometries have gotten smaller, this portion of the overall power dissipation has grown steadily. To deal with this, SoCs are also designed with multiple power domains or power islands, each either supported by its own set of power supply pins isolated from other power supply pins in the SoC, or controlled by an on-chip power regulator. For SoCs that are designed with multiple personalities, the disabled portions of the chip can also be left unconnected to the power supply. This helps save even static power that would otherwise be dissipated in those portions of the chip. Power domains that have their own power regulators can similarly be powered down by controlling the regulator either via fuses or via other on-chip means of configuration, in a technique referred to as on-chip power gating.

Power domains controlled by either on- or off-chip power regulators can also be put under software control. Thus when the portion of the chip such as a core is not utilized, it can be powered down dynamically, thereby leading to substantial power savings. This energy saving technique is useful for dynamic power reduction during times of low demand when many of the cores in the chip might be idle. As the demand changes, the software can dynamically adapt the performance and power profile of the SoC.

Maximum switching speed of a transistor is proportional to the supply voltage (V_{DD}) applied to it. But the higher this voltage becomes, the higher will be the static leakage dissipation and the dynamic power spent in switching the transistor's state. Thus V_{DD} has a big impact on the overall power dissipated in the chip in general and in a core in particular. Many SoC therefore specify different V_{DD} for different voltage islands in order to optimize power for a given performance level.

Dynamic change in power profile can be achieved in certain applications at a much finer granularity. In the technique known as Dynamic Voltage and Frequency Scaling (DVFS), based on the current processing demand, the software adjusts the frequency of the cores to match the system's performance. This leads to savings in dynamic power. But as the frequency is lowered, the supply voltage of the core can also be adjusted down opportunistically to save static power as well.

9.9.2 The Critical Role of Interconnect

Multicore systems are characterized by high traffic levels from multiple sources, including processor cores and other hardware assets. Therefore, robustness of the system's interconnect is often critical to the overall performance of the system.

There are a number of considerations that come into play when choosing an interconnection for a multicore system. This section examines some of the more important ones.

The choice of an interconnection for a multicore system is primarily dictated by communication requirements of the applications running in the system.

Processes running in a multicore system could communicate with each other via Shared memory or via Message Passing.

In a shared memory model of interaction, the communicating devices can access a common system address region. Interprocessor communication is achieved via the basic Read and Write operations with low overhead. As a result, shared memory is often the primary method of choice for intercore communication. The key performance parameters for a shared memory system are the latency of Read operations and peak and sustained bandwidths available for Read and Write operations.

In the message passing model, different communicating cores do not share a common address space. Since most current popular processor Instruction Set Architectures lack native extensions to provide message passing primitives at the instruction level, general purpose use of message passing typically involves prohibitively large software overheads. Because of these high overheads, message passing is typically limited to infrequent activity or as door bells between processors with bulk data communication being still carried out via shared memory.

A multicore system may be constructed with a centralized memory organization or it could be distributed. In the centralized organization, all memory is equidistant from all cores and all the data accesses nominally have equal latency. Such a memory organization is called Uniform Memory Architecture (UMA). The UMA model greatly simplifies the task of the software by not having to worry about data placement to achieve efficiency and performance goals. In a distributed organization, not all memory is equidistant from a processor and is therefore referred to as Non-Uniform Memory Architecture (NUMA). In a NUMA system, the performance of the application can be very sensitive to placement of data, which in turn increases the complexity of software. The UMA model, while being physically impractical for large systems with tens of processors, often is employed for small to medium-sized systems.

Producer–consumer is a fundamental model of cooperation between concurrent processes. Storage operation ordering is essential for a pair of producer and consumer processes to work together correctly. Support for enforcing these semantics across the interconnection is essential for multiple cores to cooperate with each other. Coherency is an agreement achieved in the system among various entities regarding the apparent order of values observed to have been stored at a given location. In the presence of caches, hardware-maintained coherency relieves the software of flushing the caches to drive updated data to the memory for other entities to see. Thus, support for operation ordering and coherency is an essential aspect of any interconnection network for modern multicore systems.

9.9.3 Choice of Interconnection Topologies

There is a wide range of topologies available to choose from, each differing in cost, capabilities, and performance [10, 11]. Some are more amenable to implementation on a single die than the other and are therefore more practicable. We discuss a few of these fabric-oriented options and their capabilities and properties.

The primary problem with a standard broadcast bus is the frequency it can be run at, which decreases as the number of devices attached to it increases. Thus, as the

demand for bandwidth increases, the available bandwidth decreases. This has an adverse effect on system performance as the system scales in size [12].

This problem with a broadcast bus can be alleviated by pipelining the bus. Thus, an electrical broadcast of signals within a single cycle is no more longe required. Pipelining the bus increases the latency of transactions, but that increase usually is more than compensated by the increase in bandwidth of the interconnection.

Instead of a single bus, multiple buses might be used and transactions distributed among them. Multiple buses allow the traffic to be partitioned. If the number of buses is B, the cost of de-multiplexers at each device will be proportional to logB. If these multiple buses are pipelined, the cost of the interconnection will grow proportional to $B*N*\log N$, where N is the number of devices attached to the bus.

In a ring topology, the devices form nodes and are connected in the form of a ring with point to point wires connections between neighboring devices. For example, a ring topology is used in IBM's Cell processor [8]. In a ring topology, transactions flow around the ring from device to device. Because the propagation of signals is limited to be between neighboring devices, the ring can be operated at a high frequency. However, the latency is not uniform to all devices; some are farther than others, with average being $N/2$, where N is the number of devices connected in the ring. The cost of ring interconnection grows proportional to N.

In a crossbar topology, during any cycle, there can be N simultaneous pairwise connections active from any of the N devices to any other device. This enables high throughput across the interconnection. Depending on the construction, the interconnection can also support broadcast or multicast functionality. The biggest drawback of a crossbar is its cost, which grows proportional to N^2, where N is the number of devices. However, for small values of N, cost could be acceptable. Depending on implementation, a crossbar can be amenable to partitioning. A Crossbar supports the UMA model.

In a mesh topology, a device sits at every grid point of the mesh. A mesh is a two-dimensional topology well suited for implementing on a die. The cost of a mesh is proportional to N, where N is the number of devices in the system. Like the ring, the latency is not uniform, average being proportional to $\sqrt{N}$. Because of high latency, a mesh is more suitable for a NUMA model with the core at each grid point carrying its own local memory unit. A mesh also exhibits high throughput capability. Because of its properties, a mesh is an attractive choice for a scalable interconnection to support a system with large number of cores.

9.9.4 Software

With highly integrated systems comes a need to produce new software that takes advantage of system resources efficiently. Often legacy software will not run efficiently since it will not have been written to take advantage of new on-demand acceleration features. While virtualization is a key enabler for Multicore software systems, it alone does not alleviate the programmer of the burden of creating software that exploits the capabilities of the system.

With Multicore systems the programmer has to keep in mind that the work of *N* cores does not necessarily equal *N* times the work of a single core, especially for software that was not written properly. Care must be taken to achieve maximum software scalability using such techniques as minimizing software synchronization and serialization, as well as avoiding deadlocks and race conditions. To aid the programmer in this task it is essential that SoC creators take into consideration how programmers will create software, how they will debug it, and how they will optimize it. Multicore SoC systems must provide hardware hooks to enable these capabilities, and optimized system software must be provided to speed the customer's time-to-market.

9.9.5 Heterogeneous Manycore

With the ability to integrate so many things on a Multicore SoC it is inevitable that these systems will be heterogeneous in multiple dimensions, including operating system, instruction sets, and memory uniformity (see Fig. 9.12). This poses an opportunity for system designers working in specialized application domains, while at the same time posing additional challenges for Multicore design and systems programming. The design and programming challenges will be explored in following sections.

If the challenges can be conquered, heterogeneous systems are likely to provide an almost ideal set of features for a given application domain. Examples of things that are commonly integrated onto a single die include general purpose cores, digital signal processing (DSP) cores, and graphics processing units (GPUs). These may be complemented by a specialized memory hierarchy that supports partial concurrency (e.g., shared vs. private memory regions), as well as distribution of memory in space and size that is appropriate for the given application. Such rich combinations of features can be shaped by many different application spaces.

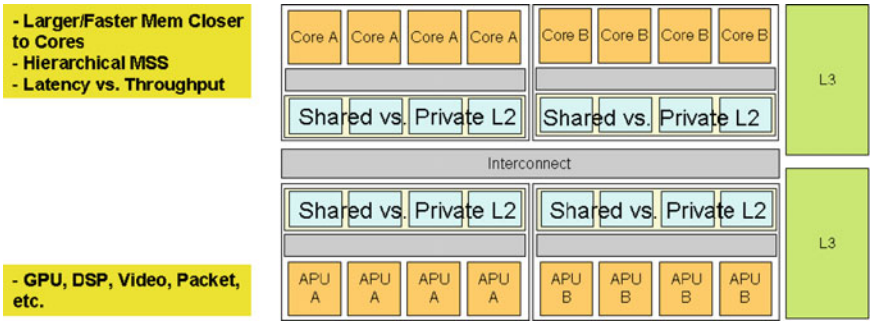


Fig. 9.12 Heterogeneous Manycore

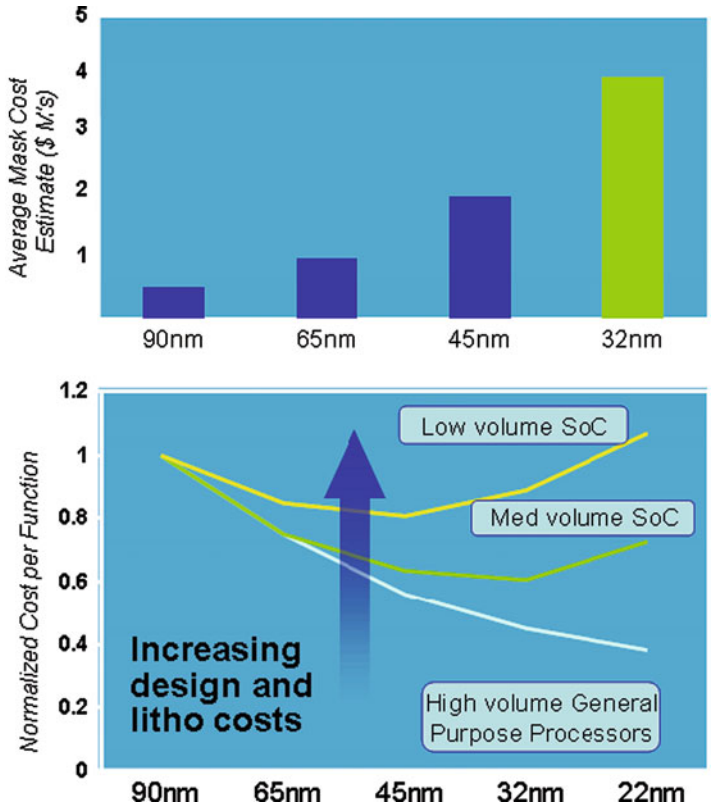


Fig. 9.13 The cost of scaling

9.10 Multicore Design: Challenges and Opportunities

Multicore design presents some major challenges, with associated opportunities. We discuss important examples of these below, including methods for meeting performance goals, standards-based multicore programming models, and advanced debugging and optimization techniques.

9.10.1 Meeting Performance Goals

Because of the complexities of heterogeneous multicore SoC designs and the cost of mask sets and manufacturing, performance verification before final tapeout is critical [13]. Presilicon performance verification focuses on ensuring that the system meets performance criteria for complex, real-world applications, in contrast

to verifying that models of the system can accurately predict performance, or that subsystems of the design meet latency or timing requirements, or that the manufactured parts meet performance criteria. Furthermore, formal techniques are not yet ready for efficient use by industry. For these reasons, we concentrate on performance verification using full-system applications that reasonably model the workloads used by customers, while meeting the constraints of presilicon environments.

Three factors affect a successful outcome: metrics, infrastructure, and methodology. Metrics set the goals of the effort and guide development of infrastructure and workloads. Infrastructure helps to manage complexities of running workloads, collecting metrics, and analyzing results. Methodology must support separation of concerns leading to a structured breakdown of the problem space with confidence in the data being produced. While these are important for functional verification, performance verification has critical differences. Careful coordination can reduce inefficiencies.

We have successfully used a performance methodology that is divided between bottom-up and top-down phases (Fig. 9.14). The bottom-up phase, consisting of micro-benchmarking using an Hardware Definition Language (HDL) simulator, is executed first to provide early exposure of negative performance indicators. This verifies acceptable best case performance of various system-level transaction types.

The top-down phase employs macro-benchmarks using hardware emulation. This phase explores whether resource contention will degrade system-level performance by measuring and tuning representative applications. Emulation is required to reasonably provide the number of cycles required for workloads to reach steady state for performance measurement.

Bottom-up workloads target memory subsystem latency and bandwidth. Therefore, multiple scenarios are executed for one to N cores reading small buffers of data using nonoverlapped strides. Each core is assigned a different memory region (for example, Core0: 0–8K, Core1 8K-16K, etc.) The core and off-chip memory are then configured using a predefined set of frequencies, ratios, etc. This allows engineers to execute multiple performance scenarios, for example one processor L3 hit, one processor L3 miss, eight processor L3 hit, eight processor L3 miss, etc. Three performance classes are targeted: *unloaded latency*, *single device throughput*, and *multi-device throughput*.

Unloaded latency stimulus involves a single requestor, a single target, and a single outstanding transaction. Scenarios may include various combinations of (a) requestors (i.e., core, various hardware accelerators, various I/O devices, etc.), (b) targets (cores, caches, memory, hardware accelerators, etc.), (c) target hit/miss

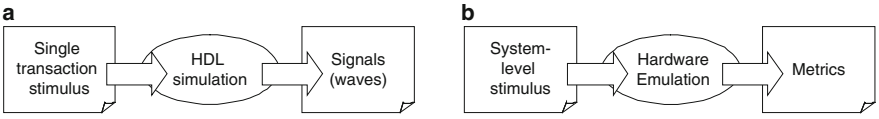


Fig. 9.14 High-level methodology; (**a**) Bottom-up, and (**b**) Top-down

types (memory page open, memory page closed, cache and directory hit/miss, cache conflict, etc.), (d) coherency attributes, and (e) operation type (read, write, etc.) Combinations of parameters are executed, with expected results provided as timing diagrams by the logic team.

Single-device throughput stimuli comprises transactions from a single requestor to a single or interleaved target. Transactions included reads of varying buffer sizes, with varying strides and different paging/interleaving. We visually inspect waves to establish a baseline. This verifies turnaround times, reveals unnecessary bus utilization gaps, and exposes needless extra cycles of latency.

Multidevice throughput stimuli are applied to all ports of a particular type (for example, all N cores). Addresses are varied to produce interference scenarios, such as memory paging and various arbitration algorithms. The goal is to identify bottlenecks and validate fairness as resources became saturated.

Executing the top-down methodology is challenging because system-level performance verification work typically must be done concurrently with functional verification activities. Furthermore, the functional verification methodology used for an SoC may not satisfy critical performance verification needs, such as how to partition and debug workloads and how to measure system responses.

Top-down workloads typically include core-to-memory latency/bandwidth as well as domain-specific applications such as network packet forwarding. The workloads must be partitioned for structured bring-up since they are significantly more complex than the bottom-up scenarios. The approach used is to first run stimulus only involving cores and memory, then test driver code to exercise core-to-accelerator communication, then single-core full system scenarios, and finally full system scenarios with multiple cores.

The top-down phase requires more complex testbenches, longer simulation times, and is exceptionally challenging for two additional reasons. First, it is important to ensure that top-down workloads are functionally correct before running them on emulation. Second, the workloads must be "emulator friendly."

Correct workloads are essential because debugging on emulation is difficult. This can be mitigated by using a functional simulator for workload development, although this cannot fully prevent latent timing and coherency issues in the workloads.

Emulation-friendly workloads must avoid file I/O, must not require an OS, and must be scalable to allow orderly bring-up and debug. Scalable versions must be created to (a) bypass or utilize hardware acceleration, (b) run on one, three, . . ., up to N cores, (c) allow configuration to employ features of the design or work around them in software (to make it easier to remove components that are not fully functionally verified), and (d) work with various memory size and location constraints.

Multicore SoC pre-silicon performance verification requires broad knowledge of complex integrated components and real-world workloads. Yet, the effort is well spent because everywhere a team looks they will find potential performance defects. Through such efforts, we often identify enhancements for future designs, and gain invaluable experience and insight into debugging and optimizing performance for our post-silicon customers. A fortunate side-effect is the discovery of many functional defects along the way.

There are many other side benefits of pre-silicon performance verification. For example, the ability to measure performance of the system allows for customer benchmarks to be run pre-silicon with high confidence, and it allows last minute confidence for design adjustments to make timing closure.

9.10.2 Standards-Based Programming Models

A programming model comprises a set of software technologies that allow programmers to express algorithms and map applications to underlying computing systems. Sequential programming models do so in terms of serial programming steps which occur in a strict order, with no notion of concurrency. Universities have taught this programming model for decades. Outside of a few specialized areas, such as distributed systems, scientific applications, and signal processing applications, the sequential programming model permeates the design and implementation of software [14–16].

