Modern development methods and tools for embedded reconfigurable systems: A survey
Lech Jóźwiak a, Nadia Nedjah b,*, Miguel Figueroa c

a Eindhoven University of Technology, The Netherlands
b State University of Rio de Janeiro, Brazil
c University of Concepción, Chile

Article info
Article history:
Received 6 May 2009
Received in revised form
9 June 2009
Accepted 25 June 2009

Keywords:
Embedded systems
Reconfigurable systems
Development tools
Development methods

Abstract
Heterogeneous reconfigurable systems provide drastically higher performance and lower power consumption than traditional CPU-centric systems. Moreover, they do it at much lower costs and shorter times to market than non-reconfigurable hardware solutions. They also provide the flexibility that is often required for the engineering of modern robust and adaptive systems. Due to their heterogeneity, flexibility and potential for highly optimized application-specific instantiation, reconfigurable systems are adequate for a very broad class of applications across different industry sectors. What prevents the reconfigurable system paradigm from a broad proliferation is the lack of adequate development methodologies and electronics design tools for this kind of systems. The ideal would be a seamless compilation of a high-level computation process specification into an optimized mixture of machine code executed on traditional CPU-centric processors and on the application-specific decentralized parallel data-flow-dominated reconfigurable processors and hardware accelerators. Although much research and development in this direction was recently performed, the adequate methodologies and tools necessary to implement this compilation process as an effective and efficient hardware/software co-synthesis flow are unfortunately not yet in place. This paper focuses on the recent developments and development trends in the design methods and synthesis tools for reconfigurable systems. Reconfigurable system synthesis performs two basic tasks: system structure construction and application process mapping on the structure. It is thus more complex than standard (multi-)processor-based system synthesis for software-programmable systems that only involves application mapping. The system structure construction may involve the macro-architecture synthesis, the micro-architecture synthesis, and the actual hardware synthesis. Also, the application process mapping can be more complicated and dynamic in reconfigurable systems. This paper reviews the recent methods and tools for the macro- and micro-architecture synthesis, and for the application mapping of reconfigurable systems. It puts much attention to the relevant and currently hot topic of (re-)configurable application-specific instruction set processors (ASIP) synthesis, and specifically, ASIP instruction set extension. It also discusses the methods and tools for reconfigurable systems involving CPU-centric processors collaborating with reconfigurable hardware sub-systems, for which the main problem is to decide which computation processes should be implemented in software and which in hardware, but the hardware/software partitioning has to account for the hardware sharing by different computation processes and for the reconfiguration processes. The reconfigurable system area is a very promising, but quite a new field, with many open research and development topics. The paper reviews some of the future trends in the reconfigurable system development methods and tools. Finally, the discussion of the paper is summarized and concluded.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction................................................................. 2
2. Need of enabling development technology.......................... 3
3. Development of embedded RC systems.............................. 4

* Corresponding author.
E-mail addresses: L.Jozwiak@tue.nl (L. Jóźwiak), nadia@eng.uerj.br (N. Nedjah), miguel.figueroa@udec.cl (M. Figueroa).

0167-9260/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.vlsi.2009.06.002
1. Introduction

The recent spectacular progress in modern micro- and nano-electronic technology has enabled the implementation of complex information-processing systems on single chips and created a big push towards the development of various kinds of application-specific embedded and high-performance systems. Traditional applications can now be served much better and numerous new sorts of systems have become technologically feasible and economically justified, especially for applications that require miniaturization, high-performance, autonomous computing, and wireless or distant communication.

On the other hand, however, this spectacular progress introduced unusual technology and system complexity. Increasingly complex and sophisticated embedded systems in telecommunications, consumer electronics, advanced machinery, signal, image and video processing, medical equipment, instrumentation, avionics, the military area and virtually all other areas of human activity are required to reliably perform real-time computations to extremely tight schedules with energy, power and area efficiency never demanded before. The computational demands of high-performance systems for scientific and engineering computation also grow rapidly. Moreover, the system and technology complexity, as well as the rapidly growing proliferation of microelectronic systems in the general consumer markets result in serious development and implementation challenges, such as the rapidly growing design productivity gap and development costs. There is a general consensus that the progress in microelectronic technology alone cannot guarantee satisfaction of the growing computational demands, physical requirements and economical challenges of most modern applications.

Opportunities created by modern technology can effectively be exploited only through adequate usage of efficient application-specific system architectures and circuit implementations exploiting more adequate concepts of computation, storage and communication. This requires effective and efficient design methods and Electronic Design Automation (EDA) tools for synthesizing the actual high-quality hardware platforms implementing the architectures, and for efficient mapping of the applications onto hardware platforms. The new system architectures and computation, storage and communication concepts, as well as, the new design methods and EDA-tools need to adequately address the problems discussed above.

Numerous embedded system and scientific computation projects have demonstrated that heterogeneous reconfigurable systems, exploiting a mixture of traditional CPU-centric instruction-stream-based processing and decentralized parallel application-specific data-dominated processing, provide drastically higher performance and lower power consumption than traditional CPU-centric systems. Moreover, they do it at much lower costs and shorter times to market than non-reconfigurable hardware solutions. They also provide the flexibility that is often required for the engineering of modern robust and adaptive systems. Due to their heterogeneity, flexibility and potential for highly optimized application-specific instantiation, reconfigurable computing (RC) systems are adequate for a very broad class of applications across different industry sectors. For many application areas, the heterogeneous parallel reconfigurable systems can be even several orders of magnitude faster, while consuming several times less power than the traditional CPU-based systems [1].

A reconfigurable system is a system whose sub-systems and/or sub-system configurations can be changed or modified after fabrication to (better) serve a certain purpose. Reconfigurable computing designates computing based on computers whose processing units, memory units, communication channels and/or their composition can change function and/or (spatial) configuration after fabrication, prior or during the run-time of a particular set of applications, application, program or part of a program. The term “reconfigurable system” has a broad meaning, and effectively subsumes the concepts of “reconfigurable computing”. Further in this paper the term “reconfigurable system” (or “RC system”) will be used in its narrow meaning to designate the “reconfigurable computing system”. The core component of a reconfigurable computing system is programmable hardware that can be temporarily (partly) customized/programmed to (better) operate a specific computation or set of computations.

In our previous survey paper on modern architectures of embedded reconfigurable systems [1], the basic definitions, concepts and features of reconfigurable systems are discussed, as well as their role, the purposes they can serve and applications that can most significantly benefit from them. Also, a comparison of hardwired systems, RC systems and CPU-centric systems is made, and some main concepts of an effective and efficient reconfigurable computing are discussed. Moreover, a classification of RC systems is introduced and their various architecture classes...
are overviewed. Finally, some major drivers and requirements of recent and future developments in the modern RC system area are discussed, and the recent and future development trends in the RC architectures are overviewed.

This paper focuses on modern development methods and tools for embedded reconfigurable systems, and the architecture aspects of reconfigurable systems will be considered here only to the degree necessary for understanding of its discussion of the development methods and tools. More information on RC system architectures can be found in our previous paper devoted to them [1]. The reconfigurable system area is a very promising, but quite a new field. New opportunities have been opened for this field through the introduction of the system-on-a-chip technology, and a big progress has been made in the recent years. Many different reconfigurable devices and computers have become commercially available. Nevertheless, multiple aspects of the RC systems, their development and their supporting tools still belong to the open research or development topics.

2. Need of enabling development technology

What prevents the reconfigurable system paradigm from becoming the actual third main practically used paradigm, in parallel to the traditional CPU-centric programmable processors and non-programmable hardware, is the lack of adequate development methodologies and EDA-tools for this kind of systems. The ideal would be a seamless compilation of a high-level computation process specification (e.g. in MATLAB/Simulink, C, or C++) into an optimized mixture of machine code executed on traditional CPU-centric processors and on the application-specific decentralized parallel data-flow dominated reconfigurable processors and hardware accelerators. Although much research and development in this direction was recently performed, the adequate methodologies and tools necessary to implement this compilation process as an effective and efficient HW/SW co-synthesis flow are unfortunately not yet in place. Currently, the algorithms must be analyzed by experts supported by profiling tools, and their computation and data-intensive parallelizable parts selected for the (re-)configurable parallel hardware implementation. These parts have to be then tackled by HW/SW or HW designers who either develop suitable RTL-level hardware architectures manually or try to convert them into such architectures using the emerging, but still limited, behavioural compilation. Subsequently, from the register transfer level (RTL) specification, the actual hardware is synthesized using traditional logic synthesis tools that, unfortunately, not always match well the specifics of the reconfigurable parallel architectures and of the modern implementation technologies, and using the technology-specific mapping and physical synthesis tools. There is a lot of inefficiency in this largely manual and inconsistent RC system development process.

To drastically reduce this inefficiency, effective uniform synthesis flows and related EDA-tool-sets should be developed for the HW/SW compilation of high-level behavioural specifications (e.g. in MATLAB/Simulink, C or C++) into their highly optimized application-specific parallel heterogeneous reconfigurable HW/SW implementations. This compilation process should include both an adequate application-specific reconfigurable hardware platform synthesis, as well as a corresponding compilation of high-level algorithms into their highly optimized code executed on the synthesized application-specific reconfigurable parallel hardware platform. Its result will be a heterogeneous system with the most critical computation- and data-intensive parts (computation kernels) implemented in the application-specific reconfigurable fabric. One should not forget that the hardware of the application-specific instruction set extensions, co-processors or accelerators must actually be synthesized, and the satisfaction of the parametric requirements of applications very much depends on the quality of the hardware synthesis process. The co-processors, accelerators, intellectual property (IP) cores and instruction set extensions can only be as good as the actual hardware implementing them. Therefore, a high-quality technology-aware multi-objective hardware synthesis chain is of primary importance for the success of the reconfigurable system paradigm. The hardware synthesis chain should cover the high-level, RTL-level, logic-level and physical-level synthesis, and account for performance, area and power consumption requirements. The consistency, tight coupling and high quality of the particular tools in this chain are crucial for the actual satisfaction of the parametric requirements of the demanding embedded and high-performance applications. Such a high-quality uniform hardware/software synthesis flow and related EDA-tool-set do not exist yet, but much work has been performed in this direction recently both in the academia and industry, and there is a substantial body of ongoing research and development.

This paper focuses on the recent developments and development trends in the design methods and synthesis tools for reconfigurable systems. Reconfigurable system synthesis performs two basic tasks: system structure construction and application process mapping on the structure. It is thus more complex than standard software-programmable (multi-)processor-based system synthesis that only involves application mapping. Moreover, the reconfigurable system architecture is usually a heterogeneous multi-processor architecture, involving one or more standard or application-specific software-programmable processors, and some hardware (re-)configurable (co-)processors or accelerators. Depending on the configuration generality (abstraction level) of the reconfigurable hardware resources, the system structure construction may involve the macro-architecture synthesis, the micro-architecture synthesis, and the actual hardware synthesis, which involves RTL-level optimization, logic synthesis, technology mapping and physical synthesis. For instance, if the configuration is very general and performed at a low-level of system/hardware abstraction, as e.g. in standard field programmable gate arrays (FPGAs), then the structure construction process is complex and involves all the architecture and hardware synthesis steps mentioned above. If the configuration is very specific and performed at a high-level of system abstraction (e.g. RTL-level or higher one), then the structure construction process can be much simpler and limited to only a specific architecture synthesis or modification. Also, the application process mapping can be more complicated and dynamic in reconfigurable systems.

Currently, the most technologically viable and economically justified reconfigurable system solutions tend to include:

- hardware systems implemented with various sorts of FPGAs and other kinds of programmable logic;
- hardware/software systems based on (re-)configurable ASIPs;
- multi-processor hardware/software systems involving one or more (possibly different) CPU-centric (general purpose/signal/multimedia) processors collaborating with one or more reconfigurable hardware sub-systems implemented e.g. using platform FPGAs or system-on-chip (SoCs) with embedded FPGAs;
- systems implemented with coarse-grained reconfigurable hardware (different than ASIPs);
- systems representing various mixtures of the above solutions.

Due to uniform character, the execution models and design automation are the easiest for the first two of the above kinds of
systems. In consequence, the reconfigurable hardware systems and ASIP-based systems seem to be the most popular. For the reconfigurable hardware systems the uniform hardware execution models and standard programmable logic synthesis flows and tools can be used (possibly with some minor extensions), as delivered by vendors of the programmable logic devices and third-party EDA-tools for those devices. In this paper we will therefore limit the discussion of the methods and tools for the purely hardware-implemented RC systems to the necessary minimum.