But there is no widely accepted multicore programming model outside the possibility of those that were defined for symmetric shared-memory architectures exhibited by workstations and personal computers, or those that were designed for very specific embedded application domains, such as media processing [17, 18]. While these programming models work well for certain kinds of applications, many multicore applications cannot take advantage of them. Standards that require uniform shared memory are not always suitable because multicore systems display a wide variety of nonuniform architectures including combinations of general purpose cores, digital signal processors, and hardware accelerators. And domain-specific standards do not cover the breadth of application domains that multicore encompasses.

The lack of a flexible, general purpose multicore programming model limits the ability of programmers to transition from sequential programming to multicore programming. It forces companies to create custom software for their chips. It prevents toolchain providers from maximally leveraging their engineering, forcing them to repeatedly produce custom solutions. It requires end-users to craft custom infrastructure to support their programming needs. Furthermore, time-to-market pressures often force multicore solutions to target narrow vertical markets, which limits the viable market to fewer application domains and also prevents efficient reuse of software.

In SMP, multicore programmers can use existing standards, such as POSIX® threads (Pthreads) or OpenMP for their needs [18, 19]. In other contexts, the programmer may have the option of using existing standards for distributed or parallel computing, such as sockets, CORBA, or MPI [20, 21]. Yet, primarily due to two factors, there are many contexts in which these standards are unsuitable: (1) heterogeneity of hardware and software, and (2) constraints on code size and execution time overhead.

For heterogeneous multicore contexts, it is impractical to use any standard that makes an implicit assumption about underlying homogeneity. For example,

Pthreads is insufficient for interactions with offload engines, with cores that do not share a memory domain, or with cores that are not running a shared SMP OS instance. Furthermore, implementers of other standards such as OpenMP use Pthreads as an underlying Applications Programming Interface (API). Until more suitable and widely applicable multicore APIs become available, these layered standards will also have limited applicability.

Systems with heterogeneous cores, ISAs, and memory architectures have programming characteristics similar to those of distributed or scientific computing. A variety of standards exist for this system context including sockets, CORBA, and MPI. However, there are fundamental differences between interconnected computer systems (common in distributed and scientific computing) and multicore computers. This limits the scalability of these standards into the embedded world. At issue is the overhead required to support features that are not required in the multicore system context. For example, sockets are designed to support lossy packet transmission, which is unnecessary with reliable interconnect, CORBA requires data marshalling that may not be optimal between any particular set of communicators, and MPI defines a process/group model not always appropriate for heterogeneous systems.

These concerns justify a set of complementary multicore standards that will: (1) embrace both homogeneous and heterogeneous multicore hardware and software, (2) provide a widely applicable API suitable for application-level programming as well as for layering of higher level tools, (3) allow implementations to scale efficiently within embedded system contexts, and (4) not preclude use of other standards within a multicore system.

A roadmap for producing a generalized multicore programming model has been published by the companies working together in the Multicore Association (MCA) [22]. Because it is a fundamental capability for multicore programming, intercore communications was deemed a top priority feature of this roadmap. The consortium's working group completed the Multicore Communications API (MCAPI) specification in March 2008 [23].

The MCAPI specification defines three communication types (Fig. 9.15), as follows:

messages – connection-less datagrams
packets – connection-oriented, arbitrary size, unidirectional, FIFO streams
scalars – connection-oriented, fixed size, uni-directional, FIFO streams

Messages support flexible payloads and support dynamically changing receivers and priorities, incurring only a slight performance penalty in return for these features. *Packets* also support flexible payloads, but utilize connected channels to provide higher performance at the expense of slightly more setup code. *Scalars* are intended to be the highest performance, exploiting connected channels and a set of fixed payload sizes. For programming flexibility and performance opportunities, MCAPI messages and packets also support nonblocking sends and receives to allow for overlapping of communications and computations.

Communications in MCAPI occurs between *nodes*, which can be mapped to many entities, including but not limited to: a process, a thread, an instance of an OS,

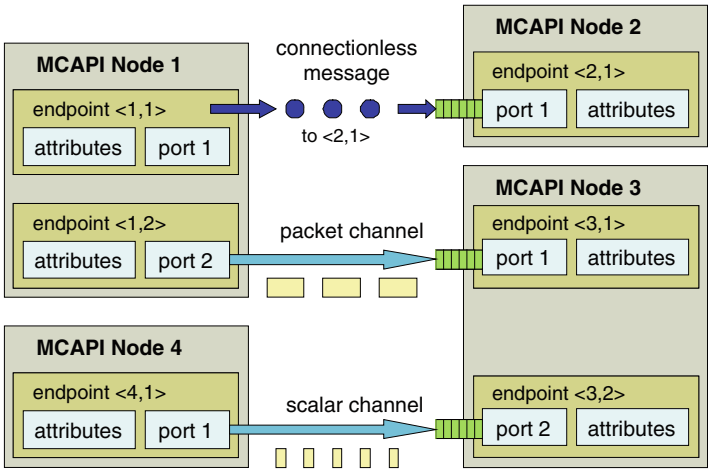


Fig. 9.15 The Three MCAPI communications types

a hardware accelerator, or a processor core. A given MCAPI implementation will specify what defines a node.

MCAPI nodes communicate via socket-like communication termination points called *endpoints*. These are identified by a topology-global unique identifier (a tuple of $<node, port>$). An MCAPI node can have multiple endpoints. MCAPI channels provide point-to-point FIFO connections between a pair of endpoints.

Additional features include the ability to test or wait for completion of a non-blocking communications operation, the ability to cancel in-progress operations, and support for buffer management, priorities, and back-pressure mechanisms to control the management of data between producers and consumers.

Scalable performance is a critical feature of any multicore programming model. Figure 9.16 illustrates performance of an MCAPI *echo* benchmark for 64 byte data size. This data was collected from a dual core Freescale evaluation system. The results are normalized to the performance of a Berkeley sockets-based version of *echo*. The other data series in the graph compare the sockets baseline to a unix pipes version of *echo* and an MCAPI message-based version of *echo*. For distances of eight or less hops, MCAPI outperforms both pipes and sockets. It is important to emphasize that these results were collected on a dual core processor with a user-level implementation of MCAPI. This means that sockets and pipes had the advantage of kernel support for task preemption on blocking calls, whereas the example MCAPI implementation uses polling. Despite these disadvantages, the example MCAPI implementation performed quite well, and we expect optimized versions to exhibit better performance characteristics.

The MCAPI specification does not complete the multicore programming model, but it provides an important piece. Programmers can find enough capability in MCAPI to implement significant multicore applications. We know of four

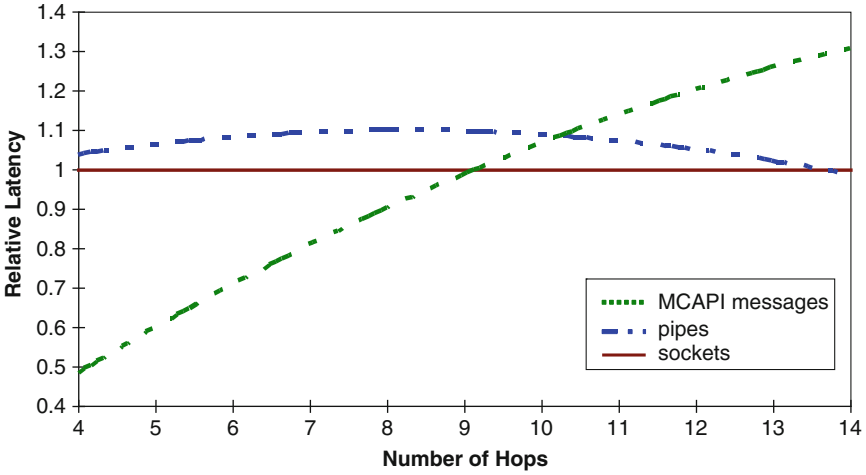


Fig. 9.16 Latencies for the MCAPI Echo benchmark

MCAPI implementations at the time of this writing, which should help foster adoption of the standard.

The MCA roadmap continues to be pursued by additional active workgroups, notably the Multicore Resource Management API (MRAPI), and Hypervisor working groups.

MRAPI will be a standard for synchronization primitives, memory management, and metadata. The desire is for MRAPI to be complementary in form and goals to MCAPI. While the targeted features are typically provided by OS, there is a need for a standard that can provide a unified API to these capabilities in a wider variety of multicore system contexts.

The Hypervisor working group is seeking to unify the software interface for paravirtualized OS. Such a standard would allow OS vendors to better support multiple hypervisors and thus more multicore chips with a faster time to market.

We believe that multicore programming model standards can provide a foundation for higher levels of functionality. For example, we can envision OpenMP residing atop a generalized multicore programming model, language extensions for C/C++ to express concurrency, and compilers that target multicore standards to implement that concurrency. Another natural evolution to programming models, languages, and compilers would be debug and optimization tools providing higher levels of abstraction than we have today. With MCAPI, it may be possible for creators of debug and optimization tools to begin considering how to exploit the standard.

9.10.3 Advanced Debugging and Optimization

As with the multicore programming model, no high level multicore debug standard exists. There are standards for the "plumbing" that is required in hardware to make

multicore debug work; and debug and optimization tools exist for multicore workstations and personal computers. Toolchain providers have created multicore debug and optimization software tools, but these have been custom tailored to each specific chip. This situation leaves many programmers lacking tool support.

We learned decades ago that source-level debugging tied to the programming language provided efficient debugging techniques to the programmer, such as visual examination of complex data structures and stepping through stack backtraces. Raising the abstraction of multicore debugging is another natural evolutionary step. In a multicore system context, the programmer creates a set of software tasks which are then allocated to the various processor cores in the system. The programmer would benefit from monitoring task lifecycles and from seeing how tasks interact via communications and resource sharing. However, the lack of standards means that debug and optimization for most multicore chips is an art, not a science. This is especially troubling because concurrent programming introduces new functional and performance pitfalls including deadlocks, race conditions, false sharing, unbalanced task schedules, and many more.

9.11 Conclusions

The vision of "networking where every connection matters" is the driving force behind this next generation MCP, setting a new performance standard by exploiting parallelism through multiple Out-of-Order Superscalar Power Architecture™ cores with Hypervisor support, multiple application-specific accelerators, innovative memory hierarchies, and advanced interconnect via the CoreNet coherency fabric. In this chapter, several key considerations for the design were discussed as well as the opportunities that multicore System-on-Chip provided. This MCP is clearly designed for reliability, security, scalability, and broad bandwidth with virtualized resources supporting concurrent activities.

Multicore is being driven by the power wall and by ever increasing demands for computational power; these computational demands come especially from emerging application domains that (1) exploit interactivity and connectivity and (2) make the world a safer place by augmenting our ability to manage complexity in a fast-paced environment. These trends unfortunately expose end-users to security and privacy risks, thus requiring additional bandwidth and computational power to mitigate their exposure.

Successful multicore SoC chips must allow end users to leverage the benefits of performance scaling through increased concurrency and bandwidth, but this must not come at the expense of power. Thus, designers must pay careful attention to system designs that make appropriate power/performance tradeoffs; additional techniques that must be leveraged are appropriate technology scaling and high degrees of system integration with specialized hardware acceleration units and numerous I/O interfaces. This introduces complexity in managing shared system resources, and we mitigate this with virtualization techniques. In short, multicore

SoC has a number of intrinsic characteristics which both distinguishes them from previous generations of chips and enables them to provide new levels of system performance.

Finally, with the increased integration and parallel compute power of Multicore SoC comes an increased need for tools to aid in programming, debugging, and optimizing software. We believe emerging standards for multicore software will mitigate some of this need.

References

1. Creeger, M., *Multicore CPUs for the Masses*. ACM Queue, 2005. **3**(7): p. 63–64
2. Donald, J., Martonosi, M., *Techniques for Multicore Thermal Management: Classification and New Exploration*. In *33rd International Symposium on Computer Architecture*. 2006
3. Geer, D., *Chip Makers Turn to Multicore Processors*. IEEE Computer, 2005. **38**(5): p. 11–13
4. Cisco. *Hyperconnectivity and the Approaching Zetabyte Era*. 2009 Available from: http://www.mycisco.biz/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/VNI_-Hyperconnectivity_WP.html
5. Bell, S., et al. *TILE64 Processor: A 64-Core SoC with Mesh Interconnect*. in *International Solid-State Circuits Conference*. 2008
6. Freescale Semiconductor, I. *P4080 Product Summary Page*. 2008 Available from: www.freescale.com/files/netcomm/doc/fact_sheet/QorIQ_P4080.pdf
7. Intel. *Next Generation Intel Architecture - Nehalem*. 2008 Available from: http://www.intel.com/technology/architecture-silicon/next-gen/index.htm
8. Pham, D.C., et al., *Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor*. IEEE Journal of Solid-State Circuits, 2006. **41**(1): p. 179–196
9. Holt, J., et al., *Software Standards for the Multicore Era*. IEEE Micro, 2009a. **29**(3): p. 40–51
10. Dally, W., Towels, B., Principles and Practices of Interconnection Networks, Morgan Kaufman, CA. 2004
11. Diato, J., Yalamanchili, S., Ni, L., Interconnection Networks, Morgan Kaufmann, CA. 1993
12. Deshpande, S.R., Interconnections for Multi-core Systems; Embedded Systems Conference, April 2008
13. Holt, J., et al. *System-level Performance Verification of Multicore Systems-on-Chip*. in *IEEE Workshop on Microprocessor Test and Verification*. 2009b
14. Bridges, M.J., et al., *Revisiting the Sequential Programming Model for Multi-core*. IEEE Micro, 2008. **28**(1): p. 12–20
15. Hwu, W.-m.W., Keutzer, K., Mattson, T.G., *The Concurrency Challenge*. IEEE Design and Test of Computers, 2008. **25**(4): p. 312–320
16. McCool, M.D., *Scalable Programming Models for Massively Multicore Models*. Proceedings of the IEEE, 2008. **96**(5): p. 816–831
17. The Khronos Group. *Open Standards for Media Authoring and Acceleration*. 2008 Available from: http://www.khronos.org/
18. The Open Group. *The Open Group Base Specifications Issue 6*. 2008 Available from: http://www.opengroup.org/onlinepubs/009695399/
19. OpenMP.org. *The OpenMP API specification for parallel programming* 2008 Available from: http://openmp.org/wp/
20. The MPI Forum. *MPI v2.1*. 2008 Available from: http://www.mpi-forum.org/
21. The Object Management Group. *CORBA 3.1 Specification*. 2008 Available from: http://www.omg.org/spec/CORBA/3.1/

22. The Multicore Association. *The Multicore Association Roadmap*. 2008a Available from: http://www.multicore-association.org/home.php
23. The Multicore Association. *Multicore Communications API Specification V1.065*. 2008b Available from: http://www.multicore-association.org/workgroup/comapi.php

Chapter 10
High-Performance Multiprocessor System on Chip: Trends in Chip Architecture for the Mass Market

Rob Aitken, Krisztian Flautner, and John Goodacre

10.1 Introduction

10.1.1 Mass Markets and High Performance

The proliferation of embedded processors in recent years is astonishing. The most obvious example is mobile phones, with annual sales of over a billion units. Such volumes clearly form a "mass market," but other businesses also involve huge unit processor volumes as well. These include microcontrollers, enterprise (e.g., disk drive controllers), home entertainment (HDTV), automotive, and more. Historically, most of these have been single processor systems, or collections of single processor subsystems, rather than true multiprocessor systems, and as such are not of direct relevance to the topic of this book, but the same trends that are driving the move to multiprocessing in other domains are at work in these mass markets as well. To see why, let us look in more detail at the example of a mobile phone.

Classically, mobile phones have consisted of three main functions: radio communication, user interface, and digital processing. The digital processing portion is usually confined to a single chip, with additional memory around it. Cost, size, and reliability all drive a single chip digital processing solution. However, as circuit density has increased, the amount of functionality achievable with a single chip has also risen. So while a 1998 vintage phone (Fig. 10.1) was limited to simple number lookup and text messaging using its digital chip, a 2008 generation smart phone is able to perform numerous other functions, including email, web surfing, video player and camera, music player, digital camera, video game player, mapping/GPS, and more.

Traditionally, this added functionality has been achieved in three ways: (1) silicon scaling; (2) increasing microarchitecture complexity to extract instruction level parallelism (ILP); and (3) adding more cache to reduce the effect of off-chip access. CMOS scaling historically provides for lower power consumption and

R. Aitken (✉)
ARM Inc., San Jose, CA, USA
e-mail: rob.aitken@arm.com

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_10,

Fig. 10.1 Mobile phones from 1998 (*top*) and 2008 (*bottom*). Source: Nokia

increasing frequency to provide more performance, while reducing area. Because the goal of a single digital processing System-on-Chip (SoC) had already been achieved, scaling was also used to provide for increased functionality while maintaining area and power, leading to more complex SoCs, with embedded graphics and video, for example. Recently, scaling has run into trouble, however, leading to a situation where increasing frequency past a design-optimal point has significant and exponential effect on power, increased microarchitecture complexity is failing to extract more ILP and exponentially increases power and area, typical embedded software working-set can fit within on-chip caches, and, finally, process geometry reduction is failing to provide expected power benefits.