For the systems based on the (re-)configurable ASIPs, standard software execution models and standard software development tools can be used, supplemented with tools for instruction set extension/configuration and extendable/configurable compilers. In this paper, we will put much attention to the currently hot topic of ASIP instruction set extension and will also discuss the related compilation issues. For instruction set extension, the application code can be considered with different granularity, which results in fine-grained and coarse-grained instructions. The fine-grained approach aims at identification of the most promising patterns composed of several basic operations to be implemented as custom instructions that process and produce relatively small amounts of data. The fine-grained instructions are implemented in reconfigurable functional units internal or closely coupled to the CPU of the basic processor. The coarse-grained approach aims at the implementation of the whole timing- or energy-critical procedures, complex loops or stream-based processing blocks as custom instructions that may process and produce substantial amounts of data. The coarse-grained instructions, depending on their granularity and other factors, are typically implemented as internal, closely or loosely coupled data and control-path segments or loosely coupled co-processors. The coarse-grained approach often requires solving of a substantial application parallelisation and high-level synthesis (scheduling and binding) problems, before the actual RTL-level and circuit-level hardware synthesis for the custom instructions. The coarse-grained approach facilitates streaming applications, such as (de-)coding, encryption, filtering and various transforms in signal and multimedia processing, etc. It can result in substantial speedups and energy savings, but its generality is limited. A coarse-grained instruction serves one particular application and one particular computation process in a particular system in most cases. The fine-grained approach is much more general and can be successfully used for many applications and in many computation processes of a particular system, but its speedups and energy savings are lower for typical coarse-grained applications. Consequently, the best solution is often a heterogeneous (hybrid) approach consisting of a mixture of the above two. The paper will review the recent methods and tools related to all the three approaches.

The paper will also discuss the methods and tools for reconfigurable hardware/software systems involving CPU-centric processors collaborating with reconfigurable hardware subsystems. In these systems the coarse-grained approach is typically used, and the main problem is to decide which computation processes should be implemented in software (SW) and which in hardware (HW), a problem known as HW/SW co-design (partitioning). In the case of reconfigurable hardware, the co-design problem solution has, however, to account for the possibility of reconfigurable hardware sharing by different computation processes and for the reconfiguration processes. The solution methods are here thus somewhat different than for the standard co-design problem, and the paper will briefly review them. Systems based on coarse-grained reconfigurable hardware (different than ASIPs) are much less popular, because coarse-grained hardware is less flexible, and the development automation of such systems requires very specific hardware configuration and software compilation tools. Nevertheless, they can deliver high speedups and energy savings to applications involving word-level computations and parallelism, and populated with operations implemented in the coarse-grained hardware, with much lower silicon area overhead than systems based on fine-grained reconfigurable hardware. The paper will also briefly discuss the emerging commercial tools and pay some attention to the hot issue of energy efficiency.

The reconfigurable system area is a very promising, but quite a new field. New opportunities have been opened for this field through the introduction of the system-on-a-chip technology, and a big progress has been made in the recent years. Many different reconfigurable devices and computers have become commercially available. Nevertheless, multiple aspects of the RC systems, their development and their supporting tools still belong to the open research or development topics. The paper will therefore review the future trends in the RC system development methods and tools. Finally, the discussion of the paper will be summarized and concluded.

3. Development of embedded RC systems

The development of reconfigurable embedded systems has to be discussed in the broader context of general embedded system development. An embedded system (unlike a stand-alone computer) is an inseparable part of a certain larger (embedding) system. It is built into this larger system and especially designed or adapted to serve a specific aim in this system. The aim it serves is strictly related to the application or application class the embedding system serves. Consequently, an embedded system is application-specific: it repeatedly executes particular computation processes required by its application or application class, and it has to be especially designed or customized to adequately serve the execution of these particular computation processes, as well as adequately satisfy specific application constraints and objectives. Typically, embedded systems are reactive real-time systems that involve in their implementation various mixtures of digital and analog hardware, as well as, hardware-dependent and embedded software. They are very difficult to develop. They must appropriately react in real-time to the signals from their surroundings and to be fine-tuned to particular applications by satisfying application-specific constraints and objectives related to such attributes as functional behaviour, reaction speed and throughput, power and energy consumption, geometrical dimensions, price, etc.

Moreover, the operation domains, roles and complexity of the embedded microelectronic systems more and more resemble the operation domains, roles and complexity of (parts of) the (intelligent) life organisms or organized populations of such organisms. In [2], the authors proposed to extensively exploit this parallel for the development of embedded systems. He formulated the hypothesis that future embedded microelectronic systems should have characteristics that resemble the characteristics of (intelligent) life organisms or their organized populations. Consequently, the basic concepts, principles, functional and structural organization, etc. of embedded microelectronic systems should resemble these of (intelligent) life organisms or their populations. “Resemble” does not of course mean to be identical. Although in microelectronic system development we should be much more inspired by life organisms than until now, we have also to account for the differences such as, for instance, the nature of the life systems and technological systems, their materials, implementation technologies, etc.
In particular, similarly to a real organism or brain, a life-inspired embedded system should be heterogeneous, highly decentralized and composed of largely autonomous sub-systems (organs or centres), which are diverse, have their own particular aims and are optimized for these sub-systems aims. The sub-systems should be adequately (hierarchically) organized, interconnected with an appropriate network of communication channels, properly coordinated, and should adequately collaborate with each other to synergistically achieve the global system aims. To avoid the memory and communication bottlenecks, performance and energy inefficiency, etc.:

- information, intelligence, memory and computation resources of an embedded system should be properly distributed over all its sub-systems,
- effective communication should be provided between the sub-systems,
- effective application-specific operations should be used to implement the system’s required behaviour, and
- parallel processing should be extensively applied.

This requires, among others, autonomous heterogeneous sub-systems working in parallel and involving efficient application-specific operations implemented in hardware, local distributed memories in the sub-systems, (more) global multi-port memories for sharing data and communication between the sub-systems, memory-centric processing for massive data, adequate mixture of effective and efficient application-specific communication schemes and mechanisms of all kinds dependent on the characteristics of the required communication, and (re-)configurable hardware and system approach to implement the application-specific highly parallel processing and application-specific communication schemes and mechanisms effectively and efficiently, as well as to ensure the flexibility required by many modern applications.

Summing up, for an adequate implementation of many modern complex and demanding embedded systems, heterogeneous (re-)configurable architectures are required that involve an adequate mixture of programmable processors with (re-)configurable hardware accelerators or hardwired accelerators, as well as, acceleration and mixed-grain (re-)configurability implemented at multiple levels of the system hierarchy. In recent years, we have seen a growing variety of such heterogeneous architectures. Such architectures enable realization of the effectiveness, efficiency and flexibility required by many modern embedded applications, as well as an efficient resolution of multiple tradeoffs. The ability to customize the architecture to the required computations and running of (parts of) applications on adequate (re-)configurable hardware accelerators enables the increase of the computation speed-up to several tens of times, as well as a substantial decrease of their power and energy consumption [1].

Due to the autonomous and/or wireless character of the majority of new applications and their growing complexity on the one hand, and increased leakage power of modern nanometre microelectronics implementation technologies on the other, the power and energy consumption issues are becoming more and more serious. Moreover, due to the rapidly growing silicon and system complexity, both the hardware and software of future chips tends inherently to be less reliable and more sensitive to noise and interference from the environment. However, embedded and autonomous systems play an extremely remarkable role in today’s life and are used more and more commonly in virtually all fields of human activity, in all sorts of technical, social and biological systems, in more and more important and demanding applications. They are even implanted in our bodies. Many of them are used in safety critical applications that impose extremely high-quality requirements. Our life is to a higher and higher degree dependent on their adequate operation and the expectations regarding their quality grow rapidly. We certainly cannot tolerate that the future systems will be of lower quality. Consequently, the development of future embedded systems should aim at the total multi-objective maximization of their quality, with a special focus on their guaranteed performance, robustness and dependability, as well as on energy and power issues. It should adequately exploit tradeoffs between important system characteristics and result in more coherent, compact, comprehensive, reliable, robust and lower-cost solutions.

According to [3,4] system design is actually about a definition of the required quality, in the sense of a satisfactory answer to the following two questions:

- What (new or modified) quality is required?
- How can it be achieved?

Consequently, quality-driven design methods and tools are necessary to ensure that our contemporary and future systems will represent the actually required quality. In order to bring the quality-driven design into effect, quality has to be modelled, measured and compared. To enable it, Jóźwiak [3] proposed the following generic quality definition: Quality of a purposive systemic solution is its total effectiveness and efficiency in solving the real-life problem the solution is required for. Effectiveness is the degree to which a solution attains its goals. Efficiency is the degree to which a solution uses resources in order to realize its aims. In turn, the effectiveness and efficiency can be expressed in terms of measurable parameters, and in this way quality can be modelled and measured. However, even with this new definition not all difficulties are overcome. Design does not concern the reality as it is, but as it can be imagined and will possibly be realized. Both the recognition of a design problem and the nature of its solution are subjective to a high degree. To arrive at the high-quality designs, the subjectivity has to be selectively and appropriately limited, when enabling the creativity exploitation at the same time. Furthermore, the electronic system design problems are complex, multi-aspectual, dynamic, and ill-structured. This last property means that there is no definitive formulation of the problem, any problem formulation may be inconsistent, formulations of the problem are solution dependent, proposing and considering solutions is a means for understanding the problem, and consequently, there is no definitive solution to the problem. Moreover, it is very difficult to find precise relations between various aspects of the effectiveness and between the different forms of energy and matter used to attain the system’s aim, and even more difficult to express them as one uniform measure. Also, there is a trade-off as well between effectiveness and efficiency as among different other aspects. Finally, the required quality or its perception can change in time. Therefore, quality cannot be well defined, but it can and should be modelled. Design space exploration with usage of the well-structured quality models makes possible limiting the scope of subjective design decision-making and enlarging the scope of reasoning-based decision-making with open and rational procedures that can be computerized. According to [3,4], system design is an evolutionary quality engineering process. It starts with an abstract, imprecise, incomplete and possible contradictory initial quality model (initial requirements), and tries to transform the initial model into a concrete, precise, complete, coherent and directly implementable final model. The initial model is abstract and involves mostly some behavioural and parametric
characteristics. The final model defines the system's structure explicitly. This structure supports the system's behaviour and satisfies the parametric requirements. This evolutionary quality engineering process decomposes the total design problem into several issues. For each issue, many various alternative solutions are typically possible. To guide the search for the most promising alternatives, we can and should construct adequate quality models for particular issues, composed of some selected and abstracted functional, structural and parametric requirements extracted in an appropriate manner from the total quality model of the system under design.

In particular, quality can be modelled in the form of multi-objective decision models, being partial and abstract (reduced to the relevant and/or feasible concerns and precision levels) models of the required quality, expressed in the decision-theoretical terms. Multi-objective decision models, together with methods and tools for the estimation of the design parameters of these models related to the relevant design aspects and performances, enable application of the multi-objective decision methods for construction, improvement and selection of the most promising solutions [3–7].

The main concepts of the quality-driven design [3,4] can be briefly summarized as follows:

- designing of top-quality systems is the aim of a design process;
- quality is modelled and measured to realize this aim, i.e. to enable invention and selection of the most promising design alternatives and quality improvement;
- quality models are considered to be heuristics for setting and controlling the course of design, and as such, they are also a subject to design and change;
- the design process is evolutionary and it basically consists of:
  - constructing the tentative quality models,
  - using them for constructing, selecting and improving the tentative solutions,
  - analyzing and estimating them directly and through analysis of the resulting solutions,
  - improving them and using again, etc.

Summing up, methodologies for the modern embedded RC system development should target-specific sub-sets of the heterogeneous architectures advised by the paradigm of life-inspired systems and be specific realizations of the quality-driven design process briefly introduced above. Moreover, they should exploit an adequate mixture of the design reuse at the system and sub-system levels with the automatic synthesis starting from the system level. This is necessary to efficiently develop the demanding contemporary and future complex systems. Design reuse is a very important aspect of the quality-driven design, because it simultaneously enhances the system quality (due to the "maturity" of the reused designs) and the development and fabrication efficiency (due to elimination of some development or fabrication phases). Generic system solutions, and especially the generic system platforms and architecture templates belong to the major enablers of such adequate mixture of design reuse and automatic synthesis. They can be reused and adapted to (better) suit a particular application. For a given application, the generic platform and some of its modules are reused and adequately instantiated to appropriately serve the application, but also some new application-specific modules may be added. The general form constrains the solution search space to such a degree that the construction of particular solution instances for particular applications can be performed (semi-)automatically, through an appropriate instantiation of the generic architecture platform or template, and computation process scheduling and mapping on the instance of the platform or template [3,4]. More general platforms can adequately support larger application classes, which makes them more economically justified, as their non-recurring engineering (NRE) costs can be shared by multiple applications. On the other hand, more specific platforms can be more effective and efficient in serving a particular application. In [1] some main ideas of an effective and efficient reconfigurable computing are discussed that include, among others, generalization of the design problem to a class of problems having similar characteristics and requirements, for which adequate system solutions can also be similar, and development of a generic solution (reconfigurable computation platform or architecture template) for this class of problems or reuse of an existing generic solution that is able to effectively and efficiently implement the computation patterns and operations required in a selected reconfigurable hardware fabric. The generic platform (template)-based system approach gains popularity both among researchers and in the industry [4–18]. Moreover, many different generic (re-)configurable platforms and templates exist, including several (platform) FPGAs, coarse- and mixed-grain platforms and (re-)configurable ASIP processors. The generic platform-based system approach to embedded RC system development is thus well motivated both from the technological and economical viewpoint, and therefore we will focus on the platform-based approach further in this paper.