Just as power issues drove high-performance wired CPUs to multicore solutions, the need to use energy wisely is driving high-performance mobile processing to multicore. A quick scan of current mobile application processors shows that many already include multiple processors, including the TI OMAP 44x, Qualcomm Snapdragon, nVidia Tegra, and Samsung S3C6400. Figure 10.2 shows one example. More such designs are likely in the future, and new technologies, such as 3D-IC, inherently support multicore approaches.

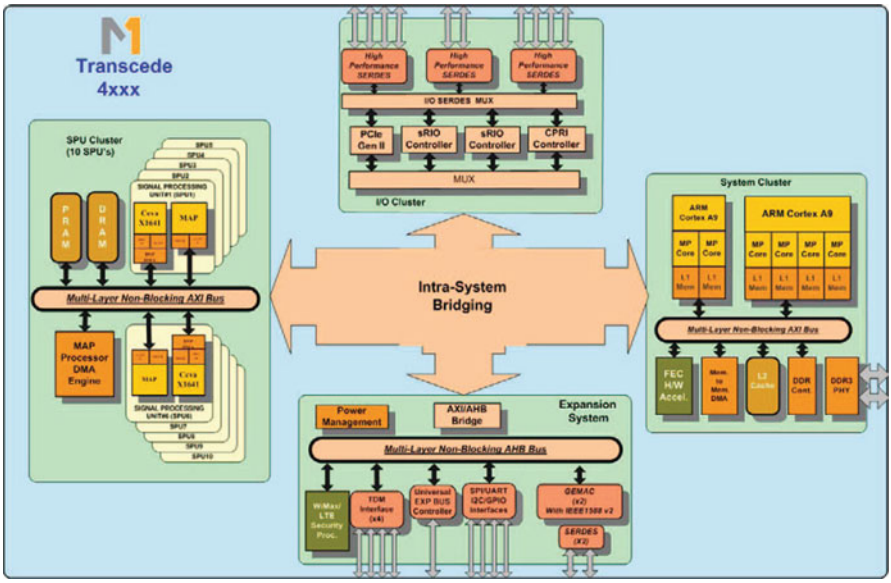


Fig. 10.2 Example Multicore baseband system. Source: Mindspeed

In fact, it is possible to view an entire SoC as a heterogeneous multicore processing system. The graphics and video processing units complement the applications processor, similarly, audio codecs are power management units are a form of specialized processor (Fig. 10.3).

10.2 Scaling and Consumer Expectations

Moore's law follows from CMOS device scaling, but the phenomenon is not strictly technological. Consumer expectations have become tightly linked to scaling, as every new device does more than its predecessor. Consider two handheld video games from Nintendo, the GameBoy and the DSi (see Fig. 10.4). When it was introduced in 1989, the GameBoy was an innovative toy. It was based around a Z80 CPU running at 1.05 MHz. It featured a 160 $\times$ 144 pixel black and white LCD screen and allowed up to four players to play together with a serial cable. On the other hand, the DSi, introduced 20 years later in 2009, features two CPUs: an ARM9 running at 133 MHz and an ARM7 running at 33 MHz. It has two screens, each 256 $\times$ 192 pixel color LCDs, as well as AAC audio, two VGA cameras, 256 Mb of flash memory and Wifi connectivity including a browser.

Clearly, the DSi is a substantially more powerful device, with at least 1,000$\times$ the computing power of the original GameBoy. The DSi was introduced at a price point of \$169. The GameBoy, in 2009 dollars, was essentially the same price. This trend is not uncommon in technology. PC prices have been declining in recent years in real

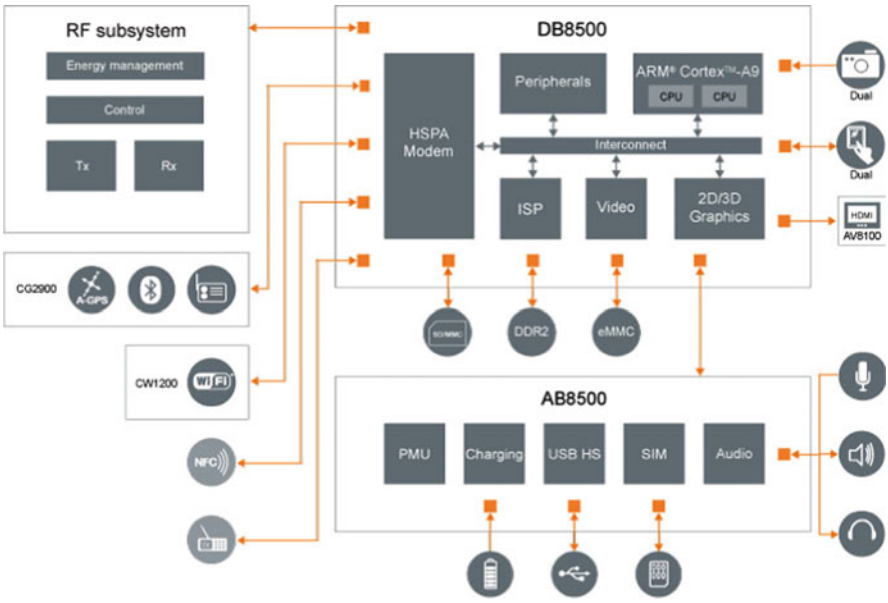


Fig. 10.3 Example Multicore mobile system. Source: ST Ericsson

dollars, even as their capabilities increase. Similarly, the trend in portable music players is to increase storage capacity while retaining a largely fixed price point.

Collectively, these observations point to a larger truth. Consumers have come to expect continuing technology improvements in performance and capability, and are unwilling to pay extra for these, and will even demand price decreases if advances are not fast enough. Competitive pressures within the electronics industry have certainly contributed to this state of affairs, but it is ingrained within consumer expectations as well.

10.2.1 Scaling Limitations

The basics of CMOS scaling, as outlined in the classic paper by Bob Dennard of IBM, are as follows:

Parameter (Scale factor = α)	Dennard scaling	Scaling now
Dimensions	$1/\alpha$	$\sim 1/\alpha$
Oxide thickness	$1/\alpha$	1
Voltage	$1/\alpha$	1
Drive current	$1/\alpha$	$1/\alpha$
Capacitance	$1/\alpha$	$1/\alpha < C < 1$
Power/circuit	$1/\alpha^2$	$1/\alpha$
Power density	1	α
Delay/circuit	$1/\alpha$	~ 1

Fig. 10.4 *Top*: GameBoy original (1989), *bottom*: GameBoy DSi (2009). Source: Nintendo

This scaling served the semiconductor industry well for nearly 30 years, but physical limitations have slowed conventional scaling. First of all, voltage scaling slowed and stopped at around 1V. As can be seen in the table, not scaling voltage stops delay scaling. It also lowers power dissipation per circuit, which results in power density increasing, rather than staying constant. Oxide thickness was next to slow down – as oxides shrunk to single digits of atoms thick, additional shrinking was simply not possible. However, economic pressure to continue scaling did not abate, and so alternative approaches were required.

Some of these are new materials. For example, high-k metal gate (HKMG) transistors provide for the benefits previously obtained by scaling gate oxide even when the oxide does not scale. Other such techniques include providing for multiple dopant concentrations and strain engineering in the silicon. See [Bohr 2009] for more information.

Additionally, even though device dimensions continue to scale, the mechanics of doing so has become increasingly complex. The wavelength of light used for photolithography has stopped scaling at 193 nm even as feature sizes continue to shrink. Printing at subwavelength dimensions is complex (it can be thought of as trying to draw lines significantly narrower than the width of the pen tip being used) and expensive. Resolution enhancement technologies (RETs) now include adjustments to masks, light sources, exposures, and even the light transmission medium (which has changed from air to water to improve optical capability). The added complexity of the photolithography and mask making process has led to more complex design rules, meaning that the simple pattern shrinking envisaged by Dennard has been impossible since the 90 nm silicon generation (∼2002).

Even with all of the technical fixes being used in the process of fabricating circuits, consumer demands for scaling can no longer be achieved without design changes. Significant standard cell and memory architecture changes are made at each process node to squeeze additional performance, area, or power out of devices that are increasingly reluctant to provide the gains. The additional complexity is reflected in the increasing cost of developing each node.

Finally, shrinking dimensions, increasing numbers of devices, and processes that increasingly bump up against physical walls has led to increasing concern about variability. Once solely a concern of analog designers, device variability has hit mainstream digital design as well. The interested reader is referred to [Bernstein 2006] for more details, but variability is now omnipresent: individual transistors vary in their dimensions and performance, within chips and among chips, by 10% in dimension and 50% in performance. Wire capacitances and resistances can easily vary by 30% or more between chips and even between adjacent metal layers. On-chip variation in voltage and temperature can further exacerbate these problems. The standard solution to this variation is to add margin to a design, which further limits its ability to scale. High-performance design teams need to quantify and account for margins to get the gains that design teams 15 years ago could achieve simply by switching to a new process.

10.3 CPU Trends

Until very recently, the differentiating factor in desktop processor design was simple; speed. Companies such as Intel and AMD were single-minded in their approach to processor design, with determined to both develop and release higher frequency processors before the other.

The race to release the world's first GHz processor was hotly contested with AMD emerging as the eventual winner. During this time both organizations were focused on their quest and slowly the industry became aware of the increased hardware complexity associated with higher MHz processors. The industry also realized that the MHz-only route could not go on indefinitely and other approaches were needed. In addition to improvement in processor efficiency, raising total performance through supporting thread-level parallelism presented themselves through Multiprocessor (MP) and Multithreading (MT) technologies.

Intel introduced an MT technology known as "Hyper-Threading," while AMD positioned themselves for what clearly became the dual core race, with both seeking to be the first to offer a true MP solution to the home computing market. What has caused this paradigm shift from two prominent semiconductor companies toward MP?

More recently, this shift to multiprocessing is imposing many of the software paradigms growing in popularity on the desktop also toward embedded designs. For many years, embedded designers have leveraged the advantage that by including multiple processors in their design, they can better provide the required computational performance within their limited power budgets. The real change now affecting the embedded market is that the application software is also being asked to view the general purpose processor element using a multiprocessing paradigm so that this processor can also benefit from the promises of higher performance and low power. Although MP and MT both assert this multiprocessing complexity to the software developer, all is not equal when the costs and complexity tradeoffs between the two are considered. As a result, MP systems are increasingly common. [Goodacre 2006].

10.3.1 Power

When designing a high performance SoC for a high volume market, power is invariably a key consideration. This is obviously the case for battery powered applications, but applies in wireline products as well: excess power consumption creates excess heat, and this in turn requires expensive cooling solutions. Adding a fan to a $29 printer is an expensive proposition, after all.

Let us consider another specific example: a pair of Nokia phones separated by a decade (Fig. 10.6). In 1998, the Nokia 5110 was a reasonably advanced mobile phone. Among its features were a 47 × 84 B/W display, 64 K RAM, 1 MB Flash, 16 buttons, and built-in entertainment such as the "Snake" game. All of this was powered by a 900 mAh battery. Ten years later, the Nokia N96 smart phone was significantly more powerful, featuring a 240 × 320 24-bit color display, 256 MB RAM, 16 GB Flash, touch screen input, a 5-megapixel camera (480p encode), 2D/3D graphics acceleration, stereo speakers with 3D audio and significantly more impressive video games than Snake. However, all of this was powered by a 950 mAh battery – barely 5% better than a decade earlier, but expected to provide energy to a significantly more powerful device.

Again, this exemplifies a common trend. Battery technology is not advancing nearly as quickly as performance or memory capacity – there is no Moore's Law for batteries. As a result, power (or more precisely, energy) management remains an essential driver for high performance mobile devices, and this in turn has led to a need for multiprocessor solutions.

Desktop processor scaling switched from single processors to multiprocessors starting in around 2005 in part because the energy dissipation trend was unsustainable (Fig. 10.5). To see why, consider the energy equation below

$$E = C V_{dd} f_{dt} + V_{dd} i_{leak}.$$

Increasing frequency directly increases energy consumption. In addition, the transistors needed to produce the extra frequency also climbs close to exponentially, leading to increased capacitance and increased leakage. Adding processors allows more tasks to be performed at a lower frequency, while avoiding the exponential transistor count increase and leading to reduced capacitance and leakage, even considering the extra processor.

10.3.2 Dark Silicon

The combination of challenging but feasible scaling in area and performance together with reduced or lack of scaling in power leads to an interesting situation

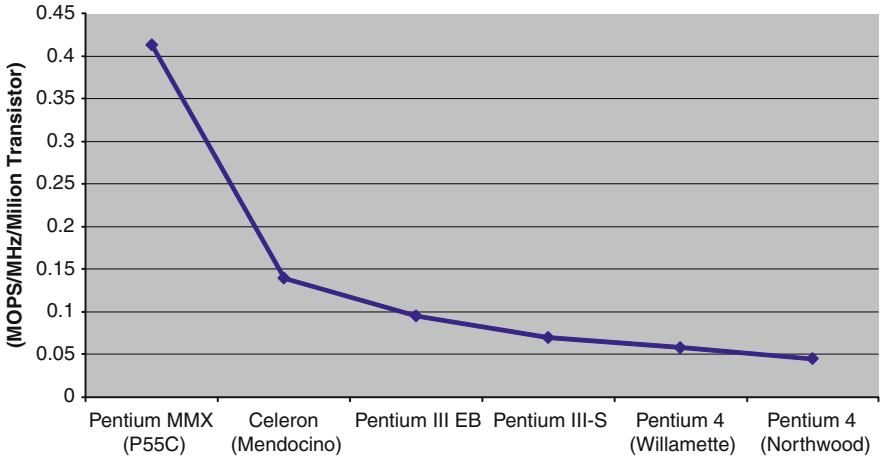


Fig. 10.5 Computation Density of Processors (Source: Wawrzynek et al, BWRC Retreat, Jan 2004)

Fig. 10.6 Phone comparison 1998 versus 2008

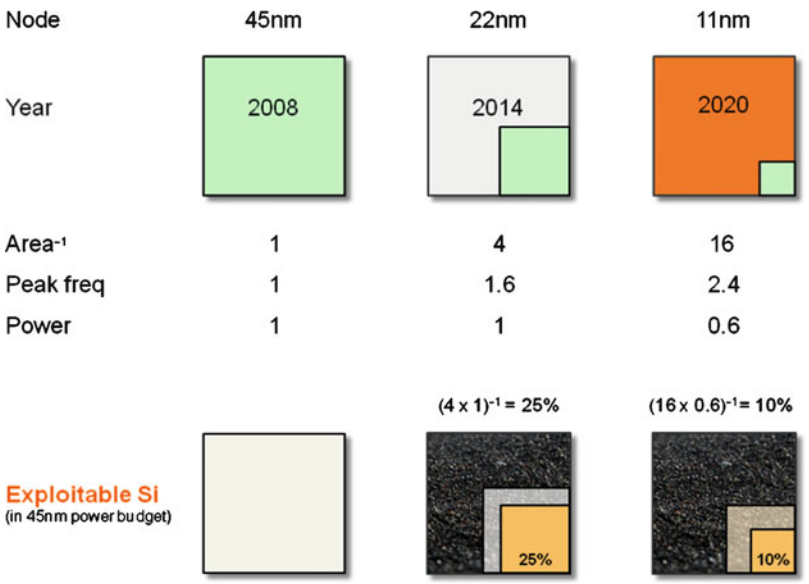


Fig. 10.7 Dark Silicon

of unexploitable silicon area, or what has been referred to as "dark silicon." To see how this works, consider the following example (shown graphically in Fig. 10.7).

Consider a design implemented at the 45 nm node with area A_{45}, frequency F_{45}, and power P_{45}. Scaling factors between process generations are open to some

debate, but we will use the ones from the ITRS roadmap for purposes of this discussion. Scaling this design through two generations to the 22 nm node results in essentially Dennard scaling of area and some frequency improvement, giving

$$A_{22} = 0.25 * A_{45}$$

$$F_{22} = 1.6 * F_{45}.$$

There are two possibilities for power. If frequency is fixed, then we have

$$P_{22,\text{fixed}} = 0.6 * P_{45}.$$

On the other hand, if the scaled frequency is used, this results in

$$P_{22,\text{fast}} = P_{45}.$$

As we have discussed already, power is often the key limitation for a given design. On the other hand, in previous generations it has usually been desirable to keep overall die area roughly constant and use scaling to offer expanded functionality with the "extra" silicon. This becomes challenging as the design is scaled beyond 45 nm – if the power for the 22 nm design is meant to be the same as the 45 nm design, the exploitable chip area is much less than it was originally. In the "fast" case, where frequency is scaled, only 25% of the 45 nm area is available (no functionality change). In the constant frequency case, the situation is somewhat improved, but still only 42% of the original area is available (an additional 67% functionality could be added).

The situation is expected to get worse over time. As scaling continues to the 11 nm node, the equations become

$$A_{11} = 0.07 * A_{45}$$

$$F_{11} = 2.4 * F_{45},$$

which again lead to two power situations:

$$P_{11,\text{fixed}} = 0.3 * P_{45}$$

$$P_{11,\text{fast}} = 0.6 * P_{45}.$$

In this case, the resulting usable areas are 23% for the fixed frequency case and only 11% for the scaled frequency case. The functionality improvements are significant (230% and 67% respectively), but still leave a lot of unused area.

This utilization challenge means that there will be plenty of room in a fixed area design for added functionality, but there will be no way to power it up, hence the

term "dark" silicon. The question of what to do with or about this dark silicon is key to the discussion of high volume, high performance multiprocessor systems.

10.3.3 What to do with Dark Silicon

Power scaling has lagged other types of scaling for a few generations, so the dark silicon problem is not entirely new. In the past, three approaches to using extra silicon area have been commonly used in high performance, low power SoCs.

1. Add more memory
2. Shrink the die
3. Change the power equation

The first of these, adding more memory, has been most prevalent. Increasing cache size has been common, but as was noted earlier, no longer provides performance improvement once the standard application run set can be located entirely within the cache. Furthermore, it is no longer feasible to ignore SRAM leakage when considering overall chip power, so adding memory is not free from a power perspective.

Shrinking the die has also been common. This provides for cost savings, but care must be taken to ensure that the design does not become I/O bound (the number of I/Os available increases only linearly with scaling, while the number of transistors increases quadratically). Also, consumer expectations require ever-increasing functionality and ignoring those expectations can result in failing products, or products that must compete in a commodity market and are unable to attract any sort of price premium.

The third approach is conceptually the best. By changing the rules of the design, it is possible to change the overall power consumption. A simple example is clock gating. By restricting clocking to only those portions of a design actively involved in an operation, dynamic power can be dramatically reduced. Similarly, reducing the leakage of noncritical gates by using higher VT transistors or longer gate lengths is another common "rule changing" approach that adds extra power reduction beyond scaling. Voltage scaling, frequency scaling, and combinations of the two also reduce power. These can be static, dynamic, or adaptive in nature. Also, power gating can be used to shut off large blocks of unused circuitry. For more details on these methods, please see Keating et al [Keating 2007].

As we continue, it is important to remember that there is more to a system on chip than the processor, but processing elements tend to be where performance and power constraints are the greatest. There is no shortage of challenges in modern SoC design, but we will concentrate on power.

While all systems-on-chip are different, there are some common components that we can consider to be part of a "typical" SoC. These include a CPU subsystem, containing one or more processors, and local cache. These are connected through a

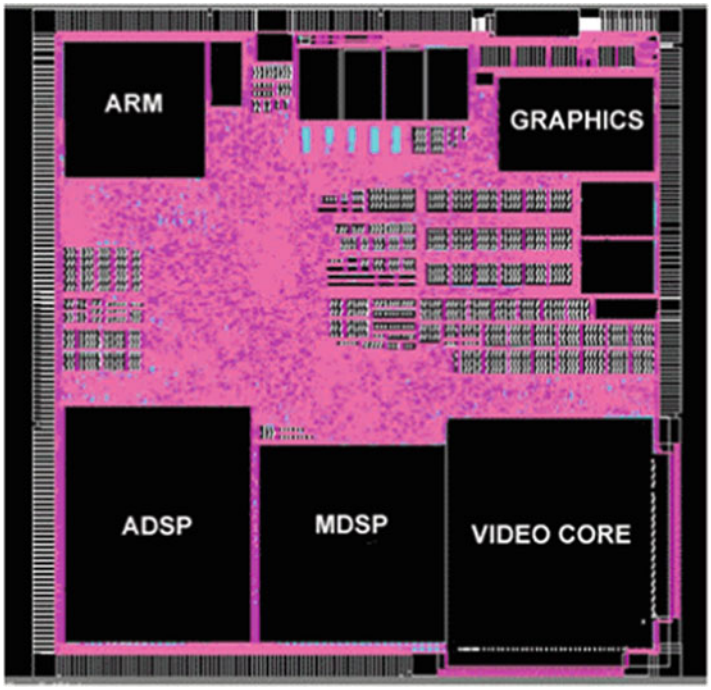


Fig. 10.8 Typical System-on-Chip

bus to other blocks, such as a graphics unit, a video system, additional system memory, mixed signal components, off-chip interfaces to DRAM, Flash, etc., as well as other components that are often lumped together as "random logic." A typical example is shown in Fig. 10.8.

The trends discussed previously also apply to these other SoC components. For example, all processing elements, whether CPU, GPU, or specialized processors, face scaling challenges in terms of processing performance and power consumption. Similarly, SRAM-based memory must confront leakage issues and minimum operating voltage difficulties. Lower-performance logic must balance area and leakage while not missing performance targets.

Expanding such a system to a multiprocessor adds new challenges.

Multiprocessing essentially uses a "divide and conquer" approach using modular design principles where a single (multi)processor is created by bringing multiple processing units together each capable of running a separate concurrent thread. Ideally, a multiprocessor architecture should enable a "plug-and-play" solution, where systems designers can simply plug in additional processors as needed, rather than using a complex multithreading approach.

In addition to the hardware-focused discussion here, software and operating system compatibility need to be considered. Additional discussion is available at in [Adve 1996 and Patterson 2009].

A challenge that is often placed against multiprocessing and the duplication of L1 across each of the processors is that for shared data held by one of the other processors, software needs to actively compensate and take care regards the additional access latencies and penalties. In traditional SMP multichip designs, where any chip to chip sharing of data passed across a much slower board level backplane, this was true. However, when MP is on chip, these cost quickly become negligible. For example, an on-chip ARM-based system can resolve a cache miss, or access to shared data around 60% faster than a processor could otherwise resolve data from a shared Level 2 cache [Goodacre 2006].

Once the decision is made to go with a multiprocessor system, the question arises "how many cores are needed"? Dual core systems are now the norm in desktop systems, quad core and more are increasingly common.

The complexities of scaling and choosing a multiprocessor architecture mean that Moore's law goal becomes a question of achieving the highest performance with limited power at given cost. Beyond a certain performance level, this can only be achieved with a multiprocessor, as shown by Fig. 10.5 on the computation density of processors. It is simply impractical to push performance at reasonable power cost without increasing the number of cores. The serial instruction stream limits parallelism, and power consumption limits performance. The result in high-volume mobile computing space is multiprocessor systems such as that shown in Fig. 10.9.

So the question becomes, where does this lead? One problem in answering the question is that it seems that everything depends on everything else. As an example, consider the challenge of reducing leakage power. A partial list of things influencing leakage includes:

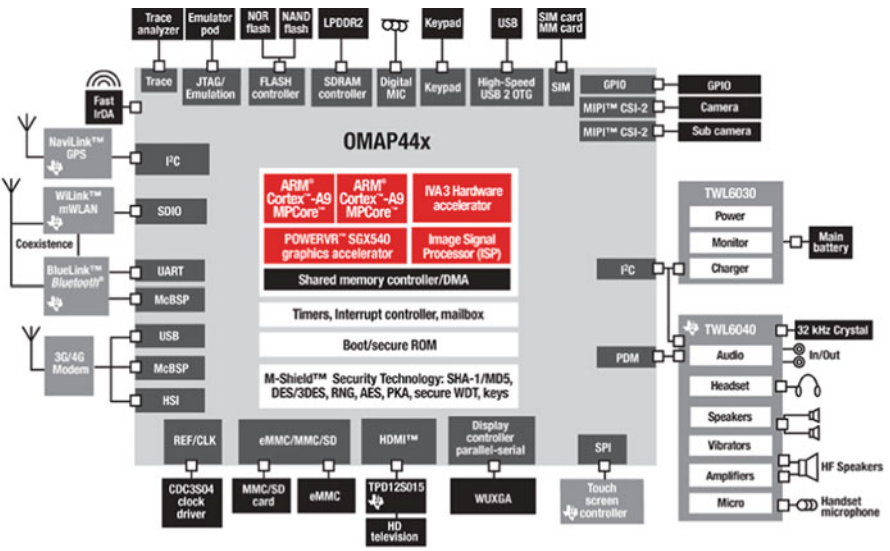


Fig. 10.9 Multicore System Today (Courtesy Texas Instruments)

- Operating System with Software Policies
  - Managing the entry and exit to and from system sleep states
- Power Supply Management
  - External power supply control, power supply tolerances, etc.
- System Level Control IP
  - Architectural design partitioning, hardware control
  - Sleep transition protocol management
- Library Level Support
  - Low power cells (e.g., level shifters, gating cells, retention flops)
  - Low power memory (architecture, sleep states)
- EDA Software
  - Support for chosen techniques
- Process Technology
  - Tradeoff between high performance and low leakage process nodes

So what is to be done? Clearly, a systems approach is called for. For example, a combination of a low power process, together with a multivoltage setup where low activity circuits always run at reduced voltage is one solution. In another approach, it might be desirable to include operating system features that enable sleep states, controlled by carefully placed power gates around modules, and architectural features that support known entry into and exit from sleep states. Any number of other approaches are possible, but they must be considered as part of an overall system, rather than simply adopting techniques piecemeal.

This same approach extends to other aspects of multiprocessor SoCs. As additional performance is needed, it is not enough to simply have more powerful "smarter" processors, since single CPUs run out of gas for power/performance reasons. Similarly, adding more CPUs is not in an itself a solution either. The costs of additional cores need to be justified by additional benefits, and this requires consideration of the memory system, interconnect, power delivery system, and all levels of software. As a result, integration is key to system design. Since not all integration approaches are equal, architecture becomes a question of integrating the right stuff the right way.

As an example, consider memory bandwidth for a display system. It can be seen from Fig. 10.10. that the Moore's law scaling trends affect display bandwidth, just as they do other aspects of SoC. To develop an architecture, it is first necessary to analyze traffic patterns. This involves identifying the bandwidths from the masters. This in turn drives the selection of DRAM technology and architecture and determines the average latency in the system. The next step is to identify the latency tolerance characteristics of the masters. This drives the arbitration policy – including "time-out" characteristics. Next, it is necessary to determine the interconnect structure. A hierarchical structure has the advantage of allowing each interface speed to be selected to support peak bandwidth for master or slave, taking into account latency and throughput considerations. Further, hierarchical interconnect uses traffic combination to minimize interconnect size, which enables scaling of complexity (e.g., power and/size). The result is a system that allows the balancing of pipeline

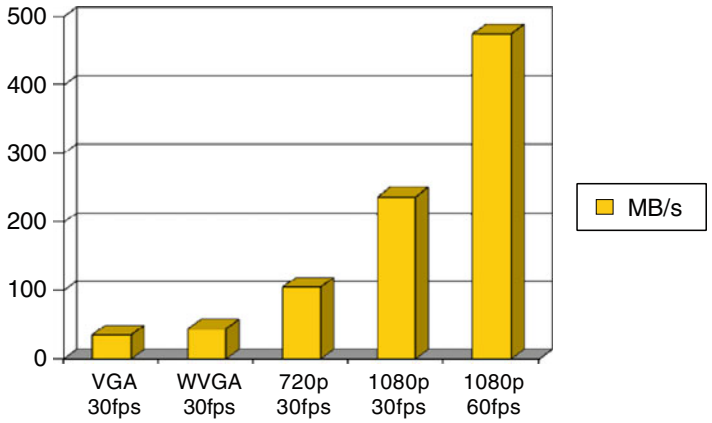


Fig. 10.10 Display bandwidth requirements

latency with synthesis frequency and gate-count, and may be considered to be a form of Network-on-Chip (NoC) that is independent of the interface protocol.

Similar approaches may be used for the remainder of the SoC. This brings us back to trends. So what will happen with regards to scaling?

While there is always concern that some challenge on the scaling roadmap will prove insurmountable, it is reasonable to expect that device scaling will continue uninterrupted below the 20 nm node as the semiconductor industry works to feed customer demand for more and more in the way of electronic products.

Nonetheless, SoC design will take place in an increasingly constrained environment: Power limitations require clever approaches (e.g., power gating, adaptive voltage scaling, etc.). Standardization of interfaces, sub-blocks, and even micro-architectures is increasingly needed to limit costs, and proliferate solutions widely. At the same time, differentiation is needed for visibility, branding. In every case, high volume is needed to justify both customization and the high fixed costs involved in SoC development.

For an IP provider, the keys to serving this market are flexibility, innovation, and establishing community. For designers limited by scaling, one avenue to solutions lies with pushing scaling by changing the rules. For example, if the rules require a constant voltage per chip, choose an adaptive approach (environmentally aware power), where voltage is set per chip and adjusted as the environment or workload changes. Alternately, if the rules mandate that computation never produce errors, choose an approach with error tolerance (e.g., numerically insignificant errors are allowed [Breuer 2008]) or with speculative execution (e.g., Razor [Das 2006], where errors may occur, but the design is self-monitoring and able to recover and redo the computation in an error-free fashion. Or, if the rules require DRAM for volatile storage and Flash for nonvolatile, adopt an emerging memory (e.g., high performance nonvolatile such as STT-MRAM [Huai 2008]).

Let us look in detail at one of these examples, the Razor approach. As shown in Fig. 10.11. conventional voltage and frequency scaling is limited by a variety of

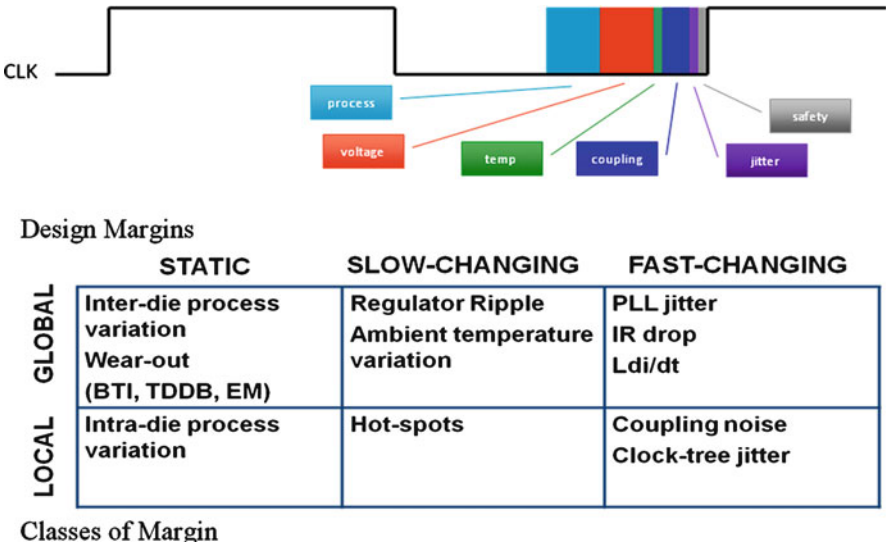


Fig. 10.11 Classes of Margin

margins designed to ensure that the system never produces an error. However, power efficiency (energy per operation) continues to improve past the point where errors occur, and the conditions that limit error-free high frequency operation do not occur with every operation. If a design is sufficiently self-aware to recognize when an error occurs, and is able to rapidly reconfigure itself into a safe operating mode, both voltage and frequency can be pushed into realms beyond the margin region that improve energy per operation without sacrificing error-free operation. The result is a design that is able to adapt itself to slowly changing conditions (temperature, global workload) while surviving rapidly changing issues (instantaneous IR drop, clock jitter, etc.). For more details, see [Bull 2010]

10.4 Conclusion

As we look to trends, it is clear that the future will contain more multicore systems. For now these are confined to higher performance mobile and stationary systems, but, if history is any guide, future low cost systems will look increasingly like today's high performance versions. What will these systems contain? If they follow current trends, they will feature improved cache architectures, with performance optimized interconnect fabrics. The chips will include heterogeneous processing elements (CPU, GPU, video, etc.) and will incorporate in-package DRAM and flash, possibly using 3D-IC techniques. However, it is also likely that winning solutions will change the rules, incorporating speculative execution (e.g., Razor), fault/error tolerance, or other innovative approaches that allow the performance/power/yield equation to be altered.

References

S. Adve and K. Gharachorloo, “Shared Memory Consistency Models, A Tutorial”. IEEE Computer, Vol. 29, No. 12, pp. 66–76, Dec. 1996.

K. Bernstein et al., “High-performance CMOS variability in the 65-nm regime and beyond”, IBM Journal of Research and Development, Vol. 50, pp. 433–449, July 2006.

M. Bohr, “The New Era of Scaling in an SoC World”, Proc. Int. Solid State Circuits Conf., pp. 22–28, Feb. 2009.

M. Breuer and H. Zhu, “An Illustrated Methodology for Analysis of Error Tolerance”, IEEE Design and Test of Computers, Vol. 25, No. 2, pp. 168–177, March-April 2008.

D. Bull et al., “A Power-Efficient 32b ARM ISA Processor Using Timing-Error Detection and Correction for Transient-Error Tolerance and Adaptation to PVT Variation”, Proc. Int. Solid State Circuits Conf., pp. 284–286, Feb. 2010.

S. Das, et al., “A Self-Tuning DVS Processor Using Delay-Error Detection and Correction”, IEEE J. Solid-State Circuits, Vol. 41, pp. 792–804, Apr. 2006.

J. Goodacre, “The Design Dilemma: Multiprocessing using Multiprocessors and Multithreading”, Design and Reuse Forum, 2006.

Y. Huai, “Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects”, AAPPS Bulletin, Vol. 18, No. 6, pp. 34–40, Dec. 2008.