For the development of the generic system platforms (templates) regular system and circuit design methodologies and electronic design automation tools can be used. However, being generic solutions they are more difficult to develop than their particular instantiations. In addition to the regular system design issues, their developers have to decide their generality and instantiation concepts, as well as to develop and implement all their instantiation mechanisms. Moreover, they require methods and EDA-tools supporting their instantiation and application mapping.

4. Platform-based RC system development

Quality-driven platform-based RC system development is model based [3,4]. Design requirements represent a general model of the required quality that models the design problem at hand through the imposition of a number of constraints and objectives in relation to the acceptable or preferred problem solutions. It is thus an abstract model of a solution to the problem. Since it limits the space of the acceptable or preferred solutions to only a certain degree, it models many solutions concurrently. Each of the solutions fulfills all the hard constraints of the model, but different solutions can satisfy its objectives to various degrees. It is possible to distinguish three kinds of requirements: functional, structural, and parametric. Requirements of each of the three classes impose limitations on the structure of a required solution, but they do it in different ways. Structural requirements define the acceptable or preferred solution structures directly, by limiting them to a certain class or imposing a preference relation on them. Parametric requirements define the structures indirectly, by requiring the structures to have specific physical, economic or other properties (described by values of some parameters) that fulfill given constraints and satisfy stated objectives. Functional requirements also define the structures indirectly, by requiring the structures to expose a certain externally observable behaviour that realizes the required behaviour.

Quality-driven RC system synthesis is an evolutionary quality engineering process. It starts with an abstract, imprecise, incomplete and possible contradictory initial quality model of the required quality (initial requirements) and transforms the
initial model into a concrete, precise, complete, coherent and directly implementable final model. The initial model is more abstract and involves mostly some behavioural and parametric characteristics. The final model defines the system’s structure explicitly. This structure supports the system’s behaviour and satisfies the parametric requirements. This evolutionary quality engineering processes applies the problem-solving framework of heuristic search and decomposes the total design problem into several issues. For each issue, many various alternative solutions are typically possible. For each issue, we can construct some issue’s quality models, composed of some selected and abstracted functional, structural and parametric requirements extracted in an appropriate manner from the total quality model of the considered system. In particular, the issue’s decision model can be constructed as a base for decision-making in the scope of a certain issue. The decision model is a partial (reduced to only certain concerns) and abstract (reduced to the necessary and/or possible precision level) model of the required quality, expressed in the decision-theoretical terms. Decision models and design parameter estimators enable application of the multi-objective decision methods for construction, improvement and selection of the most promising solutions. To be acceptable, the decision models must account for all system characteristics relevant to a particular issue. They must specify preferences of values of all the characteristics, expressed by hard constraints, objectives, and trade-off information. This can be done in several ways, and the choice of a specific formulation depends of the particular problem at hand [5,8,19]. Design decision models constructed in such a way enable evaluation of solutions, their comparison and the exploitation of tradeoffs. Through providing data for the multi-criteria decision-making, such design decision models make it possible to apply multi-criteria decision methods for invention and selection of solutions that are “totally” optimal.

Quality-driven design space exploration basically consists of the alternating phases of exploration of the space of abstract models of the required quality and exploration of the space of the more concrete issue’s solutions obtained with the selected quality models. Quality-driven design space exploration involves:

- problem analysis and modelling;
- creation of potential solutions based on the problem models;
- selection of preferred solutions; and
- analysis of a particular design process iteration and the solutions created and selected in this iteration in order to accept one of them if it is satisfactory and finish the design process or to start a new iteration of the design space exploration.

If it happens to be too difficult to solve a certain problem directly, it is decomposed into simpler sub-problems, the sub-problems are solved using the same approach to the design space exploration, and solution of the problem is composed from the sub-problem solutions. In result of the design space exploration, the system is defined as an appropriate decomposition into a network of sub-systems. Each sub-system solves a certain sub-problem. All the sub-systems cooperating together solve the system design problem by exposing the external aggregate behaviour and characteristics which match the required behaviour and characteristics. This way the design process breaks down a complex system defined in abstract and non-precise terms into a structure of cooperating sub-systems defined in more concrete and precise terms, which are in turn further broken down to simpler sub-systems that can be directly implemented with the elements and sub-systems at the designer’s disposal.

The platform-based RC system synthesis involves the following major steps: problem analysis and system requirement specification, system architecture synthesis, hardware and software synthesis, and system integration and configuration. Due to its broad scope, this paper does not discuss the steps of problem analysis and system requirement specification. We will assume that as result of this step the system requirements involving the functional (behavioural), structural and parametric requirements are created. We will also assume that:

- the functional specifications are expressed in a form (language) that enables their simulation, execution and/or formal analysis that can be supported with design automation tools,
- the structural requirements constrain the acceptable or preferred solution structures by limiting them to a certain class or classes, or imposing a preference relation on them (e.g. (partially) define the classes of generic architecture platforms (templates) that can be considered or are preferred for the RC system synthesis),
- the parametric requirements contain the necessary information on the preferences regarding the system solutions, expressed through constraints, objectives and tradeoffs’ information, enabling creation of meaningful decision models, and they are expressed in a way that makes possible an easy creation of such decision models.

We will thus directly start with the second step, i.e. system architecture synthesis.

5. Platform-based RC system architecture synthesis

RC system architecture synthesis consists in the creation of a system structure specification at the architecture level that supports the realization of the system’s behaviour as specified by its functional requirements, and fulfils the structural and parametric requirements to a satisfactory degree. This structural specification defines:

- a set of architectural structural resources (i.e. computation, memory and communication resources at a given architecture level of the design abstraction),
- an exact composition of the architectural resources to form the architecture platform, and
- a corresponding mapping of the required computation processes on the so constructed architecture platform and a (coarse) schedule of the computation processes.

The platform-based architecture synthesis involves

- an initial application (application class) analysis and generic architecture platform (template) design or selection for reuse, and
- the final platform-based architecture exploration and synthesis when using the newly designed or selected-for-reuse platform or platforms.

During the initial application (application class) analysis and generic architecture platform (template) design or selection, the application’s required behaviour, and the structural and parametric constraints are analyzed to discover the system’s main properties and bottlenecks. The information from this analysis is then used to propose one or more promising generic system architectures that could satisfy the application requirements, and to design or reuse the generic architecture platforms/templates.
corresponding to these architectures. The analysis of the application is performed using some metrics of the main system attributes involved in the requirement specifications, such as reaction time, latency or throughput, system complexity or circuit area, power or energy consumption, various costs, etc. Since the main system attributes are compound in most cases, they are expressed in terms of simpler directly measurable attributes that are directly affected by the design decisions (e.g. latency or throughput usually depend on clock frequency). Based on the impact analysis of the design decisions on the main system attributes, properties and bottlenecks, promising generic system architectures that seem to resolve the bottlenecks and satisfy the application requirements are selected for reuse, or new or modified generic architecture platforms are proposed and designed.

At the early system design stages proxy attributes and approximate attribute estimates of the relevant parameters are typically used. It is true that (more) accurate parameter estimates could also be obtained at the early design stages, but this would require a more detailed work-out of some tentative designs, which is usually expensive. Moreover, the estimates obtained this way would often be too detailed and possibly implementation-specific. Therefore, reasonable proxy attributes and approximations are preferred that can be computed relatively easy and fast, and can enable exploration of a large number of possible architecture alternatives in a short time. However, the methods used to obtain the approximations should be well understood by the system architect to adequately interpret the data.

5.1. Application analysis and restructuring

The section is sub-divided into three parts. First, we introduce the concept of application analysis. Then, we discuss the role of compilers in application analysis, parallelization and hardware compilation. Finally, we review several techniques for estimation of hardware and system characteristics.

5.1.1. Application analysis

Application analysis can be either static or dynamic. Static analysis computes the metric values when using approximate estimation methods. Usually, these methods exploit information from an (approximate) scheduling and allocation, as well as information on the relevant attributes of the architecture platform (template) and its parts (see e.g. [5,8,19,20]). Dynamic analysis finds the metric values by performing application profiling, when executing a given application on the target architecture platform or on the platform’s emulation or simulation model. Since the source-code level profiling may be not accurate enough, while the machine-code level would be too slow, the technique of micro-profileing was proposed to resolve this problem [21]. The results of static or dynamic analysis (or both) can serve several purposes. They can be used for instance to propose and reuse or design promising generic architecture platforms, to perform platform independent and dependent restructuring and optimization of the application algorithms or code, or to propose promising schedules and allocations.

For RC systems, the application algorithm and code restructuring and optimization are of primary importance. Since parallel RC platforms differ significantly from serial standard CPU-centric processor platforms, the optimal algorithms for the RC platforms usually also differ from those for the standard serial processors. Many application specifications and reference specifications of standards are formulated in a sequential C, C++ or other code. However, the (massively) parallel, systolic and pipelined computing concepts, and distributed data structures appropriate for RC platforms very much differ from the CPU-centric sequential computing concepts, and random-access and pointer-based data structures using central memory that are typical for the code for sequential processors [1,22–27]. In consequence, a direct translation into the RC hardware of the application specifications using computing concepts and data structures typical for sequential processing usually does not result in the expected acceleration or decrease of energy consumption. In parallel to the implementation of application-specific operations directly in the programmable hardware, mainly the usage of adequate parallel processing structures and the degree of parallelism exploited decide the application acceleration and decrease of energy consumption. Therefore, the application algorithms and code should be adequately restructured to effectively exploit the parallel resources of the RC platforms. Additionally, many modern RC platforms (e.g. modern FPGAs) are equipped in very efficient hardwired components for specific functions (e.g. integer multipliers or MACs) often used in signal processing and multimedia applications, and are able to implement arithmetic and other data-path circuits of an arbitrary word-length, while most RC platforms are unable to efficiently implement the floating-point operations. This results in different relative costs of various arithmetic and other operations on RC platforms than on the sequential processors.

The processes of application analysis and platform construction/selection are much less structured than the subsequent steps of the platform-based RC system design for the already existing platforms. They are also more dependent on the expertise of designers and their experience with similar applications and design problems, and on the availability of platforms that can be taken into account for the applications of a certain field. Therefore, the processes of platform selection and/or construction are usually performed using less automation than in the subsequent design steps. Nevertheless, the application analysis, platform selection or proposal, and application algorithm and code restructuring and optimization are also supported with some tools. These tools include the HLL compiler frontends, system description languages and their compilers, emulation and simulation environments, the emerging system-level architecture development environments, and other tools.

5.1.2. Role of compilers in application analysis, parallelization and hardware compilation

Very often the compiler frontends and their intermediate representations (IR) are used in collaboration with application analysis and restructuring tools for the application analysis and application algorithms or code restructuring and optimization. They enable application profiling and provide designers and application restructuring software with information on the applications, such as: average frequency or execution count of particular instructions, groups of instructions or functions, ratio of data-processing instructions to control instructions, data types and data access methods, etc. They also make possible application analysis through static performance and resources utilization analysis or scheduling techniques. The optimizing compilers that can be used for RC system design exploit the automatic parallelization techniques originally developed for the parallel processors.

Automatic parallelization consists of converting a sequential application code into a functionally equivalent parallel code that can be executed simultaneously using multiple hardware resources (e.g. processors, accelerators or functional units). In the case of multiple processors it is usually performed by the parallelizing source-to-source compilers. This represents a great challenge, because it requires a complex program analysis and
difficult decision process aiming at the selection and application of the best possible combination of many possible code restructuring operations. Additionally, part of the relevant for decision information (as e.g. variable values or ranges) is yet unknown at the compile-time. In general this process accounts for several aspects, such as loop restructuring, control restructuring, data reorganization or reuse techniques, etc. However, in many cases it mainly focuses on loop optimization, because very often loops account for the large majority of the computation effort and time.

Loop parallelization, also referred to as loop optimization, is performed as a combination of several different loop transformations. For instance, loop parallelization tries to split or unroll a loop so that its different iterations can be executed concurrently. However, before a loop can be actually parallelized, it has to be analyzed if the parallelization is safe and worth of effort. The first question is answered through data dependence analysis, i.e. determining whether the loop iterations can be executed independently of each other, and the second through estimation and comparison of the sequential and parallel execution times (also accounting for the communication overhead) and of the resources used in each case. Loop analysis is hard, mainly due to the difficulty of dependence analysis (e.g. pointers, recursion, indirect addressing, indirect calls, etc.), accesses to shared variables, I/O and other global resources, and the often unknown number of loop iterations.

Popular loop transformations include: loop splitting/strip mining (breaks a loop into multiple loops with the same bodies, but with different loop index sub-ranges, with its special case being loop peeling that decides to perform that first loop iteration before entering the loop for the remaining iterations); loop fission/tiling (splits a loop into several loops over the same index range, but each having as its body only a part of the original loop’s body); loop fusion (combines bodies of independent successive loops with the same number of iterations); loop-invariant code motion (places a loop-invariant code before the loop); loop permutation/interchange (exchanges inner loops with outer loops); loop unrolling (replicates the loop body several times to decrease the number iterations, i.e. the number of the loop condition tests and jumps – it is a kind of nested loop expansion enabling parallelism exploitation of the inner loops); loop unswitching (moves an if/else condition from inside of a loop to the outside through duplicating the loop’s body and putting a copy of the body in both if and else clauses); loop reversal (reverses the order of the index variable assignment, and this way enables some dependency elimination); loop skewing (rearranges multidimensional array accesses of a nested loop in which each iteration of the inner loop depends on previous iterations, so that the dependencies are only between iterations of the outer loop); software pipelining (a specific out-of-order execution of loop iterations used to hide the different latencies of various functional units that shifts operations across the iteration borders through subdividing the loop operations into several stages and executing in a single iteration stage 1 from iteration i, stage 2 from iteration i–1, etc.). More information on loop parallelization can be found in [28–30].