M. Keating et al., “Low Power Methodology Manual”, Springer 2007.

D. Patterson and J. Hennesy, “Computer Organization and Design, The Hardware/Software Interface, Fourth Edition”, Elsevier 2009.

Texas Instruments, OMAP 4 mobile applications platform, http://focus.ti.com/lit/ml/swpt034/swpt034.pdf, downloaded Oct. 11 2010.

J. Wawrzynek et al., “High-End Reconfigurable Computing”, Berkeley Wireless Reseasrch Center Retreat, Jan. 2004.

Chapter 11
Invasive Computing: An Overview

Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting

Abstract A novel paradigm for designing and programming future parallel computing systems called *invasive computing* is proposed. The main idea and novelty of invasive computing is to introduce *resource-aware programming* support in the sense that a given program gets the ability to explore and dynamically spread its computations to neighbour processors in a phase called invasion, then to execute portions of code of high parallelism degree in parallel based on the available *invasible* region on a given multi-processor architecture. Afterwards, once the program terminates or if the degree of parallelism should be lower again, the program may enter a *retreat* phase, deallocate resources and resume execution again, for example, sequentially on a single processor. To support this idea of self-adaptive and resource-aware programming, not only new programming concepts, languages, compilers and operating systems are necessary but also revolutionary architectural changes in the design of Multi-Processor Systems-on-a-Chip must be provided so to efficiently support invasion, infection and retreat operations involving concepts for dynamic processor, interconnect and memory reconfiguration. This contribution reveals the main ideas, potential benefits and challenges for supporting invasive computing at the architectural, programming and compiler level in the future. It serves to give an overview of required research topics rather than being able to present mature solutions yet.

Keywords B (Hardware) · B.7 (Hardware: Integrated Circuits) · C (Computer Systems Organisation) · C.1 (Computer Systems Organisation; Processor Architectures) · C.3 (Computer Systems Organisation: Special-Purpose and Application-Based Systems)

J. Teich (✉)
Hardware/Software Co-Design, Department of Computer Science,
University of Erlangen-Nuremberg, Germany
e-mail: teich@informatik.uni-erlangen.de

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1_11,

11.1 Introduction

Decreasing feature sizes have already led to a rethinking of how to design multi-million transistor system-on-a-chip architectures envisioning dramatically increasing rates of temporary and permanent faults as well as feature variations. The major question will thus be how to deal with this imperfect world [1] in which components will become more and more unreliable. As we can foresee SoCs with 1,000 or more processors on a single chip in the year 2020, static and central management concepts to control the execution of all resources might have met their limits long before and are therefore not appropriate. Invasion might provide the required self-organising behaviour to conventional programs for being able not only to tolerate certain types of faults and cope with feature variations, but also to provide scalability, higher resource utilisation numbers and, hopefully, also performance gains by adjusting the amount of allocated resources to the temporal needs of a running application. This thought might open a new way of thinking about parallel algorithm design as well. Based on algorithms utilising invasion and negotiating resources with others, we can imagine that corresponding programs become personalised objects, competing with other applications running simultaneously on a Multi-Processor System-on-a-Chip (MPSoC).

11.1.1 Parallel Processing Has Become Mainstream

Miniaturisation in the nano era makes it possible already now to implement billions of transistors, and hence, massively parallel computers on a single chip with typically 100s of processing elements.

Whereas parallel computing tended to be only possible in huge high performance computing centres some years ago, we see parallel processor technology already in home PCs, but interestingly also in domain-specific products, such as computer graphics and gaming devices. In the following description, we picked out just four representative instances out of many domain-specific examples of massively parallel computing devices using MPSoC technology that have already found their way into our homes:

- *Visual Computing and Computer Graphics*: As an example, the Fermi CUDA architecture [2], as it is implemented on NVIDIA graphics processing units (GPUs) is equipped with theoretically upto 512 thread processors which provide more computing power than 1 TFLOPS as well as 6 GB GDDR5 (Graphics Double Data Rate, version 5) RAM. To enable flexible, programmable graphics and high-performance computing, NVIDIA has developed the CUDA scalable unified graphics and parallel computing architecture [3]. Its scalable parallel array of processors is massively multi-threaded and programmable in C or via graphics APIs. Another platform originally targeting visual computing is Intel's Larrabee [4]. Although Intel will not ship Larrabee chips, their new Many Integrated Core (MIC) architecture is based on Larrabee's architecture and is focused on high-performance computing. The release of the first Intel MIC

chips codenamed *Knights Corner* is planned for 2011 and brings a new may-core programming model using multiple in-order x86 CPU cores that are enhanced by a wide vector processor unit, as well as several fixed function logic blocks. This provides dramatically higher performance per Watt and per unit of area than out-of-order CPUs in case of highly parallel workloads. It also greatly increases the flexibility and programmability of the architecture as compared to standard GPUs. A coherent on-die 2nd level cache allows efficient inter-processor communication and high-bandwidth local data access by CPU cores.

Task scheduling is performed entirely with software in Knights Corner, rather than in fixed function logic.

- *Gaming*: The Cell processor [5] such as part of Sony's PLAYSTATION 3 consists of a 64-bit Power Architecture processor coupled with multiple synergistic processors, a flexible I/O interface and a memory interface controller that supports multiple operating systems. This multi-core SoC, implemented in 65 nm Silicon On Insulator (SOI) technology, achieves a high clock rate by maximising custom circuit design while maintaining reasonable complexity through design modularity and reuse.
- *Signal Processing*: Application-specific tightly-coupled processor arrays (TCPAs). For applications such as 1D or 2D signal processing, linear algebra and image processing tasks, Fig. 11.1 shows an example of an MPSoC

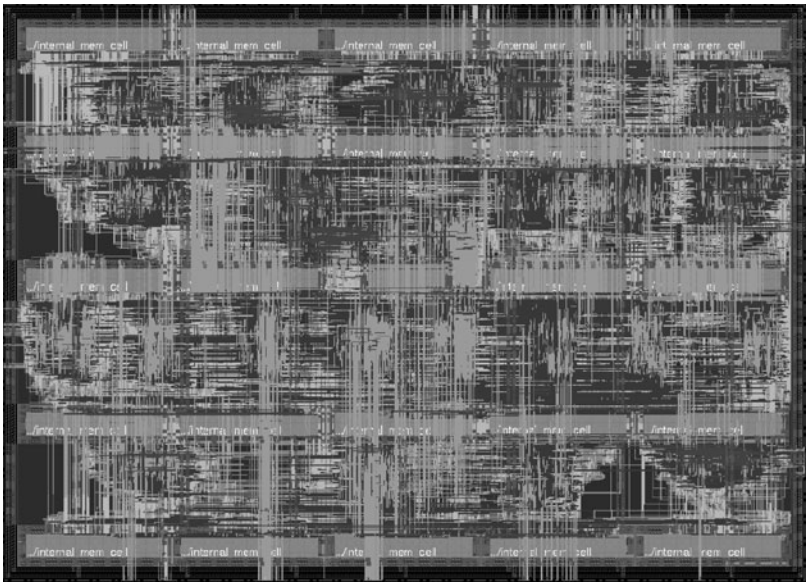


Fig. 11.1 Architecture of a 8×3 processor MPSoC customised for image filtering type of operations. Technology: CMOS 1.0 V supply voltage, 9 metal layers, 90 nm standard cell design. VLIW memory/PE: 16 times 128, FUs/PE: 2 times Add, 2 times Mul, 1 times Shift, 1 times DPU. Registers/PE: 15. Register file/PE: 11 read and 12 write ports. Configuration Memory: 1024 times 32 = 4 kByte. Operating frequency: 200 MHz. Peak Performance: 24 GOPS. Power consumption: 132.7 mW @ 200 MHz (hybrid clock gating). Power efficiency: 0.6 mW/MHz. Chair of Hardware/Software Co-Design, Erlangen, 2009

integrating 24 VLIW processors designed in Erlangen with more than one million transistors on a single chip of size about 2 mm^2. Contrary to the previous architectures, this architecture is customisable with respect to instruction set, processor types and interconnect [6, 7]. For such applications, the overhead and bottlenecks of program and data memory including caches can often be avoided giving more chip area for computations than for storage and management functions. Due to the fact that the instruction set, word precisions, number of functional units and many other parameters of the architecture may be customised for a set of dedicated application programs to run, we call such architectures weakly-programmable. It is unique that the inter-processor interconnect topology may be reconfigured at run-time within a few clock cycles time by means of hardware reconfiguration. Also, the chip features ultra-low power consumption of about 130 mW when operating at 200 MHz.

- *NoC*: In [8], Intel demonstrates the feasibility of packing 80 tile processors on a single chip by introducing a 275 mm^2 network-on-a-chip (NoC) architecture where each tile processor is arranged as a 10×8 2D array of floating-point cores and packet-switched routers, operating at 4 GHz. The design employs mesochronous clocking, fine-grained clock gating, dynamic sleep transistors and body-bias techniques. The 65 nm 100 M transistor die is designed to achieve a peak performance of 1.0 TFLOPS at 1 V while dissipating 98 W. Very recently, Intel introduced a successor chip, called Single-chip Cloud Computer (SCC), with 48 fully programmable processing cores manufactured in 45 nm technology. In contrast to the 80 core prototype, Intel plans to build 100 or more experimental SCC chips for use by industrial and academic research collaborators.

Note that there exists a multitude of other typically domain-specific massively parallel MPSoCs that cannot be listed here. Different domains of applications have also brought up completely different types of architectures. One major distinguishing factor is that concurrency is typically exploited at different levels of granularity and levels of architectural parallelism as shown, for example, in Fig. 11.2. Starting with process- and thread-level applications running on high performance computing (HPC) machines or heterogeneous Multi-Processor System-on-a-Chip architectures (MPSoCs) down to the loop-level for which TCPAs match well, and finally instruction and bit-level type of operations.

11.1.2 Obstacles and Pitfalls in the Years 2020 and Beyond

Already now can be foreseen that MPSoCs in the years 2020 and beyond will allow to incorporate about 1,000 and more processors on a single chip. However, we can anticipate several major bottlenecks and shortcomings when obeying existing and common principles of designing and programming MPSoCs. The challenges related to these problems have motivated our idea of invasive computing:

- *Programmability*: How to map algorithms and programs to 1,000 processors or more in space and time to benefit from the massive parallelism available and by

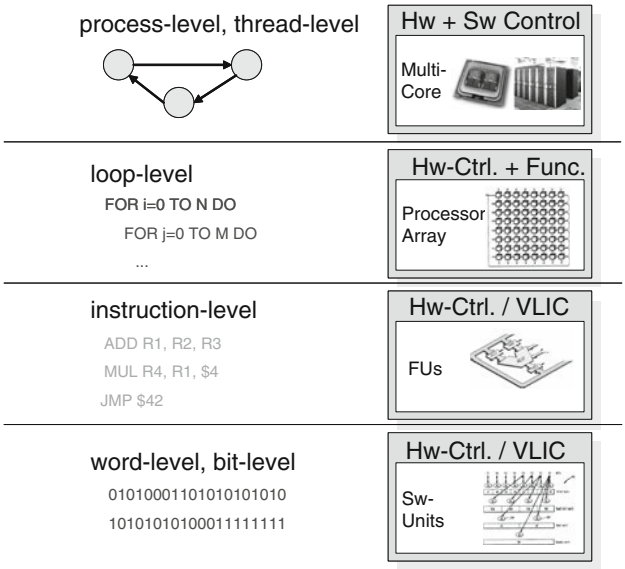


Fig. 11.2 Levels of parallelism including process-level, thread-level, loop-level, instruction-level as well as word-level and bit-level. The architectural correspondence is shown on the right side including parallel computers, heterogeneous MPSoCs and tightly-coupled processor array architectures, finally VLIW and bit-level parallel computing. Invasive computing shall be investigated on all shown levels

tolerating defects and manufacturing variations concerning memory, communication and processor resources properly?

- *Adaptivity*: The computing requirements of emerging applications to run on an MPSoC may not be known at compile-time. Furthermore, there is the problem of how to dynamically control and distribute resources among different applications running on a single chip, to satisfy high resource utilisation and high performance constraints. How and to what degree should MPSoCs therefore be equipped with support for adaptivity, for example, reconfigurability, and to what degree (hardware/software, bit, word, loop, thread, process-level)? Which gains in resource utilisation may be expected through run-time adaptivity and temporary resource occupancy?
- *Scalability*: How to specify algorithms and programs and generate executable programs that run efficiently without change on either 1, 2, or N processors? Is this possible at all?
- *Physical Constraints*: Heat dissipation will be another bottleneck. We need sophisticated methods and architectural support to run algorithms at different speeds, to exploit parallelism for power reduction and to manage the chip area in a decentralised manner.
- *Reliability and Fault-Tolerance*: The continuous decrease of feature sizes will not only inevitably lead to higher variances of physical parameters, but also

affect reliability, which is impaired by degradation effects [1]. In consequence, techniques must be developed to compensate and tolerate such variations as well as temporal and permanent faults, that is, the execution of applications shall be immune against these. Hence, conventional and centralised control will fall off this requirement, see, for example, [1]. Furthermore, the control of such a parallel computer with 100s–1,000s of processors would also become a major performance bottleneck if centrally controlled.

Finally, whereas for a single application the optimal mapping onto a set of processors may be computed and optimised often at compile-time which holds in particular for loop-level parallelism and corresponding programs [6, 9, 10], a static mapping might not be feasible for execution at run-time because of time-variant resource constraints or dynamic load changes. Ideally, the interconnect structure should be flexible enough to dynamically reconfigure different topologies between components with little reconfiguration and area overheads.

With the above problems in mind, we propose a new programming paradigm called invasive computing. In order for this kind of resource-aware programming concept to become reality and main stream, new processor, interconnect and memory architectures, exploiting dynamic hardware reconfiguration will be required. Invasive computing distinguishes itself from common mainstream principles of algorithm and architecture design in industry on multiple (e.g., dual, quadruple) and many-core architectures, as these will still be programmed more or less using conventional languages and programming concepts. To increase the scope and applicability, however, we do require that legacy programs shall still be executable within an invasive processor architecture. To achieve this, a migration path from traditional programming to the new invasive programming paradigm needs to be established.

11.1.3 Principles and Challenges of Invasive Computing

In vision of the above capabilities of todays hardware technology, we would like to propose a completely new paradigm of parallel computing called invasive computing in the following.

One way of how to manage the control of parallel execution in MPSoCs with 100s of processors in the future would obviously be to give the power to manage resources, that is, link configurations and processing elements to the programs themselves and thus, have the running programs manage and coordinate the processing resources themselves to a certain degree and in context of the state of the underlying compute hardware. This cries for the notion of a self-organising parallel program behaviour called invasive programming.

Definition: Invasive Programming denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication and memory resources in the neighbourhood of its actual computing environment, to then execute in

parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.

We shall show next what challenges will need to be solved to support invasive computing on the architectural, on the notational and on the algorithmic and programming language sides.

11.1.4 Architectural Challenges for the Support of Invasive Computing

Figure 11.3 shows how a generic invasive multi-processor architecture including loosely-coupled processors as well as tightly-coupled co-processor arrays may look like.

To present the possible operational principles of invasive computing, we shall provide an example scenario each for (a) TCPAs, (b) loosely-coupled, heterogeneous systems and (c) HPC systems.