For many applications, parallelism can also be substantially increased through control restructuring, i.e. combining of control nodes, so that the operations corresponding to these nodes can be executed in parallel. Transformation of a serial nested if-then-else structure into a parallel switch/case multi-branch structure can serve here as an example. More generally, the control restructuring involves joining together frequently executed sequences of basic operation blocks. Since the average instruction-level parallelism per basic block is between 2 and 3.5 [31], basic block combining can further increase the instruction-level parallelism. Its examples include the trace scheduling [32], as well as superblock [33,34] and hyper-block formation [35,36]. Combining of the frequently executed basic block sequences makes it possible to eliminate the control constraints associated with the alternative execution traces. For instance, trace scheduling finds complete traces from the start to the end of a given control data flow graph (CDFG) and combines the basic blocks of each such trace.

Many compilers and application analysis and restructuring tools are based on the Stanford University Intermediate Format (SUIF) compiler infrastructure [37–39] which was especially developed to support parallelism identification. The SUIF frontend supports compilation of C, C++ or FORTRAN into SUIF high-level machine-independent intermediate representation (IR), which is a data-flow representation. From this high-level IR Machine-SUIF backend creates a medium-level IR and then a low-level machine representation [39]. The medium-level IR consists of lists of basic blocks of successive instructions that are executed without control transfer (e.g. jump, branch or loop instructions), and of control-flow graphs (CFG) that represent control relationships among the basic blocks. Each basic block contains at most one control transfer instruction represented by a CFG edge. Each basic block can be transformed to the form of a corresponding data flow graph (DFG), in which each instruction has its corresponding vertex and there is a directed edge between two vertices corresponding to two particular instructions if data produced by the first instruction is consumed by the second instruction. Through combining the information from the CFGs and DFGs, control data flow graphs (CDFG) of applications can be constructed, i.e. graphs that contain information on both the data processing and control transfer instructions and their interrelationships. CDFG representation is very often used in HW/SW co-synthesis and high-level (behavioural) hardware synthesis [40–42]. This way SUIF enables application analysis and profiling, application restructuring and optimization, as well as HW/SW co-synthesis and high-level hardware synthesis.

Many compilers and other synthesis tools use SUIF for application analysis (e.g. [43,44]), as well as for machine-independent application restructuring and optimization, HW/SW co-synthesis and high-level hardware synthesis (e.g. [31–36,45–74]). For instance, Napa-C [48] and Nimble [49] compilers perform HW/SW partitioning and map the hardware kernels to FPGA hardware. MARGE maps application blocks to collaborating hardware sub-systems implemented on FPGAs as functional units composed of a data path and a controller [56]. Streams-C compiler [50], based on Napa-C and MARGE, synthesizes hardware for Wildforce board [55]. CASH [57] transforms ANSI C to RTL Verilog constructing a distributed hardware architecture [58,59]. SPC [60] and XPP-VC [61] create parallel designs through performing loop transformations. Also the RAW compiler [62], which converts C or Fortran to Verilog, uses SUIF to perform parallelization [63], and additionally increases performance through partitioning data into multiple small memories and exploiting the virtual wire technique [64] that limits the wire length between the functional units. DEFACTO [65] uses SUIF to get the data flow representation from C, perform selected loop transformations and data permutations on this representation [67], and produce VHDL for the WildStar board [55]. It performs the design space exploration when using Monet behavioural synthesis tool [66] and changing the loop unrolling factor to obtain different area/delay tradeoffs [68,69]. ROCCC compiler [70] that generates RTL VHDL from C [71] uses SUIF2 and MachSUIF [38] for the control and data flow graph analysis. It performs loop restructuring, storage optimization and pipelining [72] when using its own IR [73]. MATCH compiler [74] uses SUIF and MATLAB M-File with compiler directives to transform architecture
expressed in MATLAB into RTL VHDL for FPGA implementation. The M-File is translated into SUIF IR and a MATLAB AST (abstract syntax tree) is created that is used to analyze dependences, perform loops transformations, and memory optimizations [75]. Using MATCH the AccelChip DSP (digital signal processor) environment has been developed for MATLAB to FPGA hardware synthesis [76], and was acquired by Xilinx who distributes its new version under name AccelDSP [77].

Several other non-SUIF-based compiler frontends are used for analyzing applications for RC hardware synthesis, including the well-known GNU GCC compiler [78]. For instance, ASC [79] translates C++ programs into FPGA hardware, when making possible an iterative design space exploration and tradeoffs' exploitation among the latency, throughput and area. Trimaran [80] and LANCE [81] environments use C frontends and SUIF IR representations for the application code analysis and parallelization, as well as the, the ASIP and customized compiler generation. Also the frontends of the Edison Design Group [82] for C++, Java and FORTRAN are used for application analysis. Hardware compilation from HLLs, and specifically from C sub-sets or dialects, has been intensively studied already since the early 1990s. For instance, PRISM [83] compiled from a C sub-set to an FPGA-like architecture. DeepC compiler [84] created synthesizable Verilog from C or FORTRAN. There are also many recent works in this field. SPARK [85] compiles C to synthesizable RTL VHDL. It creates data-path/controller architectures through performing some optimizations and scheduling of the application flow graphs, when exploiting the loop transformations and other code-restructuring techniques, as well as resource binding that minimizes interconnects. In the scope of the Cameron project [86] the SA-C (single assignment C) dialect of C has been developed [87] to support hardware compilation for image processing applications for general reconfigurable architectures, when using compiler directives to describe the hardware structure [88]. SA-C is translated to behavioural VHDL using DFGs as IR [89,90], by exploiting similar techniques to SUIF (e.g. loop transformations, pipelining, data analysis, etc.) to create optimized DFGs. The DFG nodes representing arithmetic or logic operators are then implemented as combinational circuits, loops as pipelines controlled by FSM controllers, and data transfers as communication circuitry. This results in a heterogeneous mixed-grain accelerator architecture.

The exploitation of HLL compilers for application analysis and hardware compilation gets increasingly more interest of the industry. Synopsis provides hardware compilation from SystemC [91] and Cadence from SpecC [92], Mentor’s Catapult C [93] performs a semi-automatic compilation from ANSI C++ sub-set into an RTL-level hardware netlist guided by the synthesis directives. It performs loop restructuring, variable and array mapping, and scheduling. Impulse CoDeveloper [94] includes the Impulse C hardware compiler based on the standard ANSI C, interactive parallel optimizer, and Platform Support Packages for a wide range of FPGA-based platforms, and for applications including image, video and digital signal processing, data compression/encryption, and high-performance computing. It supports application profiling and HW/SW partitioning, parallelization and pipelining of critical C code sections mapped to hardware, and automatic generation of the corresponding FPGA hardware. Celoxica DK4 Design Suite [95], sold to US firm Catalytic (later Agility) in 2006 and acquired by Mentor in January 2009, translates Handel-C, SystemC, and/or ANSI C to an RTL-level netlist. MATLAB provides a hardware compilation tool HDL Coder [96] that compiles DSP applications specified in Simulink models and MATLAB code to their corresponding VHDL code for FPGA implementation, when exploiting pre-designed IP hardware blocks. Synplicity (currently a Synopsys company) offers the Synplicity DSP tool based on MATLAB and Simulink design tools that provides an ESL synthesis technology for FPGA/ASIC hardware compilation of DSP applications. The Synplicity DSP synthesis engine allows designers to explore the area/speed tradeoffs and automatically create RTL-level hardware implementations [97].

5.1.3. Estimation of hardware and system characteristics

Application analysis and restructuring tools, as well as the actual architecture synthesis tools compute and use various estimations of the main system or hardware attributes related to reaction time, latency or throughput, system complexity or circuit area, power or energy consumption, various costs, etc. Many different estimation methods and tools are described in the literature, and many other are included in the industrial software and not published.

One possible estimation approach is to perform an actual hardware synthesis and this way to get estimations of the custom hardware attributes (e.g. [98]). This is, however, time consuming for larger designs. Another approach is to pre-compute the estimations for the basic hardware modules (e.g. hardware implementing the basic operations involved in an instruction) through performing an actual hardware synthesis tools, and to calculate the estimations for more complex modules (e.g. custom instructions) through summing up the pre-computed values of the basic modules [99–102]. However, this kind of estimation is implementation technology specific, and requires to account for the technology mapping, placement and routing, and some other steps and optimizations in the synthesis chain that will actually be used for the RC hardware synthesis. Yet another promising alternative approach is to first compute some approximate high-level RC hardware attribute estimations directly from a HLL or intermediate representation (e.g. C, MATLAB, CDFG, HCDG, etc.), and in the course of the design to gradually improve the estimation precision. In particular, using DFGs created by the Cameron compiler for applications in SA-C, Kulkarni et al. [102,103] proposed an estimation technique that predicts the FPGA data-path area expressed in LUT-count with accuracy of 5%, where the estimate is calculated based on the resource characterization of particular DFG nodes. For Design Trotter [104,105], a tool that performs the design exploration through parallelization and scheduling of hierarchical CDFGs obtained from a restricted C specification, Bilavarr et al. [106] proposed a technique that computes area, delay and power estimations through constructing several RTL-level designs, and physical implementation estimation of the constructed designs using FPGA pre-characterization data obtained from an FPGA data-sheet. This resulted in 10% accuracy for delay and 18% for area. Brandoolese et al. [107] developed an area estimation technique at System-C level for FPGA-based designs. Their technique estimates basic characteristics for each hardware resource of each kind (Finite State Machine or FSM, operation, multiplexer, register, and glue logic), and using a weight corresponding to each resource kind computes the number of flip-flops (FFs) and look-up tables (LUTs). An average accuracy of this technique is 25%, but the glue logic estimation is of a very low accuracy of 59%. Another high-level FPGA are estimation technique is proposed by Enzler et al. [108]. It uses DFGs of an application to characterize all the operations in relation to a given FPGA to compute the approximate metrics of area, delay, latency, and throughput. It does this, however, for only some specific operators, and does not account e.g. for registers, routing and control. Nayak et al. [109] proposed a technique that estimates the FPGA data-path area based on the RTL code generated from MATLAB through computing a metrics dependent on some operation and register properties. For each operation its corresponding control logic block (CLB) count is obtained from the
area pre-characterization parameterized in relation to the operation type and size. The estimation error of this technique is within 16%. Bjureus et al. [110] computes area and delay estimates from a scheduled DFG that is obtained from an execution trace of a MATLAB application simulation, annotated with resource information of each operation by its FPGA implementation. The accuracy of the estimation is within 10%, but the estimation process is time consuming. To support hardware/software partitioning for a system composed of a Very Large Word (VLW) processor and a coarse-grain reconfigurable array within the Lycos environment [111], Yan et al. [112] developed estimates for hardware area and delay, as well as for software execution time, through counting all types of functional units, registers, multiplexers, and control logic, and summing up an estimated area and delay for each type. The estimation accuracy is within 13% for area and 8% for delay. In [113] Task Transformation, Cost Estimation and Decision Mapping tools are described. The Task Transformation tool performs various transformations on a parallelized C code such as loop transformations, behaviour scheduling etc. to produce several versions of each individual task appropriate for a specific processing engine (e.g. general purpose processor or GPP, DSP or FPGA). The Cost Estimation tool determines the task execution time when running on a specific processing engine through using a model involving a set of parameters extracted from the C code, such as the number of instructions of each type, instruction graph depth, degree of instruction parallelism, and number of memory accesses. The Decision Mapping tool uses this information to decide the assignment of tasks to processing engines. In [114] some macro models for high-level power and area estimation in FPGAs are presented, and in [115] a detailed power model for FPGAs. Pre-synthesis area estimation of reconfigurable streaming accelerators is also considered in [116].

5.2. Macro-architecture exploration and synthesis for RC systems

The section is sub-divided into two parts. First, we introduce the main concepts of macro-architecture exploration and synthesis, and subsequently, we discuss the scheduling and mapping of computation processes in more depth.

5.2.1. Macro-architecture exploration and synthesis concepts

Architecture exploration and synthesis results in the creation of a structure that defines a composition of the architectural hardware resources that support the application behaviour required and satisfies the parametric constraints and objectives, as well as the mapping of the application behaviour on the so-defined hardware architecture.

During architecture exploration and synthesis a set of possible architectures defined by the selected architecture platforms is analyzed in relation to the required application’s behaviour and constraints, and the most suitable architecture is decided and constructed/instantiated. To decide the most suitable architecture, the possible architectures are analyzed in relation to various metrics of interest (such as speed or throughput, power or area) and basic controllable system attributes affecting them (e.g. number of system modules of each type, clock frequency of each module, communication between the modules, schedule and binding of the required behaviour, etc.), and the results of this analysis are compared to the design constraints and optimization objectives. The analysis techniques are similar to those discussed above, but when going down in the design abstraction they need to be applied with a higher precision.

The architecture exploration and synthesis methods may have different character depending on the set and nature of applications and architectures considered, as well as the analysis and synthesis techniques used. In particular, depending on the nature of the generic architecture platform/template and final RC system, the architecture exploration and synthesis may involve macro-architecture or micro-architecture exploration and synthesis, or both.

Macro-architecture exploration and synthesis is related to the platforms with architectural resources at the processor or sub-system level (i.e. involve programmable or non-programmable processors, memories and communication resources). It aims to decide the kind of platform that is the most appropriate (in the cases when several platforms are promising), and for a given platform, its specific instance, as well as the mapping of the application required behaviour on the platform instance. In particular, it decides the platform type to be used, the number of programmable and non-programmable processors, memories and communication resources of each type possible to be instantiated on the platform, the particular instantiations of the processors, memories and communication resources, as well as the mapping of the required computation processes on the so-instantiated platform and a coarse schedule of the computation processes. Macro-architecture exploration and synthesis is often referred to as HW/SW system partitioning or system-level architecture synthesis. In this paper, we are considering automated architecture exploration and synthesis process, and discussing what kind methods and tools exist or should exist to perform this process effectively and efficiently. The core activities of the system-level architecture design involve the following:

- design or reuse of a generic architecture platform, and its modelling in the form of an abstract architecture template (once for an application/system class)
- generic architecture template instantiation (for each particular application/system),
- abstract requirement modelling (for each particular application/system),
- process scheduling and mapping on the generic architecture template instance (for each particular application/system and template instance),
- architecture refinement and optimization (processing, interfacing, and memories abstraction refinement and optimization – for the selected architecture).

To perform the system architecture exploration and synthesis effectively and efficiently, in advance a generic architecture platform corresponding to a given application class and its main resources (processors, memories and communication resources) have to be developed or decided to be reused, based on the analysis of the application class and using the prior knowledge and experience related to applications/systems from this class or analogous classes. Also, a generic architecture template, being an abstract system-level model of the architecture platform and of its modules, adequate for the architecture exploration and synthesis issue has to be developed or reused. For the same reason, the original system requirements that may be very complex have to be analyzed and a much simpler abstract system-level model of the behavioural and parametric requirements being adequate for the architecture design issue has to be constructed. The actual platform-based system-level architecture exploration starts with such constructed abstract model of the architecture design issue composed of:

- an abstract system behaviour model representing a network of collaborating computation processes that have to be realized;
- an abstract system (hardware) platform model being an instance of the generic architecture template; and
• an abstract decision model composed of a set of constraints, objectives and trade-off preferences related to all system characteristics relevant to the architecture design issue.

The decision model defines how the mapping and scheduling of the computational components (processes or tasks) onto the hardware resources are constrained and interrelated, and represents the designer’s preferences and aspirations. Its constraints and preferences have to be fulfilled to a satisfactory degree by each acceptable implementation of the required computational processes.

The network of the computational processes represented by the abstract system behaviour model has to be appropriately distributed over the structure of modules of an adequate instance of the generic architecture template and scheduled, to define the actual system architecture – i.e. the selection and interrelationships of the platform modules, assignment of the computational processes to the platform modules and their schedule – that satisfies the specific (structural, physical, etc.) constraints and optimizes the objectives of the quality model in the context of specific trade-off preferences between the objectives.

Since the abstract requirement modelling, generic architecture instantiation, and process scheduling and mapping have to be performed for each particular application/system anew, and an adequate design space exploration of complex systems requires re-iterations and refinements, these three processes should be automated to a high degree.

Providing the initial requirements are in any formal language, their abstract model can be automatically constructed through a sort of parsing, analysis and abstract translation. The main problem here is not the ability or sort of automation, but adequate decisions on what and how to abstract, and what kind of abstract modelling to use, in order to ensure that the resulting abstract models will contain all information that is necessary for the highly effective decision-making in the scope of the architecture synthesis issue, but at the same time will be as compact and easy to process as possible to ensure an efficient behavioural analysis.

Comparing to the architecture template instantiation, the process scheduling and mapping seems to be a decision task of higher complexity, because it involves a complicated network of numerous processes that have to be appropriately scheduled and assigned on a proposed heterogeneous multi-processor architecture template instance. Moreover, usually a very limited number of the template instances being the most promising candidates for the actual system implementation can be reasonably safely selected based on the even coarse analysis of the requirement specification. If one such instance fails, another one can be tried, of course provided that the process scheduling and mapping is efficient. Finally, after performing the process scheduling and mapping for certain architecture template instance and analysis of this process and its result, much information is gained on what (another) platform instance seems to be the most appropriate for implementation of the processes. Therefore, the automatic support for the decision-making on platform instantiation can remain limited, and the automatic support for the process scheduling and mapping tasks is of primary importance.

To start the actual architecture exploration and synthesis process, based on the initial requirement analysis the designer makes a proposal of a generic architecture platform instance and its resource allocation that are expected to be adequate to realize the required system behaviour and satisfy the design’s parametric requirements. His decision is implemented through an instantiation of the corresponding abstract IP models of the generic architecture template and of its modules that is supported with adequate design automation means. Using the abstract requirement model and the IP models of the architecture template instance and of its modules, the scheduling and mapping are performed of the network of collaborating computation processes defined by the behavioural requirements on the proposed architecture template instance, when observing the parametric constraints, objectives and trade-off preferences. Its result is a decomposition of the required network of the system’s computational processes into a network of sub-systems. This network of sub-systems represents the system architecture, i.e. the generic architecture template instance with the computational processes scheduled and mapped on this instance. The result of scheduling and mapping has not only to realize the required system’s behaviour, but as well to satisfy specific constraints and optimize certain objectives in the context of specific trade-off preferences between the objectives. The system architecture constructed in this way is subsequently examined and analyzed to check to which degree this all is satisfied. In this semi-automatic architecture synthesis process, both the available system resources, and the objectives, constraints and trade-off preferences are imposed by the designer. On the other hand, the mapping and scheduling decisions determine the actual system resource requests. To be acceptable, the resource requests must match in a satisfactory way the pool of the available resources, in the light of the objectives, constraints and trade-off preferences.

In the design process of (reconfigurable) embedded systems many common aspects can be identified and this makes it possible to develop reasonably general synthesis approaches applicable to a large range of applications, as the approach briefly sketched above. Also, most of the complex real-time embedded system applications can be naturally viewed to involve multiple information collection, computation, transmission and storage processes that can operate concurrently and communicate with each other and the environment. On the other hand, however, the character of a particular embedded system and its design process may considerably depend on the type of the application the system serves. In consequence, partly different specific system modelling and representation concepts, and synthesis methods may be appropriate for different application fields. In particular, a different actual system behaviour specification language, as well as, its underlying abstract semantic model may be appropriate and this may substantially influence the architecture synthesis process.

The system-level architecture synthesis problem is NP-complete and its solution space is exponential in the number of operations considered. It is a combination of interdependent NP-complete problems, because both the operation mapping (allocation, binding, and assignment) problem and scheduling problem are known to be NP-complete. Even after restriction to the Precedence Constrained Scheduling problem through assuming an equal execution time for the hardware and software task execution it remains NP-complete [117,118]. Moreover, to perform a well-informed architecture synthesis based on real facts, typical and critical system operation scenarios (traces of system behaviour) have to be analyzed to reveal the causal relationships between the system events, the precedence and conflict relations between the transformations, and the resource requests, and the information obtained have to be used together with the processor models and hardware estimators to estimate the resource usage and guide the operation scheduling and mapping. The number of the behaviour traces also grows exponentially with the number of operations considered. Therefore, the number of operations considered, and in consequence, the granularity at which operations are considered largely impacts the architecture synthesis method and tool performance, as the elementary operations can be joined together to form some coarser macro-operations.
(e.g. tasks) to decrease the number of operations considered. However, the coarser operation granularity may decrease the quality of the design parameter estimation, and result in architectures of decreased quality. Thus, there are tradeoffs between the architecture synthesis effectiveness and efficiency depending on the operation granularity considered. Summing up, to perform the architecture synthesis effectively and efficiently, the original system behavioural requirements have to be analyzed and an abstract coarser system-level model of the system behaviour being adequate for the architecture design issue has to be constructed. Providing the initial requirements are in any formal language, their abstract model can be automatically constructed through a sort of parsing, analysis and abstract translation. The main problem here is not the ability or sort of automation, but adequate decisions on what and how to abstract, and what kind of abstract modelling to use, in order to ensure that the resulting abstract models will contain all information that is necessary for the highly effective decision-making in the scope of the architecture synthesis issue, but at the same time will be as compact and easy to process as possible to ensure an efficient behavioural analysis, as well as operation scheduling and mapping.

For different application domains, different system behaviour specification languages are convenient or typically used. For instance, to specify the signal processing and some other algorithms, the textual algorithm specification and modelling language MATLAB [96], graphical Simulink [96], or Ptolemy [119] tool are often used. The signal processing, video processing, and other algorithms are also often specified in high-level languages (HLLs), such as C, Java or FORTRAN, as they are typically targeted to systems involving software-programmable processors and their developers are often software designers or programmers. For streaming computations a specific stream-oriented computation model has been proposed that is able to well express the parallelism of spatial computations [120]. The behaviour of complex real-time systems that represent mixture of data processing and control can be well modelled using the structured analysis with real-time extension (SA/RT) requirement modelling method [5,8,19,121,122]. The system behaviour specification languages can be more or less related (or unrelated) to the underlying computation platform. The closer the language relation to the computation platform, the easier can the application algorithm developer or programmer optimize the application algorithms or programs for the platform. The further is the relation the more important is the role of software and/or hardware compilers in closing of the semantic gap between the platform independent application behaviour specification and the platform, i.e. in customization of the application algorithms or programs to the platform, and in customization of the platform to the application.

Various underlying abstract semantic models referred to as models of computation may be used with different algorithm programming or specification languages. The models of computation most often used for the architecture synthesis include the already discussed (hierarchical) DFG, CDFG and similar to them Hierarchical Conditional Dependency Graphs (HCDG) [123,124], as well as, Communicating Sequential Processes (CSP) [125], Petri nets [126–128], Khan process networks [129,130], and Instruction Set Architecture (ISA) models [131,132]. Different models have different features and differently serve various kinds of applications, computations, computation platforms, and levels of system abstraction. Additionally, for the increasingly complex modern applications hierarchical heterogeneous models of computation are used more and more often, because they make possible usage of different simple models in combination to model parts of applications having different character.

5.2.2. Scheduling and mapping of computation processes on template instance

Many design environments for reconfigurable systems leave the actual solution of the platform-based macro-architecture design problem to the designer or only support the designer in an interactive design space exploration (see e.g. [48,133–145]). In recent years, however, a lot of research and development has been performed in the area of the platform-based macro-architecture synthesis automation, both in academia and industry. Several semi-automatic HW/SW compilation systems have been recently proposed for compilation from (sub-sets of) HLLs, such as C, C++ or Java [56,146–151]. Some of them allow for specification of the task-level parallelism using threads [150,151], and some other are targeted to instruction-level parallelism [152]. Due to the differences between hardware and software (e.g. the basically sequential nature of software and parallel spatial nature of hardware), the direct hardware compilation from the HLLs is, however, difficult and can easily result in a low-quality hardware [153–155]. Since both the scheduling problem and operation mapping problem are known to be NP-complete, the system-level architecture synthesis problem is NP-complete [117,118]. Consequently, the existing strictly optimal algorithms either solve some constrained versions of the problem (e.g. for a system composed of a single programmable processor and single accelerator (e.g. [156,157]) or are unable to efficiently solve larger problem instances (e.g. [158]). Therefore, heuristic approaches are used to effectively and efficiently solve practical instances of the architecture synthesis problem for larger heterogeneous systems [159–166].