An example of how invasion might operate at the level of loop programs for a TCPA as part of a heterogeneous architecture shown in Fig. 11.3 is demonstrated in

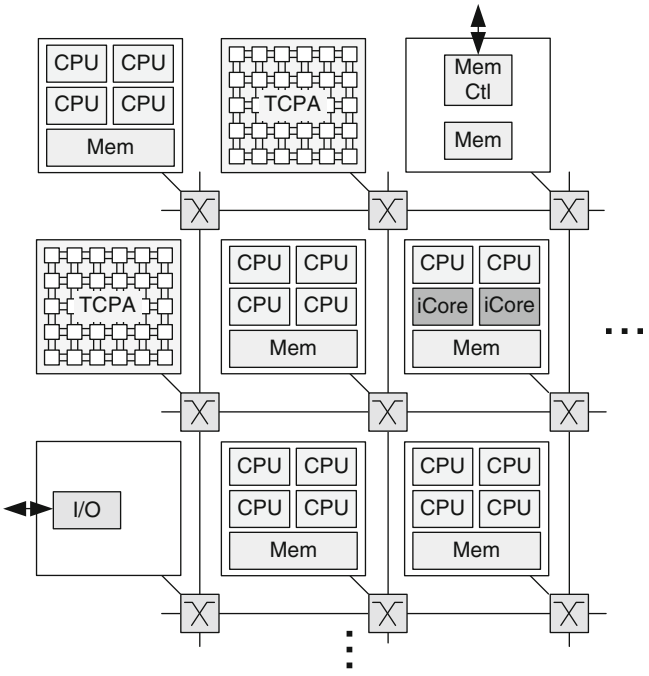


Fig. 11.3 Generic invasive multi-processor architecture including several loosely-coupled processors (standard RISC CPUs and invasive cores, so-called i-Cores) as well as tightly-coupled processor arrays (TCPAs)

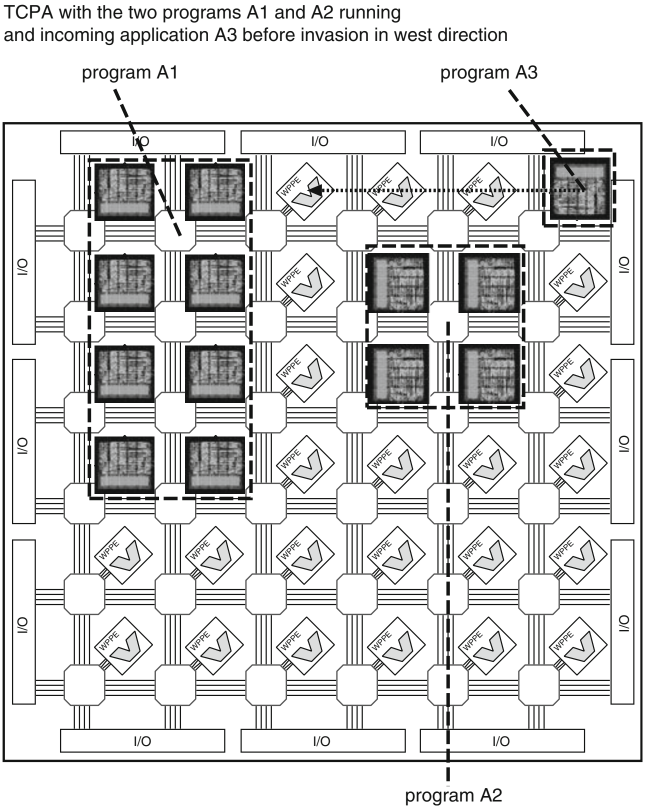


Fig. 11.4 Case study showing a signal processing application (A3) invading a tightly-coupled processor array (TCPA) on which the two programs A1 and A2 are already executing. Program A3 invades its neighbour processors to the west, infects claimed resources by implanting its program into these claimed cells and then executes in parallel until termination. Subsequently, it may free used resources again (retreat) by allowing other neighbour cells to invade

Fig. 11.4. There, two programs A1 and A2 are running in parallel and a third program A3 starting its execution on a single processor in the upper right corner.

In a phase of invasion, A3 tries to claim all of its neighbour processors to the west to contribute their resources (memory, wiring harness and processing elements) to a joint parallel execution. Once having reached borders of invasion, for example, given by resources allocated already to running applications, or, in case the degree of invasion is optimally matching the degree of available parallelism, the invasive program starts to copy its own or a different program into all claimed cells and then starts executing in parallel, see, for example, Fig. 11.5.

In case the program terminates or does not need all acquired resources any more, the program could collectively execute a retreat operation and free all processor resources again. An example of a retreat phase is shown in Fig. 11.6. Please note that invade and retreat phases may evolve concurrently in a massively parallel system, either iteratively or recursively.

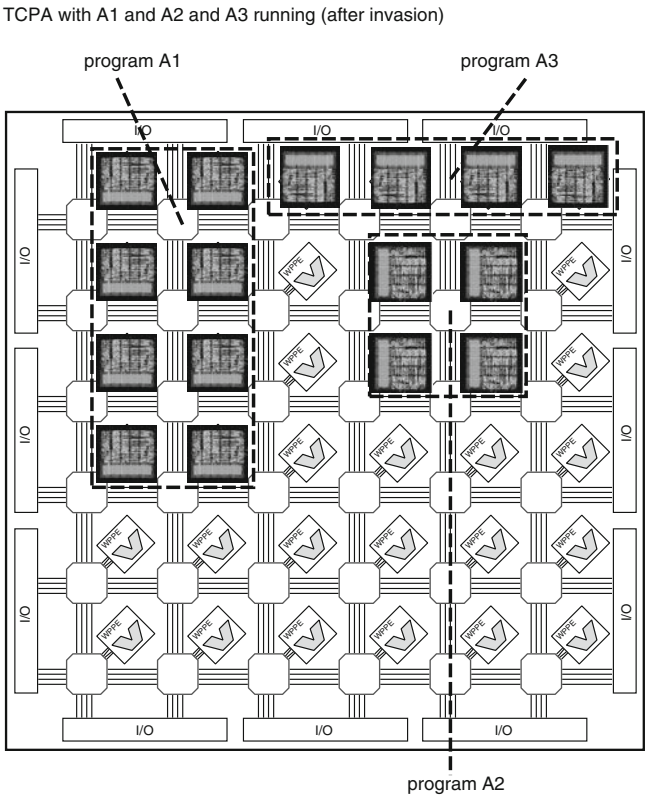


Fig. 11.5 TCPA hosting a signal processing application (A3) together with two other programs A1 and A2 (after invasion)

Technically speaking, at least three basic operations to support invasive programming will be needed, namely invade, infect and retreat. It will be explained next that these can be implemented with very little overhead on reconfigurable MPSoC architectures such as a TCPA like a WPPA [7] or the AMURHA [11] architecture in a few steps by issuing reconfiguration commands that are able to reconfigure subdomains of interconnect and cell programs collectively in just a few clock cycles, hence with very low overhead. In [6], for example, we have presented a masking scheme such that a single processor program of size L can be copied in O(L) clock cycles into an arbitrarily sized rectangular processor region of size $N \times M$.

Hence, the time overhead for an infection phase, comparable to the infection of a cell of a living being by a virus, can be implemented in linear time with respect to the size of a given binary program memory image L. In case of a tightly-coupled processor array running typically in a clock-synchronous manner, we intend to prove that invasion requires only $O(\max\{N,M\})$ clock cycles where $N \times M$ denotes the maximally claimable or claimed rectangular processor region. Before subsequent cell infection, an invasion hardware flag might be introduced to signal that a cell is

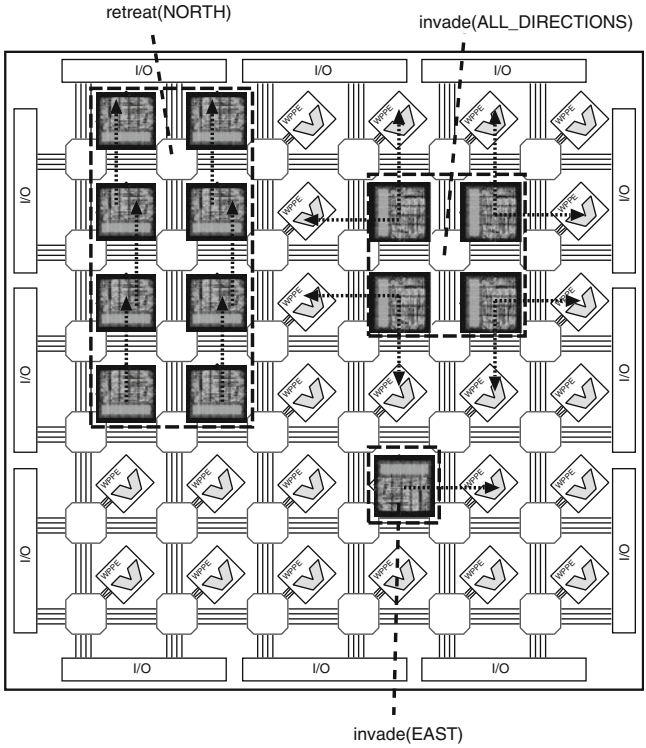


Fig. 11.6 Options for invasion (uni- vs. multi-directional) and retreat phases

immune against subsequent invasion requests until this flag is reset in the retreat phase. In contrast to the initial invasion phase, the retreat phase serves to free claimed resources after parallel execution. As for invasion, we intend to show that retreat can be performed decentrally in time O(max{N,M}) [12].

The principles of invasion apply similarly to heterogeneous MPSoC architectures, as shown in Fig. 11.2. Here, invasion might be explored at the thread-level and implemented, for example, by using an agent-based approach that distributes programs or program threads over processor resources of different kinds.

At this level, dynamic load-balancing techniques might be applied to implement invasion. For example, diffusion-based load balancing methods [13–15] are a simple and robust distributed approach for this purpose. Even centralised algorithms based on global prioritisation can be made scalable using distributed priority queues [16]. Very good load balancing can be achieved by a combination of randomisation and redundancy, using fully distributed and fast algorithms (e.g., [17]).

Figure 11.7 shows by example how invasive computing for loosely-coupled multi-core architectures consisting of standard RISC processors could work. These cores may – together with local memory blocks or hardware accelerators (not shown in the figure) – be clustered in compute tiles, which are connected through a flexible high-speed NoC interconnect. In general, an operating system is expected

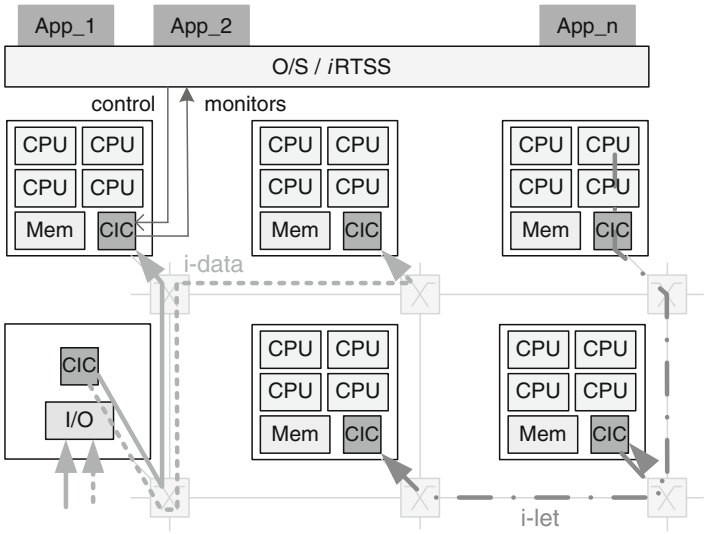


Fig. 11.7 Invasive computing on a loosely-coupled MPSoC architecture

to run in a distributed or multi-instance way on several cores and may be supported by a run-time environment.

To enable invasive computing on such MPSoCs, an efficient, dynamic assignment of processing requests to processor cores is required. Time constants for starting processing on newly claimed CPUs is expected to be considerably longer than in the case of tightly-coupled processors. Therefore, we envision the corresponding mechanisms to be implemented in a hardware-based support infrastructure using dynamic Many-Core i-let[1] Controllers (CIC), which help to limit the impairments of any overhead associated with the invasion/infection process.

Invasive operating and run-time support services invade processing resources when new processing requirements have to be fulfilled. The invasion process considers monitoring information on the status of the hardware platform received via the CICs, which are contained in each compute and I/O tile. As a result of invasion, CICs are configured for the appropriate forwarding of the associated processing requests. This forwarding actually corresponds to the infection of the invaded processor cores. The final assignment may be based on a set of rules that implement an overall optimisation strategy given by the invasive operating system. Criteria to be taken into account in this context may, for example, be the load situation of processing or communication resources, the reliability profiles of the cores or the temperature profile of the die.

[1] For the explanation of the i-let concept, see paragraph “units of invasion” below.

The CICs dynamically map processing requests to processor cores under the control of the operating system and the run-time environment (iRTSS). These requests may either be generated when

- An application wants to spawn additional parallel processes or threads, for example, depending on interim processing results (shown in the right part of Fig. 11.7, dashed-dotted line), or when
- Data arriving via external interfaces (e.g., sensor or video data, network packets), which represent processing requests, have to be distributed to the appropriate processing resources (shown in the left part of Fig. 11.7, straight and dashed arrows)

In the first case, a so-called i-let will be created for a new thread to be spawned and sent towards the invaded resource. The CIC in the target compute tile will distribute the i-let to one of the cores depending on the rules given by the operating system/iRTSS, which take into account the actual load situation and other status information. In case there is not enough processing capacity available locally, the rules may also indicate to forward the i-let to the CIC of a compute tile with free resources in the neighbourhood, as shown in Fig. 11.7 for the bottom right compute tile.

For the second case, if more traffic arrives from external senders than can be processed by the left compute tile, the invasive operating system or even the CIC itself – if authorised by the operating system – shall invade a further CPU cluster. In case of success, the CIC rules would be updated and in consequence excess requests (designated as i-data – invasive data – in Fig. 11.7) would be distributed to the newly invaded resources to cope with the increased processing requirements. To avoid latencies in the invasion triggered by the operating system, resources may already have been invaded earlier, for example, when a threshold below the acceptable load is exceeded.

In this way, MPSoCs built out of legacy IP cores can be enabled for invasion and thus provide applications with the required processing resources at system run-time, which helps to meet performance requirements and at the same time to facilitate efficient concurrent use of the platform. As applications can expand and contract on the MPSoC dynamically, we also expect that less resources are required in total to provide the same performance as would be needed if resource assignment is done at compile-time.

Finally, the paradigm of invasion offers even a new perspective for programming large-scale HPC computers according to Fig. 11.2 with respect to the problem classes of space partitioning and adaptive resource management.

Today, resource management on large-scale parallel systems is done using space partitioning: The available processors and memories are statically partitioned among parallel jobs. Once a job is started on these resources, it has exclusive access for its entire life-time. This strategy becomes inadequate if more and more parallelism has to be exploited to obtain high performance on future petascale systems. As the cores will most likely not be getting much faster (in terms of clock rates) in the future, applications will benefit from a maximum number of processors only

during certain phases of their life-time, and can run efficiently during the rest of their life-time using a smaller number of processors.

Moreover, there exist applications that have inherently variable requirements for resources. For example, multi-grid applications work on multiple grid levels ranging from fine to coarse grids. On fine grids, many processors can work efficiently in parallel while only a few are able to do so on coarse grids. Thus, processors can be freed during coarse grid computation and assigned to other jobs. Another class of applications is that of adaptive grid applications, where the grid is dynamically refined according to the current solution. Applications may also proceed through different phases in which different amount of parallelism might be available. For example, while in one phase, a pipeline structure with four stages can be used, two different functions can be computed in parallel in another phase.

11.1.5 Notational Issues for the Support of Invasive Computing

Obviously, to enable a program to distribute its computations for parallel execution through the concept of invasion, we need to establish a new programming paradigm and program notation to express the mentioned phases of (a) invasion, (b) infection and (c) retreat. Either existing parallel program notations and languages might be extended or pragma and special compiler modifications might be established to allow the specification of invasive programs.

In the following, we propose a minimal set of required commands to support resource-aware programming, independent of the level of concurrency and architectural abstraction. This informal and minimal notation only serves to give an idea of what kind of basic commands will be needed to support invasive programming and how such programs could be structured.

Invade. To explore and claim resources in the (logical) neighbourhood of a processor running a given program, the `invade` instruction is needed. This command could have the following syntax:

$$\texttt{P} = \texttt{invade}\ (\texttt{sender\_id}, \texttt{direction}, \texttt{constraints})$$

where `sender_id` is the identifier, for example, coordinate of the processor starting the invasion, and `direction` encodes the direction on the MPSoC to invade, for example, `North`, `South`, `West`, `East` or `All` in which case the invasion is carried out in all directions of its neighbourhood. For heterogeneous MPSoC architectures, the neighbourhood could be defined differently, for example, by the number of hops in a NoC. Other parameters not shown here are `constraints` that could specify whether and how not only program memory, but also data memory and interconnect structures should be claimed during invasion. Further, invasion might be restricted to certain types of processors and resources. During invasion, each claimed resource is immediately immunised against invasion by other applications and

until they are freed explicitly in the final retreat phase. Hence, the operational semantics of the invade command is resource reservation.

Now, a typical behaviour of an invasive program could be to claim as many resources in its neighbourhood as possible. Using the `invade` command, a program could determine the largest set of resources to run on in a fully decentralised manner. The return parameter P could, for example, encode either the number of processors or the size of the region it was able to successfully invade. Another variant of `invade` could be to claim only a fixed number of processors in each direction. For example, Fig. 11.4 illustrates the case of a signal processing application A3 running concurrently with the two applications A1 and A2. Here, the signal processing application is issuing an invade command to all processors to its west. Figure 11.5 shows the running algorithm A3 after successful invasion.

Infect. Once the borders of invasion are determined and corresponding resources reserved, the initial single-processor program could issue an `infect` command that copies the program like a virus into all claimed processors. In case of a TCPA architecture, we anticipate to be able to show how to implement this operation for a rectangular domain of processors in time O(L), where L is the size of the initial program. Also, the interconnect reconfiguration may be initialised for subsequent parallel execution. As for the `invade` command, `infect` could have several more parameters considering modifications to apply to the copied programs such as parameter settings, and of course also the reconfiguration of interconnect and memory resource settings. Note that the infect command in its most general form might also allow a program to copy not only its own, but also foreign code to other processors. After infection, the parallel execution of the initial and all infected resources may start.

Retreat. Once the parallel execution is finished, each program may terminate or just allow the invasion of its invaded resources by other programs. Using a special command called `retreat`, a processor can, for example, in the simplest case just initiate to reset flags that subsequently would allow other invaders to succeed. Again, this retreat procedure may hold for interconnect as well as processing and memory resources and is therefore typically parametrised. Different possible options of typical invade and retreat commands for TCPAs are shown in Fig. 11.6.