In the past, many different methods and algorithms have been proposed to solve different formulations, special cases or instances of the system architecture synthesis problem. One of the first works that proposed a heuristic method for multi-way heterogeneous system partitioning was an early work of Peng and Kuchcinski [162]. From the Petri-net-based behaviour specification they constructed an abstract model in the form of a weighted graph representing computational processes and their communication. Using a simulated annealing algorithm they partitioned the graph into several sub-graphs corresponding to different system components, so that the sum of the weights of all the cut edges was minimized and the totals of the weights of the sub-graphs were balanced. Unfortunately, their method was too abstract, as it only accounted for system complexity in abstract terms, and did not directly account for actual system performance, implementation costs or other important characteristics. In [167] a method has been proposed for mapping of a computation specification in ANSI C onto several homogeneous processors on a single chip. In [168] the problem of HW/SW system partitioning for heterogeneous multi-coprocessor HW/SW systems is formulated as an integer programming problem. First, the actual HW/SW partitioning is performed when estimating the timing of each functional node and observing the timing constraints, and subsequently, the actual scheduling is performed for each partition block, when guaranteeing the system execution time constraints. In [158,168,169] (mixed) integer linear programming is used to solve various versions of the problem. In [170–172] iterative improvement algorithms are proposed. In [163–165] constructive heuristic algorithms are used to solve various versions of the problem. Kuchcinski proposed to use the constraint programming approach [160] and Teich et al. [166] an evolutionary algorithm. The evolutionary/genetic algorithms and constraint programming belong to the most promising approaches to efficiently produce high-quality solutions for the complex architecture synthesis problems involving multi-objective optimization. Both approaches were successfully used later in several other HW/SW co-design methods [8,9,19,169,173–175].
More recently, HW/SW partitioning on programmable platforms is proposed that uses the ant colony optimization method [176,177]. In [178] a brief overview of the early HW/SW co-design approaches has been made. In [10] a taxonomy and survey of platform-based electronic-system-level (ESL) design methodologies have been presented. The authors classified more than 90 different academic and industrial ESL design methods and distinguished their several classes in relation to the following three aspects: functionality (functional design representation), platform (kind of platform used) and mapping (assignment of functionality to a set of correctly interconnected modules). Most of the analyzed methods focus at only one or two of the above aspects, but some of them address all three aspects (e.g. Ptolemy [11], Metropolis [12], SpecC [13,92], or Platform Architect [14]). Although being incomplete and not accounting for several more recently proposed important ESL methods, this taxonomy paper was one of valuable attempts to order the ESL design scene, as the early ESL methods and tools seemed to be proposed ad hoc, without first developing an appropriate general design methodology.

A design methodology that seems to be adequate to solve the complex ESL design problems is the methodology of quality-driven design proposed in [3,4] and briefly discussed in Section 3. Based on this methodology, a quality-driven model-based system-level design exploration and architecture synthesis approach that addresses the problem of an effective and efficient multi-objective optimal architecture synthesis for complex real-time embedded systems was developed [5,8,19]. Their work opened a new generation of well-funded systematic ESL design methods.

The quality-driven model-based system-level design exploration and architecture synthesis approach [5,8,19] targets heterogeneous multi-processor architectures involving a global memory, several programmable processors and (reconfigurable) hardware accelerators, each equipped with a local memory, and an efficient communication structure among the processors, accelerators and memories. It exploits an adequate mixture of the design reuse with the automatic synthesis that is necessary to efficiently develop the contemporary and future complex systems.

To perform the system architecture exploration and synthesis effectively and efficiently, the original system requirements have to be analyzed and an abstract system-level model of the behavioural and parametric requirements being adequate for the architecture design issue has to be constructed.

The actual system architecture exploration starts with such an abstract model of the architecture design issue composed of:

- an abstract system behavioural model representing a network of collaborating computation processes that have to be realized;
- an abstract system (hardware) platform model being an instance of the generic architecture template; and
- an abstract decision model composed of a set of constraints, objectives and trade-off preferences related to all system characteristics important for the system architecture synthesis issue.

The decision model defines how the mapping and scheduling of the computational components (tasks) onto the hardware resources are constrained and interrelated, and represents the designer's preferences and aspirations. Its constraints and preferences have to be fulfilled to a satisfactory degree by each acceptable implementation of the required computational processes.

The network of the computational processes represented by the abstract system behavioural model has to be appropriately distributed over the structure of modules of an adequate instance of the generic architecture template and scheduled, to define the actual system architecture – i.e. the selection and interrelationships of the platform modules, assignment of the computational processes to the platform modules and their schedule – that satisfies the specific (structural, physical, etc.) constraints and optimizes the objectives of the quality model in the context of specific trade-off preferences between the objectives.

Since the abstract requirement modelling, generic architecture instantiation, and process scheduling and mapping have to be performed for each particular application/system anew, and an adequate design space exploration of complex systems requires re-iterations and refinements, these three processes should be automated to a high degree.

Providing the initial requirements are in any formal language, their abstract model can be automatically constructed through a sort of parsing, analysis and abstract translation. The main problem here is not the ability or sort of automation, but adequate decisions on what and how to abstract, and what kind of abstract modelling to use, in order to ensure that the resulting abstract models will contain all information that is necessary for the highly effective decision-making in the scope of the architecture synthesis issue, but at the same time will be as compact and easy to process as possible to ensure an efficient behavioural analysis.

Comparing to the architecture template instantiation, the process scheduling and mapping seems to be a decision task of higher complexity, because it involves a complicated network of numerous processes that have to be appropriately scheduled and assigned on the proposed multi-processor architecture template instance. Moreover, usually a very limited number of the template instances being the most promising candidates for the actual system implementation can be quite safely selected based on the even coarse analysis of the requirement specification. If one such instance fails, another one can be tried, of course provided that the process scheduling and mapping is efficient. Finally, after performing the process scheduling and mapping for certain architecture template instance and analysis of this process and its result, much information is gained on what (another) platform instance seems to be the most appropriate for implementation of the processes. Therefore, the automatic support for the process scheduling and mapping task is of primary importance. In consequence, the system architecture design process organization and its automatic support as represented in Fig. 1 and described below has been proposed.

To start the actual architecture exploration and synthesis process, based on the initial requirement analysis the designer makes a proposal of a generic architecture platform instance and its resource allocation that are expected to be adequate to realize the required system behaviour and satisfy the design's parametric requirements. His decision is implemented through an instantiation of the corresponding abstract IP models of the generic architecture template and of its modules that is supported with adequate design automation means. Using the abstract requirement model and the IP models of the architecture template instance and of its modules, the scheduling and mapping are performed of the network of collaborating computation processes defined by the behavioural requirements on the proposed architecture template instance, when observing the parametric constraints, objectives and trade-off preferences. Its result is a decomposition of the required network of the system's computational processes into a network of sub-systems. This network of sub-systems represents the system architecture, i.e. the generic architecture template instance with the computational processes scheduled and mapped on this instance. The result of scheduling and mapping has not only to realize the required system's behaviour, but as well to satisfy specific constraints and optimize certain objectives in the context of some specific trade-off...
preferences among the objectives. The system architecture constructed this way is subsequently examined and analyzed to check to what degree this all is satisfied. In this semi-automatic architecture synthesis process, both the available system resources, and the objectives, constraints and trade-off preferences are imposed by the designer. On the other hand, the mapping and scheduling decisions determine the actual system resource requests. To be acceptable, the resource requests must match in a satisfactory way the pool of the available resources, in the light of the objectives, constraints and trade-off preferences (see Fig. 2). In this architecture design process, the scheduling, mapping and architecture analysis processes are fully automated through development of methods and corresponding prototype EDA-tools for:

- **automatic system architecture construction and selection** through scheduling and mapping of the processes represented with the abstract behaviour model on the architecture template instance, when using a corresponding multi-objective decision model and parametric estimates,
- **abstract behaviour modelling and behavioural analysis**, based on the abstract formal model of net systems and representative system work scenarios, and
- **system architecture parameters’ estimation**, by using information from the abstract behaviour model, mapping configuration, schedule, and characteristics of the hardware resources.

To search for satisfactory solutions, and to determine whether the constraints can be met with the available resources and to what degrees are the objectives satisfied, a search method equipped with multi-criteria decision-making aids is used that represents a mixture of a genetic algorithm with constraint programming, and is similar to the search methods successfully used in [9,19,160,166,169,173–175,179].

From these EDA-tools, the designer receives feedback composed of a model of the constructed architecture and important characteristics of the architecture showing to what degrees the design objectives and constraints are satisfied by this architecture. Using this feedback, the designer can decide:

- **to successfully finish the architecture synthesis phase** (if he/she is satisfied with its result) or **to stop it** (if it turns out to be impossible to realize the required system given the available resources);
- **to make a new proposal of the generic architecture template instance and its resource allocation that are expected by him to be more adequate** (if he is not satisfied with the degree to which the design objectives and constraints are satisfied); or
- **to modify the generic architecture platform/template, some of its modules or design requirements** (if it turns out that it is impossible to realize the required system with the currently used generic architecture).

In the last two cases after making a new platform instance and/or requirement proposal, the next iteration of the scheduling, mapping and architecture analysis is started.

A crucial issue in the system macro-architecture synthesis is an adequate abstract behaviour modelling, and the method considered focused on this issue. Many embedded systems require a behaviour modelling capability that can well account for the control and data-processing mixture. One of the methods that can adequately model the control and data-processing mixture is the SA/RT requirement modelling method [5,19,121,122]. Some important advantages of the SA/RT method follow from the fact that the SA/RT models are capable of capturing a complex system behaviour involving hierarchies of multiple information processes that can operate concurrently and communicate with each other and the environment, but at the same time, the models are to a large extend graphical, intuitively clear and simple enough to be well understood and easily discussed by the system designers, their customers and supervisors. Therefore, the SA/RT modelling is widely used in industry, and was adopted in the considered method.
For complex designs, the system-level behavioural analysis and estimation of the architecture parameters directly from the original SA/RT behavioural specification are very difficult and time consuming. This results from the fact that the original model contains a lot of information that is not relevant for the architecture synthesis issue. Therefore, a much more abstract and simpler transition system model is extracted for the system architecture design issue from the original behaviour model. To efficiently deal with design complexity, an abstraction is used that consists of:

- data hiding,
- process abstraction and encapsulation (the internal transformation behaviour is modelled as an atomic action),
- grouping of zero-time internal events into compound transitions, and
- considering the compound transitions with interleaved events to be equivalent.

Using this abstraction, a state machine representation is constructed for each SA/RT modelling component that only contains the states and transitions of substantial importance for the architecture synthesis issue. In particular, the state machine graph only contains the system actions that do not take place instantaneously. The behaviour of each discrete data transformation is represented by a simple state machine with three states: off, waiting, and active. The coupled state machines of several data transformation tasks form an event structure. The causal relationships between the system events for a representative use case are obtained by unfolding the event structure of the coupled state machines. In this way, based on the original system behaviour model, an abstract transition system model is constructed that is expressed in our case as a formal elementary net system model [19,180]. For this transition system model, the state reachability analysis is performed to determine the typical and/or critical system work scenarios, i.e. the traces implementing particular system functions for the typical or critical system use cases, and based on those traces, the precedence and conflict relations between the transformations (see Fig. 2). This information is used for the process scheduling and mapping, and together with the processor models and hardware estimators, to estimate the resource usage (e.g. hardware resource usage, execution time, etc.). The resource usage estimates are then used as values of some basic parameters in the formulas expressing the constraints and objectives in the decision model of the system architecture synthesis issue, and this way guide the architecture synthesis process.

5.3. Specific issues of run-time reconfigurable systems

Run-time reconfiguration has multiple advantages [181,182]. However, HW/SW system architecture synthesis methods aimed originally at the development of static system architectures for non-reconfigurable HW/SW systems. For the statically (compile-time) reconfigurable systems, the macro-architecture synthesis problem is basically the same as for the non-reconfigurable HW/SW systems. Although for different applications or their different executions, different particular architectures can be instantiated in the RC system through reconfiguration, each particular architecture can be developed using the general
methods of the platform-based system macro-architecture synthesis, providing that they account for the characteristics of a given reconfigurable platform. The situation is different for the dynamic (run-time) reconfigurable systems, for which the nature of the architecture synthesis problem is different regarding some aspects. Run-time reconfiguration introduces the extra freedom of reconfiguration during the application execution, and this way introduces two additional inherent basic problems: how to adequately perform the temporal partitioning of the application behaviour in combination with its spatial partitioning, and how to minimize the reconfiguration overhead. The temporal partitioning consists of decomposition of a computation process or set of computation processes into a set of in time mutually exclusive configurations that are scheduled and dynamically re-configured in the reconfigurable hardware [183–192]. The reconfiguration overhead involves both the extra reconfiguration time and energy consumption, and may also involve some extra hardware or software resources devoted to run-time reconfiguration. Reconfiguration overhead can be reduced in several ways that include configuration pre-fetching, caching, compression, relocation, de-fragmentation and partial reconfiguration [193–196]. To reduce the time and energy overhead associated with the run-time reconfiguration, several architectures have been proposed for dynamically reconfigurable FPGAs that enable partial reconfiguration in several cycles without influencing the execution of the remaining FPGA parts, e.g. [197–204]. Nevertheless, the reconfiguration overhead remains substantial and requires careful consideration during the system partitioning. For a run-time reconfigurable system, both spatial and temporal partitioning of computational processes have to be performed, so that at any time both the software and reconfigurable hardware partitions satisfy the resource constraints, and the computation execution time (which now may also involve the reconfiguration time) satisfies the performance constraints. The dynamic reconfiguration process requires introduction of a software- or hardware-implemented reconfiguration controller [205–219]. In some RC systems configurations are not only dynamically re-placed, but also dynamically re-connected to some other RC system modules [208,209,220,221]. Also, some tools for generation of partial configurations and bitstreams have been proposed [51,222,223], and a development environment and run-time support system for tightly coupled pipelines in dynamically reconfigurable platform FPGAs [221]. Moreover, with the introduction of the dynamic reconfiguration the constraints and objectives of the system macro-architecture synthesis partially change. The computation execution time for its particular schedule and its given temporal and spatial partitioning now includes the related reconfiguration time [210], and should be minimized or satisfy given constraints, provided at any time each reconfigurable hardware partition satisfies the reconfigurable hardware resource constraints. The total reconfigurable hardware area is not required anymore to be minimized or fit into the available hardware resources, but each hardware partition has to fit in its corresponding reconfigurable hardware resource. Additionally, the scheduling, partitioning and partition allocation are tightly coupled, in the sense that a change in one of them substantially influences each other, and consequently, to arrive at high-quality architectures these processes to be performed together. Summing up, the extra freedom of the run-time reconfiguration and additional dimension of time substantially complicates the architecture synthesis problem. The evolutionary/genetic algorithms and constraint programming belong to the most promising approaches to efficiently solve such complicated discrete optimization problems. Moreover, the scheduling in a HW/SW system can be static (the schedule is determined off-line prior to execution) or dynamic (the schedule is determined on-line), and consequently, in a run-time reconfigurable system the subsequent configurations may be scheduled dynamically [211]. In [212] a dynamic real-time software scheduler for HW/SW systems is presented, and in [213] a survey of several scheduling approaches for control-dominated and data-flow-dominated systems [224,225]. Many different algorithms for scheduling and/or temporal partitioning for the run-time reconfigurable systems were presented [134,183–192,214–218,226–229]. Also some mapping methods have been proposed in the context of dynamic reconfiguration [46,214,230–233]. In the past, several specific system architecture synthesis methods and algorithms have been proposed for the dynamically reconfigurable systems [147,174,234–239], but they all make some serious simplifying assumptions (e.g. assume a system composed of a single programmable processor and single FPGA [147,236,239] or do not allow concurrent tasks on the same FPGA [174]) or are not efficient for larger problem instances due to using MILP formulation [235]. Recently some more advanced system architecture synthesis methods have been proposed for the dynamically reconfigurable systems [9,134,192]. The method presented in [9] deserves a special attention, as it accounts for the partial run-time reconfiguration and performs the multi-objective Pareto out-ranking-based optimization [240], when using an efficient parallel re-combinative simulated annealing algorithm, being a composition of a genetic and simulated annealing algorithm [241]. Several new design methods and frameworks have been recently proposed for the dynamic reconfigurable systems, including those presented in [242–247]. Also, the very important issue of power consumption has received some attention [114,115,248–253].