11.1.6 Algorithmic and Language Challenges for the Support of Invasive Computing

We have stated that resource-awareness will be central to invasive computing. Accordingly, not only the programmer, but already the algorithm designers should reflect and incorporate this idea that algorithms may interact and react to the temporal availability and state of processing resources and possible external conditions.

However, this invasive computing paradigm raises interesting questions for algorithm design and complexity analysis. It will also generate questions

concerning programming languages, such as semantic properties of a core invasive language with explicit resource-awareness.

We would like to mention, however, that the idea of invasion is not tightly related or restricted to a certain programming notation or language. We plan to define fundamental language constructs for invasion and resource-awareness, and then embed these constructs into existing languages, such as C++ or X10. In fact, according to preliminary studies it seems that X10 [18] is the only available parallel language which already offers a fundamental concept necessary for invasive computing: X10 supports distributed, heterogeneous processor/memory architectures. Also, we would like to show how invasion can be supported in current programming models, such as OpenMP and MPI.

What is essential and novel in the presented idea of invasive algorithms is that to support the concept of invasion properly, a program must be able to issue instructions, commands, statements, function calls or process creation and termination commands that allow itself to explore and claim hardware resources. There is a need to study architectural changes with respect to existing MPSoC architectures to support these concepts properly.

Resource-aware Programming. Invade, infect and retreat constitute the basic operations that shall help a programmer to manipulate the execution behaviour of a program on the underlying parallel hardware platform.

On the other hand, invasive computing shall provide and help the programmer to decide whether to invade at a certain point of program execution in dependence of the state of the underlying machine. For example, such a decision might be influenced by the local temperature profile of a processor, by the current load, by certain permissions to invade resources and, most importantly, also by the correct functioning of the resources. Taking into account such information from the hardware up to the application-level provides an interesting feedback-loop as shown in Fig. 11.8 that enables resource-aware programming.

For example, the decision to invade a set of processors may be taken conditionally at a certain point within a given invasive program depending on whether the temperature of a processor is exceeding 85°C and if there are processors around with permission to be invaded and average load under 50%. More complex scenarios may be defined as well.

Such information provided from the hardware to the application program could thus lead to program executions that take the dynamic situation of the underlying hardware platform into account and permits to dynamically exploit the major benefits of invasive computing, namely increase of fault-tolerance, performance, utilisation and reliability.

Units of Invasion. In the following, a piece of program subjected to invasive parallel execution is referred to as an "invasive-let": in short, i-let.[2] An i-let is the

[2] This conception goes back to the notion of a "servlet", which is a (Java) application program snippet targeted for execution within a web server.

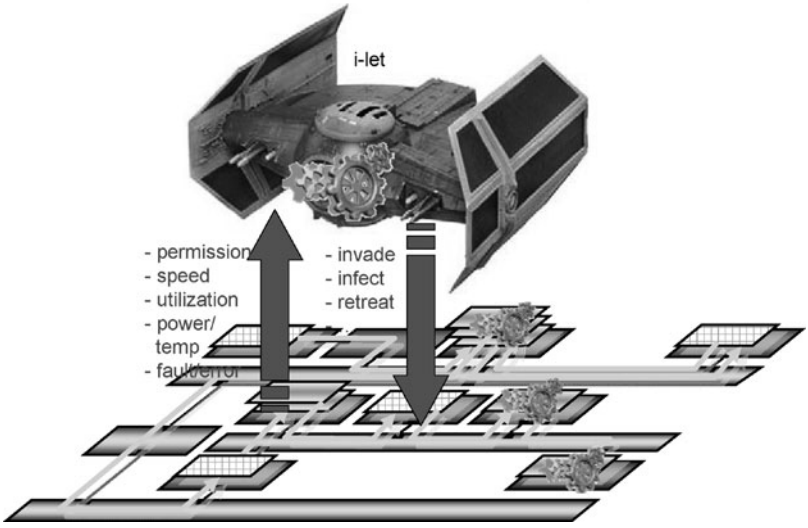


Fig. 11.8 Resource-aware programming is a main feature of invasive computing. By providing a feedback-loop between application and underlying hardware platform, an application program/thread, called i-let, may decide if and which resources to invade, infect, or retreat at run-time; depending on the current state of the underlying parallel hardware platform. Examples of properties that need to be exploited are permissions, speed/performance as well as utilisation monitor information, but also power and temperature information and, most importantly, also information about faults and errors

fundamental abstraction of a program section being aware of potential concurrent execution. Potential because of the semantics of an `invade` command, which may indicate allocation of only one processing unit, for example, although plenty of these might have been requested. Concurrent, instead of parallel, because of the possibility that an allocated processing element will have to be multiplexed (in time) amongst several threads of control to make available the grade of "parallelism" as demanded by the respective application.

Such an abstraction becomes indispensable as a consequence of resource-aware programming, in which the program structure and organisation must allow for execution patterns independently of the actual number of processing elements available at a time. By matching the result of an `invade` command, an i-let "entity" will then be handed over to `infect` to deploy the program snippet to be run concurrently. Similarly, `retreat` cleans processing elements up from the i-let entities that have been setup by `infect`.

Depending on the considered level of abstraction, different i-let entities are distinguished: candidate, instance, incarnation and execution. An i-let candidate represents an occurrence of a parallel program section that might result in different samples. These samples discriminate in the grade of parallelism as, for example, specified by a set of algorithms given the same problem to be solved. In such a

setting, each of these algorithms is considered to be optimal only for a certain range in the exploration space.

In general, i-let candidates will be identified at compilation-time based on dedicated concepts/constructs of the programming language (e.g., `async` in X10 [18]), assisted by the programmer. Technically, a candidate is made up of a specific composition of code and data. This composition is dealt with as a single unit of potential concurrent processing. Each of these unit descriptions is referred to as an i-let instance. Given that an i-let candidate possibly comes in different samples, as explained above, within a single invasive-parallel program, the existence of different i-let instances will be a logical consequence. However, this is not confined to a categorically one-to-one mapping between i-let candidate and instance. A one-to-many mapping is conceivable as well. Cases of the latter are, for example, invasive-parallel program patterns whose i-let candidates arrange for different granularity in terms of program text and data sections, depending on the characteristics of the hardware resources (logically, virtually) available for parallel processing. Each of these will then make up an i-let instance. Options include, for a single i-let candidate, a set of i-let instances likewise tailor-made for a TCPA, ASIP, dual-, quad-, hexa-, octa- and even many-core RISC or CISC.

An i-let instance will be the actual parameter to the `infect` command. Upon execution of `infect`, the specified instance becomes an i-let incarnation; that is, an i-let entity bound to (physical) resources and set ready for execution. Depending on these resources as well as on the operating mode subjected to a particular processing element, an i-let incarnation technically represents a thread of control of a different "weight class". In case of a TCPA, for example, each of these incarnations will hold its own processing elements. In contrast, several incarnations of the same or different i-let instances may share a single processing element in case of a conventional (multi-core) processor. The latter mode of operation typically assumes the implementation of a thread concept as a technical means for processor multiplexing. The need for processor multiplexing may be a temporary demand, depending on the actual load of the computing machine and the respective user profile of an application program.

To be able to abstract from the actual mode of operation of some processing element, an i-let incarnation does not yet make assumptions about a specific "medium of activity", but it only knows about the type of its dedicated processing element. It will be the occurrence as an i-let execution that manifests that very medium. Thus, at different points in time, an i-let incarnation for the same processing element may result in different sorts of i-let executions: The binding between incarnation and execution of the same i-let may be dynamic and may change between periods of dispatching.

Behind this approach stands the idea of an integrated cooperation of different domains at different levels of abstraction. At the bottom, the operating system takes care of i-let incarnation/execution management; in the middle, the language-level run-time system does so for i-let instances; and at the top, the compiler, assisted by the programmers, provides for the i-let candidates. Altogether, this establishes an application-centric environment for resource-aware programming and invasive-parallel execution of concurrent processes.

11.1.7 Operating System Issues of Invasive Computing

The concept of resource-aware programming calls for operating-system functions by means of which the use of hardware as well as software resources becomes possible in a way that allows applications to make controlled progress depending on the actual state of the underlying machine. Resources must be related to invading execution threads in an application-oriented manner. If necessary, a certain resource needs to be bound, for example, exclusively to a particular thread or it has to be shareable by a specific group of threads, physically or virtually. Optionally, the binding may be static or dynamic, possibly accompanied by a signalling mechanism, likewise to asynchronously communicate resource-related events (e.g., demand, release, consumption, or contention) from system to user level.

To support resource-aware execution of invasive-parallel programs as indicated above, two fundamental operating system abstractions are being considered: the claim and the team. A claim represents a particular set of hardware resources made available to an invading application. Typically, a claim is a set of (tightly- or loosely coupled) processing elements, but it may also describe memory or communication resources. Claims are hierarchically structured as (1) each of its constituents is already a (single-element) claim and (2) a claim consists of a set of claims. This shall allow for the marshalling of homogeneous or heterogeneous clusters of processing elements. More specifically, a claim of processing elements also provides means for implementing a place, which is the concept of the programming language X10 [18] to support a partitioned global address space. However, unlike places, claims do not only define a shared memory domain but also aim at providing a distributed-memory dimension.

In contrast, a team is the means of abstraction from a specific use of a particular claim to model some run-time behaviour as intended by a given application. Similar to conventional computing, where a process represents a program in execution, a team represents an invasive-parallel program in execution. More specifically, a team is a set of i-let entities and may be hierarchically structured as well: (1) every i-let already makes up a (single-element) team and (2) a team consists of a set of teams. Teams provide means for the clustering or arrangement of interrelated threads of execution of an invasive-parallel program. In this setting, an execution thread may characterise an i-let instance, incarnation, or execution, depending on whether that thread has been marshalled only, already deployed, or dispatched.

Application-oriented Run-Time Executive. A team needs to be made fit to its claim. Reconsidering the three fundamental primitives for invasive computing, `invade` allocates and returns a claim, which, in addition to a team, will be handed over to `infect` to deploy i-let instances in accordance with the claim properties. For deallocation (`invade` unaccompanied by `infect`) or depollution (`invade` accompanied by `infect`), `retreat` is provided with the claim (set-out by `invade`) to be released or cleaned up, respectively.

Asserting a claim using `invade` will entail local and global resource allocation decisions to be made by the operating system. Depending on the invading

application, different criteria with respect to performance and efficiency need to be taken into account and brought in line. In such a setting of possibly conflicting resource allocation demands, teams are considered as the kind of mechanism that enables the operating system to let the computing machine work for applications in a flexible and optimal manner. Teams will be dispatched on their claims according to a schedule that aims at satisfying the application demands. To improve application performance, for example, this may result in a team schedule that prevents or avoids contention in case a particular claim is being multiplexed by otherwise unrelated teams. As a consequence – and to come full circle – resource-aware programming also means to pass (statically or dynamically derived) a priori knowledge about prospective run-time behaviour from user to system level to aid or direct the operating system in the process of conflict resolution and negotiating compromises.

Integrated Cooperative Execution. To achieve high performance and efficiency in the execution of thread-parallel invasive programs, various functions related to different levels of abstractions of the computing system need to cooperate. Figure 11.9 exemplifies such an interaction by roughly sketching major activities associated with the release and execution of `invade`, `infect` and `retreat`. As in conventional computing systems, developers are free to choose the proper level of abstraction for application programming and thus may directly employ `invade`, `infect` and `retreat` in their programs. One of the ideas of invasive computing, however, is to also let a compiler (semi-) automatically derive these primitives from programs written in a problem-oriented programming language. The displayed nuance of abstraction interrelates a problem-oriented programming language level (application, X10), an assembly level (compiler, run-time system), a machine programming level (run-time support system, operating system) and the hardware level. In Fig. 11.9, these levels are vertically arranged, in terms of columns from left to right. In this setting, the hardware level implements the real machine of the computing system, while the other three levels implement abstract machines. The functions (i.e., operations) provided by each of these machines are dedicated to the purpose of supporting invasive-parallel resource-aware programming.

11.2 Examples of Invasive Programs

To illustrate resource-aware programming and invasive computing, we shall present four preliminary, but representative examples of invasive programs. Note that these examples are pseudocode and are designed to demonstrate fundamental invasive techniques. They should not be interpreted as examples for a new invasive programming language.

The first example (Fig. 11.10) is a simple invasive ray tracer. Note that the goal of this fragment of an invasive ray tracer is not ultimate performance, but maximal flexibility and portability of code between different platforms. In the figure, the lower implementation of the function `shade()` belongs to an invasive ray tracer

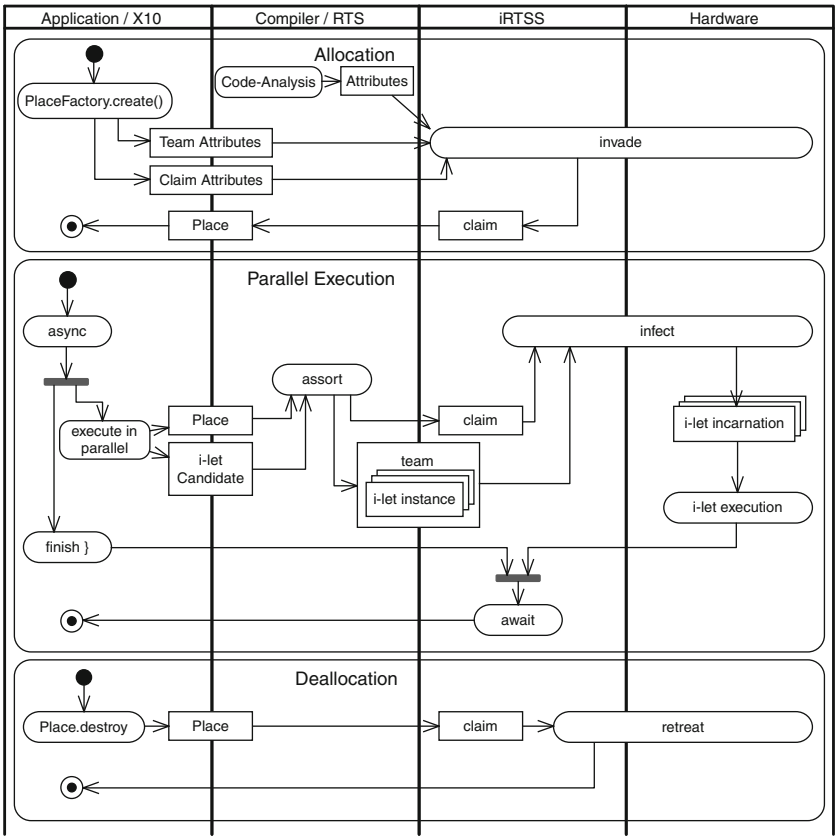


Fig. 11.9 Possible levels ("columns") of abstraction for achieving an integrated cooperative execution of invasive-parallel programs. The activity diagram sketches the flow of control in the use of `invade`, `infect` and `retreat` and shows three different phases of processing: resource allocation, parallel execution and resource deallocation

which first tries to obtain an SIMD array of processors for the computation of the shadow rays and, if successful, runs all intersect computations in parallel on the invaded and then infected array. Note how the invade command specifies the processor type and the number of processors, and the infect command uses higher-order programming[3] by providing a method name as parameter, which is to be applied to all elements of the second parameter, namely the array of data. In case an SIMD processor cannot be obtained, the algorithm tries to obtain another ordinary processor, and uses it for the intersection computation. If this fails also, a sequential loop is executed on the current processor. Note that resource-aware programming here means that the application asks for the availability of processors of a specific type. For the reflected rays, a similar resource-aware computation is shown.

[3] Actually a map construct.