5.4. Micro-architecture exploration and synthesis for RC systems

The section is sub-divided in two parts. First, we introduce the main concepts of micro-architecture exploration and synthesis, and subsequently, we focus on more specific issues of hardware accelerator design.

5.4.1. Micro-architecture exploration and synthesis concepts

Micro-architecture exploration and synthesis is related to the platforms in which the architectural resources are at the register transfer level (RTL) or instruction level. It results in the creation of a structure that defines a composition of the RTL-level hardware resources that supports the application behaviour required and satisfies the parametric constraints and objectives, as well as the mapping of the application behaviour on the so-defined RTL-level hardware architecture. The micro-architecture is a specific realization of the Glushkov model representing the data-path/control-path dualism. Consequently, the RTL-level hardware resources include data-path components, that can be sub-divided into computation components (e.g. arithmetic and logic units or ALUs, multipliers, dividers, application-specific computation units etc.), memory components (e.g. registers, embedded memories etc.) and interconnection components (e.g. point-to-point interconnections, buses, multiplexers, etc.), and control-path components (e.g. combinational logic components, registers, embedded memories, interconnections, etc.) that serve for the control-path implementation and its interconnection with the data-path. The data-path performs the major computation, transmission and storage operations on data. The control-path decides and issues the control signals to adequately select and perform the data-path operations and route the data, depending on the current situation that involves the controllers’ own state and its input composed of the data-path status signals and some external signals. During the micro-architecture exploration and synthesis a set of possible architectures defined by the selected architecture platform(s) is
analyzed in relation to the required application’s behaviour and constraints, and the most suitable architecture is decided and constructed/instantiated. To decide the most suitable architecture, the possible architectures are analyzed in relation to various metrics of interest (such as speed or throughput, power or area) and basic controllable system attributes affecting them (e.g. number of system modules of each type, clock frequency of each module, communication between the modules, schedule and binding of the required behaviour, etc.), and the results of this analysis are compared to the design constraints and optimization objectives. Dependent on the concrete design situation, the micro-architecture synthesis can be the only or one of more steps in the architecture synthesis of a (re-)configurable system. The analysis and synthesis techniques are here similar to those discussed in the previous sections, but when going down in the design abstraction they have to be applied with a higher precision.

The micro-architecture exploration and synthesis methods may have different character dependent on the set and nature of applications and architectures considered, as well as the modelling, analysis and synthesis techniques used. For the data dominated, streaming and mixed applications data flow graph-based modelling techniques are typically used (e.g. a CDFG or HCDG), while control-dominated applications use Petri-net or FSM-based modelling techniques. Currently, two basic micro-architecture classes commonly considered are extensible or reconfigurable application-specific instruction set processors (ASIPs), and (re-)configurable coarse and mixed-grain hardware accelerators. For ASIPs the (final) micro-architecture exploration and synthesis is performed through the construction, configuration or extension of an application-specific instruction set, and of its corresponding hardware. Due to the fact that currently this is a very hot research issue of high practical relevance, it is discussed in a special separate section of the paper.

5.4.2. Hardware accelerator design
For hardware accelerators, the micro-architecture exploration and synthesis is supported by the methods of application analysis and parallelization and high-level synthesis (HLS). For a given specification of the application behaviour and of the parametric constraints and objectives, hardware accelerator synthesis creates a corresponding RTL-level hardware structure that realizes the application behaviour required and satisfies the constraints and objectives. In general, this structure involves a data-path and control-path. The data-path represents a network of computation, memory and interconnection components, and the control-path one or more collaborating control automata. In the case of the most popular synchronous systems, the control automaton is a finite state machine (FSM). Consequently, hardware accelerator synthesis can produce a single (in most cases coarse-grain) hardware accelerator or a network of collaborating (mixed-grain) accelerators [23,254,255].

In the past there has been a substantial body of research and development in the coarse-grain acceleration, especially related to the loop extraction and their parallel hardware implementation. For instance, the Nimble compiler [256] extracts loops for hardware acceleration and performs hardware-oriented loop optimizations (e.g. loop unrolling, fusion, and pipelining) to generate multiple optimized versions of each loop. Subsequently, it decides which loops and which loop version will be actually implemented in hardware using a quality metric accounting for the execution times in hardware and software, and hardware reconfiguration time. In a similar way, BRASS [45,257] and DEFACTO [46] select and process loops for their realization in hardware. Also Weinhardt and Luk investigated innermost loop synthesis using some loop transformations techniques for recon-figurable co-processors through pipeline vectorization [258]. Derrien and Rajopadhye considered loop tiling for reconfigurable accelerators [259] and Schreiber at al high-level synthesis of hardware accelerators [260].

The three main HLS tasks are: resource allocation, operation binding and operation scheduling. Resource allocation consists in deciding the quantity of hardware resources of each type. Operation binding (assignment) consists in the mapping of one or more operations on a given allocated resource instance. Binding of several mutually exclusive operations to one resource instance realizes resource sharing and facilitates hardware minimization. Scheduling determines the temporal ordering of operations through assigning each operation to a given time (control) step that corresponds to a state in the related control FSM. Two specific scheduling algorithms are ASAP (as soon as possible) and ALAP (as late as possible). The difference between the ALAP and ASAP time steps of a given operation represents the scheduling freedom of the operation. ASAP optimally solves the unconstrained scheduling problem and its run-time is proportional to the number of vertices and edges in the scheduled DFG. The constraint scheduling problem is NP-complete [117,261]. Therefore, heuristic algorithms are usually used to solve it efficiently, as e.g. list scheduling [262], trace scheduling [32], force directed scheduling [263], percolation scheduling [264], etc. Scheduling can be performed before, after or together with binding. Since scheduling and binding are strictly interrelated, several methods have been proposed to deal with their interdependencies [42,265,266] or to perform them together [267]. It is out of the scope of this paper to overview even the most important HLS techniques, because HLS is well established for the data-dominated applications and fixed or compile-time-reconfigurable systems [42,268,269], and numerous papers were published in this field. However, HLS still requires adequate extensions for the control-dominated and mixed applications, speculative execution and conditional resource sharing [270,271], as well as, for the run-time reconfigurable systems. Moreover, most of the HLS methods were developed for the former ASIC technologies. They made assumptions on costs of different kinds of hardware resources that do not hold for the modern nano-dimension technologies and FPGAs. For instance, the interconnection costs of secondary importance in traditional ASIC technologies are dominant in the modern nano-dimension technologies and FPGAs, simple arithmetic units (e.g. adder) traditionally assumed to be much more expensive than multiplexers or registers are of comparable cost in FPGAs, etc. Also power and energy minimization is very important for modern applications and implementation technologies, with static (leakage) power becoming more and more important factor. Consequently, while traditional HLS methods were focused on the optimization of a certain trade-off between the hardware speed and arithmetic resources needed, the HLS methods required for the modern reconfigurable systems have to perform the multi-objective optimization of the overall architecture, while adequately addressing the growing interconnect and power/energy consumption importance. Below we will briefly overview the recent research in the fields that require re-work or extensions.

Some of the HLS methods that account for interconnect and other resource minimization are discussed in [272–279]. Most of them are focused on bit-width aware scheduling and binding or multiplexer cost minimization during binding. However, one should also not forget the importance of accounting for the complexity of controllers necessary for the implementation of resource sharing, conditional execution and other concepts, their interrelations and interconnections with the controlled datapaths, as well as for the overall hardware architecture refinement and optimization at the RTL level between the traditional output of high-level synthesis and input of the actual circuit synthesis.
Adequate addressing of these issues can give substantial resource reduction effects [280,281]. For applications involving both data and control dependencies specific scheduling heuristics [269, 289–289], as well as, speculative execution [284,286,288–291] and conditional resource sharing [282,283,287,289,291] techniques have been proposed. Since scheduling, allocation and binding are strictly interrelated, a global HLS approach has also been proposed, which performs them together and solves the problem of conditional behaviour synthesis under resource constraints [267]. In this approach, the mixed conditional behaviours are represented using HCDSs [123,124,289], and the Java Constraint Programming package (JaCoP) [292] that involves domain-specific constraints is used to solve the global HLS problem. Experimental results demonstrated that this approach is able to produce very good results, what once more reconfirms the usefulness of the constraint programming approach in solution of such complex discrete optimization problems.

Traditional HLS tools work for a static hardware and require a substantial extension to support the run-time reconfiguration, and particularly, to account for the temporal partitioning and reconfiguration overhead. One of the first works on scheduling and temporal partitioning when accounting for the reconfiguration time is [183]. In [183,293] temporal partitioning, scheduling and clustering is discussed for DFG modelled behaviours, and in [185,214,294] for mixed behaviours. In [295] a tabular method is proposed for synthesis of reconfigurable control-dominated systems. It should be stressed that algorithm parallelization techniques, pipelining, as well as scheduling and binding optimization of the loop bodies are the main techniques that have to be used in combination to create high-quality coarse-grain hardware accelerators. Consequently, these techniques have attracted much attention in recent research and development in the field of the coarse-grain accelerator synthesis [23–25,255, 267,296–305].

Also different accelerator core generators and libraries of accelerator cores are available for FPGAs. For instance, the SPIRAL project delivered behavioural Verilog generators for several signal processing functions [306,307]. Xilinx provides Core Generator [308] and Altera offers MegaCore [309]. The generated hardware can eventually be optimized through generation of several different versions of an algorithm.

5.5. Reconfigurable ASIP design

Modern embedded systems require highly effective and efficient computing platforms to satisfy the growing computational demands, parametric requirements and complexity of the applications, but at the same time they demand highly flexible customizable platforms to decrease the development effort, costs and time-to-market, and to rapidly adapt to changes. An application-specific instruction set processor (ASIP) is a processor of whose architecture and instruction set are or can be optimized to a specific application or to a relatively narrow application domain. Being software-programmable, it provides a high degree of flexibility, but due to its customization, it can offer to particular applications a substantially higher effectiveness and efficiency than a general purpose processor. In recent years, configurable ASIPs [310–316] have been introduced that allow for configuration and/or extension of their instruction sets for a specific application, through customization of their instructions and corresponding functional units or introduction of new custom instructions and functional units. Comparing to the basic static ASIPs, they offer an additional flexibility of the computation platform adaptation to a specific application. Their hardware is often sub-divided into two parts: static (corresponding to a minimum ISA) and configurable. The configurable part is then customized to a given application and implemented together with the static part as an ASIC or a part of a SoC, through a regular chip synthesis process.