Fig. 11.10 Pseudocode for an invasive ray tracer. The upper code of the shader shows a simple sequential code. The lower code is invasive and relies on resource-aware programming

```
// common code:
trace(Ray ray)
{
   // shoot ray
   hit = ray.intersect();
   // determine color for hitpoint
   return shade(hit);
}

// shade() without invasion:
shade(Hit hit)
{
   // determine shadow rays
   Ray shadowRays[] = computeShadowRays();
   boolean occluded[];
   for (int i = 0; i < shadowRays.length; i++)
      occluded[i] = shadowRays[i].intersect();
   // determine reflected rays
   Ray reflRays[] = computeReflRays();
   Color refl[];
      for (int i = 0; i < reflRays.length; i++)
         refl[i] = reflRays[i].trace();
   // determine colors
   return avgOcclusion(occlusion)
          *avgColor(refl);
}

// shade() using invasion:
shade(Hit hit)
{
   // shadow rays: coherent computation
   Ray shadowRays[] = computeShadowRays();
   boolean occluded[];
   // try to do it SIMD-style
   if ((ret = invade(SIMD,shadowRays.length))
         == success)
      occluded = infect(intersect,shadowRays);
   // otherwise give me an extra core ?
   else if ((ret = invade(MIMD,1)) == success)
      occluded = infect(intersect,shadowRays);
   // otherwise, I must do it on my own
   else
      for (int i = 0; i < shadowRays.length; i++)
         occluded[i] = shadowRays[i].intersect();
   // reflection rays: non coherent,
   // SIMD doesn't make sense
   Ray reflRays[] = computeReflRays();
   Color refl[];
   // potentially we can use
   //    nrOfReflectionRays processors
   ret = invade(MIMD,reflRays.length);
   if (ret == success)
      refl[] = infect(trace,reflRays);
   else
      // do it on my own
      for (int i = 0; i < reflRays.length; i++)
         refl[i] = reflRays[i].trace();
   return avgOcclusion(occlusion)*avgColor(refl);
}
```

```
quadtreeTraversal(v1, v2, v3, v4) {
if (isQuadtreeLeaf(v1, v2, v3, v4}) {
  processLeaf(v1, v2, v3, v4);
} else {
  if (isSmallTree(v1,v2,v3,v4))
     numCores = 0
  else {
     claim = invade(3);
     numCores = claim.length;
  }
  vctr = (v1+v2+v3+v4)/4;

  // last recursive call is
  // always on current processor
  // other recursive calls infect,
  // if processors available
  // and tree big enough
  if (numCores>0) {
     infect(claim[1], quadTreeTraversal(
          (v1+v2)/2, v2, (v2+v3)/3, vctr));
     numCores--;
  }
  else quadTreeTraversal((v1+v2)/2, v2,
                         (v2+v3)/3, vctr);
  if (numCores>0) {
     infect(claim[2], quadTreeTraversal(
          vctr, (v2+v3)/2, v3, (v3+v4)/2));
     numCores--;
  }
  else quadTreeTraversal(vctr, (v2+v3)/2,
                         v3, (v3+v4)/2);
  if (numCores>0) {
     infect(claim[3], quadTreeTraversal(
          (v3+v4)/2, vctr, (v1+v4)/2, v4));
     numCores--;
  }
  else quadTreeTraversal((v3+v4)/2, vctr,
                         (v1+v4)/2, v4);

  quadTreeTraversal(v1, (v1+v4)/2,
                    vctr, (v1+v2)/2);
 }
}
```

Fig. 11.11 Invasive quadtree traversal. The algorithm dynamically adapts to the available resources and the subtree size

The second example (Fig. 11.11) goes one step further into resource-aware programming. The example is a traversal of a quadtree, where the coordinates of the current cell's vertices are parameters to a standard recursive tree traversal method. Leafs, that is, the last recursions are always processed on the current processor. If, however, the tree is "big enough," the first three recursive calls are done in parallel, if processors are available. If not enough processors can be infected, recursive calls are done on the current processor.

Note that the algorithm adapts dynamically to its own workload, as well as to the available resources. Whether a tree is "big enough" to make invasion useful, not only depends on the tree size, but also on the system parameters such as cost of invasion or communication overhead. Resource-aware programming must take such overhead into account when deciding about invasions. Notably, invasion also adds flexibility and fault-tolerance.

The next example is an invasive version of the Shearsort algorithm (Fig. 11.12). Shearsort is a parallel sorting algorithm that works on $n \times m$-grids, for any n (width) and m (height). It performs $(n + m)\cdot(\lceil \log m \rceil + 1)$ steps. An invasive implementation will try to invade an $n \times m$ grid of processors, but will not necessarily obtain all these processors. If it gets an $n' \times m'$-grid, $n' \leq n$, $m' \leq m$, it adapts to these values. Most significantly, it may choose to use the received grid as an $m' \times n'$-grid, rather than an $n' \times m'$-grid.

The pseudocode thus uses invade to obtain an initial row of m' processors, and for each row processor a column of n' additional processors. Note that the invade command specifies the direction of invasion: in the example, SOUTH and EAST. For coarse-grained invasion such as in case of the ray-tracing example, the direction

Shearsort:
– determine optimal values for n and m;
(estimation of free resources)
– Invasion to the south n;
– obtain n' processing elements (PE);
– Invasion from every PE to the east m;
– obtain minimal number of m' PEs;
– unused PEs are freed;
– PEs will handle a total of
$\lceil n \cdot m / (n' \cdot m') \rceil$ keys;
– if $n' > m'$
then
do Shearsort on the $m' \times n'$ grid
else
do Shearsort on the $n' \times m'$ grid

```
program InvasiveShearSorter
  /* Variable declarations */
  int Pinv[M];
  int N_prime, M_prime;
  int keys[N*M];
  /* Parameter declarations */
  parameter M;
  parameter N;
  /* Program blocks */
  M_prime = invade(PE(1,1), SOUTH,
M);
  seq {
    par (i >= 1 and i <= M_prime)
{
      Pinv[i] = invade(PE(i,1),
                       EAST, N);
    }
    N_prime
      = MIN[1 <= i <= M]
Pinv[i];
  /* Free PEs again such that all
arrays have
   same size N_prime */
  par (i >= 1 and i <= M) {
    retreat(PE(i,1), N_prime+1,
Pinv[i]);
  }
  if N_prime > M_prime
    swap(N_prime, M_prime)
  infect columns and rows with Odd-Even
    Transposition Sort
  repeat ⌈log M_prime⌉+1 times
  {
  par (i >= 1 and i <= M_prime) {
    if odd(i) {
      sort in row i the keys
                2*N_prime*(i-1)+1, ...,
2*N_prime*i
      into ascending order }
    else {
      sort in row i the keys
                2*N_prime*(i-1)+1, ...,
2*N_prime*i
      into descending order }
    }

  par (j >= 1 and j <= N_prime) {
    sort in column j the keys
      j, j+2*N_prime, j+4*N_prime, j+6*N_prime ...
    into ascending order }
  par (j >= 1 and j <= N_prime) {
    sort in column j the keys
      N_prime+j, j+3*N_prime, j+5*N_prime,
        j+7*N_prime ...
    into ascending order }
  }/* Here, more invasion is possible:
Check
    whether more resources are available in
    the meanwhile and act appropriately */
 }
}
```

Fig. 11.12 Pseudocode for invasive Shearsort

of invasion is usually irrelevant, but for medium-grained or loop-level invasion, it may be very relevant. Thus, a so-called invasive command space needs to be defined and include a variety of options for invade and infect.

Next, the rows are infected with a transposition sort algorithm, which is used to do a parallel sort in the rows first and then a parallel sort in the columns. These row and column sort phases constitute a round. Rounds are performed $\log m' + 1$ times, and an appropriate subspace of the key space is sorted in parallel in each sequential iteration. In this example, invasion is more fine-grained than in the previous one; here, resource-awareness means that the algorithm adapts to the available grid size, where the initial invasion is based on the problem size.

Invasion cannot only be used to receive the $n' \times m'$-grid. It is also possible to check after every loop execution, i.e., after every round whether the resources requested in the beginning became available in the meantime such that by a further invasion phase the execution can be speeded up, as noted in the pseudocode of Fig. 11.12.

While the previous examples demonstrated coarse-grained and medium-grained invasion, the last example (Fig. 11.13) demonstrates fine-grained invasion at the

Sequential C code:

```
for (i=0; i<T; i++)
  for (j=0; j<N; j++)
    y[i] += a[j] * u[i-j];
```

Code 1 (sequential assembler code):

```
; write input to feedback FIFO of
depth N
1: mov ffo, in0
; set the number of Taps
2: mov r0, N
3: mov r2, 0
; filter coefficient a
4: mul r1, ffo, a
5: add r2, r2, r1
; decrement the tap
6: sub r0, r0,1
; loop N times
7: if zeroflag!=true jmp 4
; get the output
8: mov out1, r2
9: jmp 1
```

Control code (pseudo notation):

```
while (stop!=1) do
  P = invade(N)
  if (P>0) then
    // execute code on P processors
    infect(P, ProgID)
    for (i=0; i<T; i++) do
      Code 2
    end for
    retreat()
  else
    // execute code on one processor
    for (i=0; i<T; i++) do
      Code 1
    end for
  end if
end while
```

Code 2 (VLIW program):

```
add out1 r0 in1, mul r0 in0 a, mov out0 in0
```

Fig. 11.13 FIR filter exploiting loop-level invasion. Sequential C and assembler code is shown left. To the right, the i-let code controlling an invaded TCPA is shown, as well as the assembler code (VLIW) executed on each invaded processing element

loop level. For every iteration of a parallelised loop, a separate processor element may be invaded. To avoid the overhead of i-let incarnation, there is just one controller i-let, which synchronises all the invaded processor elements of a TCPA at a maximal invasion speed of a single clock cycle/processor. Each processor element is infected with "code 2" (Fig. 11.13, right column) and executes the initial loop program in parallel. This kind of invasion is particularly suited for a myriad of nested loop algorithms (loop-level parallelism).

All examples follow a more generic scheme and are presented here to give a better idea of the invasive process (cf. Fig. 11.14). In particular, `invade`, `infect` and `retreat` operate on sets of resources and processes, called "claims" and "teams."

This example also demonstrates the optional integration of exception handling concepts by means of which resource-aware application programs are enabled to reflect on the outcome of claim and team assembly. Handling an "invasion exception" may result in reissuing `invade` with alternate parameter values. Similar concepts hold with respect to the marshalling of a team (i.e., assembly of code and data sections) to fit a selected claim. Note that further origins of invasion exceptions may be the implementations of `invade`, `infect` and `retreat`. At the level of abstraction assumed in Fig. 11.14, this eventually implies that the operating system will be in charge of raising exceptions. Adequate linguistic support for robust resource-aware programming like this comes with the exception handling concept of X10 [18].

```
claim = invade(type, quantity, properties);
if (!useful(claim))      /* unrealisable claim request */
    raise(IMPROPER_CLAIM);

team = assort(claim, code, data);
if (!viable(team))       /* inadmissible team assembly */
    raise(UNVIABLE_TEAM);

infect(claim, team);     /* employ resource(s) */
retreat(claim);          /* clean-up of resource(s) */
```

Fig. 11.14 Pattern of invasive programming (in the programming language C) by adopting an operating system machine level of abstraction. Imagine requests of `invade`, `infect` and `retreat` as "system calls" to an abstract machine, for example, an operating system, while all other primitives execute as part of a run-time system or even an application program by using that machine

Let us conclude with the important remark that true resource-aware programming will not just check the availability of processors. A resource-aware application in general will first of all determine its own needs based on the dynamic work load, then check for available resources of a specific kind and finally infect the obtained resources. The "kind of resource" may include parameters such as permission, speed or even processor temperature. In the background, the operating system and the reconfigurable hardware cooperate to give the application its desired resources in the most efficient and appropriate way.

11.3 Expected Impact and Risks

In the following, we summarise the expected benefits and impact factors we see for a broad and multi-disciplinary research in invasive computing but also potential risks.

Impact. We have motivated invasive computing as a means to cope with the exploding complexity of future massively parallel MPSoCs with the major call to provide scalability, higher resource utilisation, higher efficiency, and also higher speed as compared to applications with statically partitioned allocation of resources. We intend to achieve these goals on the basis of resource-aware programming and new reconfigurable MPSoC architecture inventions. Both revolutionary architectures as well as new programming concepts in synergy shall provide a boost in efficiency and usability of future MPSoC platforms that are expected to contain 1,000 and more processors.

The areas in which research in invasive computing might create a substantial impact are summarized as follows:

- *Processor Architecture of Future Multi-Core Systems*: Even if we will not be able to compete in our design concepts and demonstrators with high-end

processor designs as developed by teams of 100 and more designers at processor companies such as Intel and AMD, we believe that some of our architectural inventions will influence their way of how to design large processor systems in the future. For example, without research and inventions on previously non-common RISC architectures performed at universities such as by Hennessy and Patterson, the chip design companies might still produce other types of processors.

- *Design Environments for Programming Parallel Many-Processor Systems*: Similarly, our paradigm of invasive programs and resource-aware programming will have an impact on future programming languages and programming environments for the development of parallel programs.
- *Design of Parallel Algorithms*: Even more, the idea of invasive algorithm design will influence the development of parallel algorithms as well. Never before algorithm designers had the opportunity to dynamically adapt an algorithm's behaviour and parallelism to the dynamic workload and the dynamic availability of resources.

Risks. Nevertheless, we do not conceal that our challenging goals might also hide some risks:

- *Acceptance of Resource-aware Programming*: At a first look, resource-aware programming seems questionable and counter-productive when looking at modern software-technological principles: High-level languages as well as operating systems have, for good reason, more and more abstracted away from specific hardware details or resource politics. Instead of offering progress, resource-aware programming thus sounds contradictory and a step back into the past when looking at the achievements of modern programming languages, which abstract away from specific architectural details.
- *Cost in Terms of Time and Area*: Increasing the non-determinism by self-organised algorithm execution when allowing programs to control hardware resources directly might naturally lead to cases with lower performance and worse resource utilisation than statically mapped and scheduled applications, of course as the time to invade and retreat from resource occupations produces overhead. Any comparison of cost and speed-up against a statically mapped non-invasive algorithm must therefore be done carefully and, to be fair, consider the case of overload situations: Here, due to invasion, resources will be freed which enables other applications to dynamically claim more resources than in a statically partitioned case between several competing applications. If the degree of parallelism of considered applications is varying in time, also speed-up will result naturally over static processor partitions apart from higher resource utilisation, savings of power and fault-tolerance. A natural scenario of invasive computing is therefore that not only one but several programs are simultaneously trying to invade a common pool of resources.

In summary, it is evident that there is a price to pay to exploit the benefits of invasive computing. Therefore, it needs to be investigated carefully where the border of centralised control versus invasive control reaches its greatest benefit

and how a maximum of abstraction can be maintained even for resource-aware computing. The goal of this survey was to give an overview into the fascinating emerging paradigm of invasive computing that might solve many problems of MP-SoC architectures and their programming with more than 1,000 cores for the years 2020 and beyond. Here, only the basic principles and fields of required research could be drafted.

Acknowledgements We thank the following people for their support (in alphabetical order): Dr. Tamim Asfour, Dr. Lars Bauer, Prof. Jürgen Becker, Prof. Hans-Joachim Bungartz, Prof. Rüdiger Dillmann, Prof. Michael Gerndt, Dr. Frank Hannig, Sebastian Harl, Dr. Michael Hübner, Dr. Daniel Lohmann, Prof. Peter Sanders, Prof. Ulf Schlichtmann, Prof. Marc Stamminger, Prof. Walter Stechele, Prof. Rolf Wanka, Dr. Thomas Wild and all of their scientific staff members. Finally, we would like to express our sincere gratitude to the German Research Foundation (DFG) to establish its collaborative research center TCRC89 on the topic of invasive computing, see http://www.invasic.de

References

1. Rabaey, J.M., Malik, S.: Challenges and solutions for late- and post-silicon design. IEEE Design and Test of Computers 25(4), 296–302 (2008). DOI http://dx.doi.org/10.1109/MDT.2008.91. URL http://dx.doi.org/10.1109/MDT.2008.91
2. Corporation, N.: NVIDIA Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_-Compute_Architecture_Whitepaper.pdf (2009)
3. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28, 39–55 (2008)
4. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. ACM Transactions on Graphics 27(3), 1–15 (2008). DOI http://doi.acm.org/10.1145/1360612.1360617
5. Pham, D., Aipperspach, T., Boerstler, D., Bolliger, M., Chaudhry, R., Cox, D., Harvey, P., Harvey, P., Hofstee, H., Johns, C., et al.: Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor. IEEE Journal of Solid-State Circuits 41(1), 179–196 (2006)
6. Hannig, F., Dutta, H., Teich, J.: Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology. International Journal of Embedded Systems 2(1/2), 114–127 (2006)
7. Kissler, D., Hannig, F., Kupriyanov, A., Teich, J.: A Highly Parameterizable Parallel Processor Array Architecture. In: Proceedings of the IEEE International Conference on Field Programmable Technology (FPT), pp. 105–112. Bangkok, Thailand (2006)
8. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., et al.: An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In: SolidState Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International, pp. 98–589 (2007)
9. Feautrier, P.: Automatic Parallelization in the Polytope Model. Tech. Rep. 8, Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex (1996)
10. Hannig, F., Teich, J.: Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals. In: Proceedings of the 15th IEEE International

Conference on Application-specific Systems, Architectures, and Processors (ASAP), pp. 17–27. Galveston, TX, USA (2004)
11. Thomas, A., Becker, J.: New adaptive multi-grained hardware architecture for processing of dynamic function patterns. it - Information Technology 49(3), 165–173 (2007)
12. Teich, J.: Invasive Algorithms and Architectures. it - Information Technology 50(5), 300–310 (2008)
13. Cybenko, G.: Dynamic load balancing for distributed memory multiprocessors. Journal of Parallel and Distributed Computing 7, 279–301 (1989)
14. Boillat, J.E.: Load balancing and poisson equation in a graph. Concurrency: Practice and Experience 2, 289–313 (1990)
15. Rabani, Y., Sinclair, A., Wanka, R.: Local divergence of Markov chains and the analysis of iterative load-balancing schemes. In: Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 694–703 (1998)
16. Sanders, P.: Randomized priority queues for fast parallel access. Journal Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures 49, 86–97 (1998)
17. Sanders, P.: Asynchronous scheduling of redundant disk arrays. IEEE Transactions on Computers 52(9), 1170–1184 (2003). Short version in 12th ACM Symposium on Parallel Algorithms and Architectures, pages 89–98, 2000
18. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielsstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 519–538. ACM (2005)

Index

M. Hübner and J. Becker (eds.), *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, DOI 10.1007/978-1-4419-6460-1,