Unfortunately, introduction of the nanometre CMOS technologies have resulted in a large increase of chip development and fabrication costs. This caused a big increase of the attractiveness of programmable hardware technologies compared to hardwired ASIC or SoC technologies, and a rapid increase of the share of the FPGA-based systems among the newly started designs, including high-volume designs. Consequently, for many applications reconfigurable ASIPs, in which a processor core is tightly coupled with reconfigurable functional units implemented in programmable hardware (e.g. FPGA), have become more attractive than one-time configurable solutions [317–319]. Programmable hardware allows the designers to use a single low-cost standard programmable chip (e.g. FPGA, platform FPGA or SoC with embedded FPGA) to implement very different application-specific architectures and instruction sets through simple chip reprogramming. This makes possible a relatively easy, effective and efficient adaptation of processors to different applications and various changes. Additionally, re-targetable optimizing compilers allow for an effective and efficient compilation of the high-level application specifications to their corresponding code executed on the optimized reconfigurable ASIP platforms.

Moreover, in most configurable ASIPs the instruction-level parallelism exploitation was limited and mainly implemented through constructing VLIW processors and simple single pipelines. Even the well-known Tensilica's tool for Xtensa processors [313] or ASIPMeister [320] initially only supported single pipeline designs, and a new instruction set that enables limited multipipeline designs was introduced by Tensilica not a long time ago. However, the modern complex embedded applications, as e.g. the multimedia or wireless communication applications, involve many different parts, each with its own character and requiring a specific kind of processing for its optimal implementation. Consequently, heterogeneous multi-pipeline ASIPs are required for an adequate implementation of the modern complex embedded applications, with each single pipeline especially synthesized for a corresponding application part [18,321]. Also, different parts of a complex application may require different grain acceleration, what may result in a mixed-grain acceleration structure to optimally implement the whole application. Reconfigurable hardware (e.g. FPGAs, platform FPGA or SoC with embedded FPGA) very well facilitates implementation of such reconfigurable heterogeneous multi-pipeline structures (see e.g. [18,254,321,322]). Currently, the commercially available reconfigurable processors include: Xilinx MicroBlaze [323], Altera Nios II [324], and Stretch S5000 and S6000 [325] processors.

Usually, a reconfigurable ASIP implemented in reprogrammable hardware has a much lower performance, larger area, and substantially higher power consumption than a corresponding configurable ASIP implemented as an ASIC or SoC. It can be as much as 3–5 times slower and up to 35 times larger [326]. However, for a large share of embedded system designs, the very big NRE cost reduction comparing to an ASIC or SoC implementation, extra flexibility and independence of the chip producer compensate for these losses. Moreover, the difference in performance, area and power consumption can be significantly reduced through:

- exploitation of the coarser-grain and more specific reconfigurable fabrics involving less reconfiguration resources, and enabling a higher performance and faster reconfiguration,
- hardware instruction/configuration caching, and
Summing up, it is envisioned that the importance of reconfigur-
ability) ASIP synthesis can be a good starting point for further
applications on such a platform.

Configuration caching consists in storing multiple configura-
tions on a chip, and switching among them at the run time. It was
most probably introduced by DeHon [328]. In [329], instruction
 caching is presented for which a configuration table is used to
to check if the required instruction is already configured. If it is
configured, it is immediately executed. Otherwise, its configura-
tion stream is loaded replacing previous configuration data in
some segments of the reconfigurable fabric, and subsequently the
instruction is executed. Hardware configuration caching can also
be implemented using instead of segments several configuration
planes assigned to different instructions and switching among the
planes. In the past it has been explored in relation to 2D
reconfigurable devices [197,330,331]. Recently, exploitation of
3D integration technologies has been proposed for configuration
caching, which has a potential to dramatically increase the
reconfiguration speed, thus eliminating one of the main barriers
of the broad RC proliferation [332,333]. Also several other 3D
FPGA architectures have been proposed recently [334,335].
Summing up, it is envisioned that the importance of reconfigur-
able ASIPs for the modern embedded system implementation will
further grow [336,337]. The major challenge of their broad
proliferation is, however, effectiveness and efficiency of their
synthesis that involves two basic tasks: their (final) micro-
architecture construction, and application mapping on the
constructed micro-architecture. Consequently, their broad prolif-
eration strategically depends on the availability of adequate
design automation tools that will enable effective and efficient
application analysis, reconfigurable ASIP micro-architecture de-
sign exploration, ASIC platform construction, and mapping of
applications on such a platform.

Fortunately, it is not necessary to start the research and
development of design methods and tools for reconfigurable ASICs
from scratch. Methods and tools earlier developed for (configur-
able) ASIC synthesis can be a good starting point for further
research and development needed for reconfigurable ASICs.
However, it is also necessary to account for the fact that the
development methods and tools for the reconfigurable ASICs have
to satisfy partly different and more stringent requirements than
those for (configurable) ASICs, such as e.g.:

- different implementation technology, and specifically, much
  more freedom of architecture (re-)design, configuration and
  run-time reconfiguration, and consequently, much richer
  system concepts and architectures possible, and much more
  complex and comprehensive development methods and tools
  required;
- partially different developer and tool user community, invol-
v- not only the selected highly advanced specialists in
  processor architecture, software, and ASIC chip development,
as in the case of regular ASICs, but mainly much less advanced
gen- eral developers of FPGA-based (embedded) systems, and
  consequently, more completely automatic, and much more
  user-friendly development methods and tools are required;
- since the reconfigurable ASIC development is required to be
  more automatic, and will not be performed by highly advanced
  specialists only assisted by some automation tools, not only
  the automation, but also the quality of the development
  methods and tools has to be substantially higher.

Consequently, the methods and tools developed in the past for
(configurable) ASICs cannot just be reused for the reconfigurable
ASICs, but have to be adequately and substantially extended and
enhanced. Nevertheless, since they constitute a good starting
point for further research and development, we will briefly
comment on them.

Two main micro-architecture development approaches for
(configurable) ASICs are the template-based and language-based
approaches. The template-based approach uses dedicated para-
meterized architecture templates. In this approach the architec-
ture exploration and modification is very limited, mainly to the
precise instruction and data formats, as well as selection and
limited extension of operations. The main processor architecture
remains unchanged. On the other hand, the compiler and
simulator generation is easy. This approach is used for example
for Xtensa [313] and Jazz [314]. The language-based approach
exploits specific architectural description languages (ADL)
[338–342]. In this approach the main processor architecture can
be changed to some degree, and the architecture modification
through changes in ADL specification is quite easy, but it is limited
by the features of a particular ADL and often time consuming.
This approach is used for instance in Chess [343,344], HiveFlex [316],
and ASIPMeister [320]. For ASIC architecture exploration and code
generation, similar partitioning, scheduling, retiming and binding
methods can be used to those discussed in Section 5.4.2, as
presented for instance in [47,345–348]. More information on
(configurable) ASICs, their design and its automation can be found
in [349–361].

5.5.1. Instruction set customization

Instruction set customization is one of the most effective ways
of processor effectiveness and efficiency enhancement for a given
application or application domain. Its aim is to design an
instruction set for an application-specific processor that satisfies
the hard constraints and optimizes the objectives of the quality
model of the micro-architecture synthesis issue. In practical
terms, this instruction set optimization process enables the
embedded and high-performance processors to achieve signifi-
cant performance and power efficiency by a limited use of extra
hardware resources.

The instruction set customization can involve construction of a
whole application-specific instruction set of a new processor
[354,355] or modification of an instruction set of an existing
processor, mainly involving its extension through construction of a
new application-specific instruction sub-set [355,357–364]. In
the first case, it is referred to as full customization, and in the
second case as partial customization. To perform the customiza-
tion process, the original HLL application specification (e.g. in C
or C++) is usually first converted into a graph-based representation,
typically DFG, CFG, HCDG or similar. This graph-based applica-
tion representation is analyzed, parallelized, and if needed,
scheduled and assigned, and its critical and repeating sub-graphs
are then transformed into application-specific instructions and
adequately implemented in hardware to minimize execution time
or maximize throughput, when observing the hardware resources
required. Sometimes, several application-specific instructions are
implemented jointly to share the hardware costs. Although the so
constructed hardware of custom instructions can be very effective,
the cost and time of designing a whole new processor for each
application is usually excessive. Consequently, a (re-)configurable
approach and extension of an instruction set of an existing
processor or hard/soft processor core is preferred. In this case, the
hardware accelerators implementing the extra instructions are
added to the existing processor/core. The customization involves
synthesis of a new hardware strictly for supporting the computa-
tions required by a particular application, and the application
mapping on a partially customized processor involves decisions
on which application operations will be implemented through the
newly synthesized hardware and which through the software (instructions) of the pre-existing processor/core. From this it should be clear that the instruction set customization and application mapping constitute a special case of the HW/SW co-design, and in consequence, this process is often referred to as HW/SW co-design or partitioning [365–368].

Before considering the instruction set customization, several remarks have to be made on basic instruction types, and instruction execution in processors of different types. From the viewpoint of the instruction and related accelerator granularity, the instructions can be sub-divided into: fine-grain and coarse-grain. The fine-grain instructions implement small groups of basic operations (e.g. multiply accumulate) [356–363,369–371]. The coarse-grain instructions implement large blocks of basic operations, as whole (nested) loops or procedures (e.g. codecs, discrete filters or transforms) [23–25,255,267,296–305]. Being small clusters of basic operations, the fine-grain instructions can be reused in more applications than the coarse-grain, but the effectiveness of the coarse-grain instructions, and specifically speed-up and energy reduction, can be much higher than this offered by the fine-grained one. In general, the fine-grained approach is thus preferred when considering processor specialization to a large application class covering applications with various characteristics, while the coarse-grain approach is preferred when specializing processor to a particular single application or small class of similar applications. Nevertheless, one should remember that the more closely the operation patterns of the application-specific instructions resemble the most frequent and critical operation patterns of a particular application, the higher speed-up and energy gain can be expected. Consequently, the type of instructions considered should not be pre-decided, but should be decided after a careful analysis of the application or application class at hand and related design requirements. In many cases, a mixed fine-/coarse-grain hybrid approach will be perhaps the most appropriate [372–374]. Moreover, the instruction type selection is related to instruction execution in processors of different types. In general, the simpler is a given processor and the closer the application-specific instruction resembles the basic instructions of the processor, the easier is the integration of the new instruction into the processor. For instance, independent of the instruction character, the application-specific instruction integration is quite straightforward for the simple RISC and CISC processors that do not exploit the instruction-level parallelism. The processor can just wait with execution of the next instruction when the “ready signal” from the accelerator implementing a given application-specific instruction (e.g. [375]) or can perform some other unrelated computations when “actively waiting” on the “ready signal”. The more complex parallel VLIW processors have usually instructions with a fixed execution time. In consequence, the extra instructions should also have the same execution time, because various instruction execution times would cause stalls in a VLIW pipeline and degrade its performance. This problem does not exist for superscalar processors, where the hardware accelerator implementing an application-specific instruction can be used the same way as a regular functional unit of the superscalar [376]. Unfortunately, superscalar processors often use complex hardware to dynamically identify parallel processing opportunities, and in consequence, are less suitable for embedded applications. In contrary, VLIW processors exploit compilers for finding instructions for parallel execution, and consequently, have usually much simpler hardware and better suit embedded applications. Here a remark has to be made that accelerators can also be exploited without any explicit changes to the instruction set architecture of a processor, for instance as discussed in [377,378].

The major challenge of the instruction set customization is the availability of adequate design automation tools that will enable an effective and efficient automated application analysis and instruction set customization. In the current engineering practice, the application profiling and other kinds of analysis are quite well supported by compilers and other automatic tools, but the instruction customization is often limited, and either performed manually or only partly automated for some specific configurable ASIP architectures (see e.g. [312,314,317,320,338,340,343,367,368]).

In the reconfigurable ASIP area the situation is much worse. Basically, the soft processor cores of the major FPGA vendors, as Xilinx or Altera, have to be manually tuned to applications. Although the recent Mimosys Clarity tool [379,380] automatically identifies hardware accelerators from C code and automates the HDL generation and implementation of an application on the PowerPC and accelerators in XILINX FPGAs, it only delivers a single set of accelerators and does not enable any broader design space exploration for finding different sets of accelerators that could meet various constraints, optimize different objectives and realize different tradeoffs among the objectives. It is just a step in a good direction. Another related tool, Synfora’s PICO Express FPGA synthesizes (hierarchical) coarse-grain application-specific accelerators (instructions) for implementation in Xilinx Virtex and Spartan FPGAs. It performs algorithmic synthesis of C algorithms into their corresponding optimized RTL code that is further synthesized into the actual FPGA hardware with the tools of Synplicity and Xilinx. It makes possible a specific design space exploration, creation of multiple implementations characterized with area and performance estimates, and trade-off analysis [381]. The recent research and development effort related to the design and design automation of the coarse-grain accelerators is discussed in Section 5.4.2. Yet another recent development in the reconfigurable ASIP field, the integrated development environment (IDE) of Stretch, partially automates the instruction set extension and application mapping on the Stretch families of S5000 and S6000 processors based on Xtensa and having an embedded reconfigurable instruction set extension fabric (ISEF) within the processor. The developers of systems based on the Stretch processors profile their applications expressed in C/C++, using the Stretch profiler, identify the parts of application code that have to be accelerated in ISEF, and appropriately annotate the C/C++ application code. These parts are then implemented as new instructions that are executed in a single cycle. From the annotated code, the Stretch compiler produces both the ISEF configuration and optimized application code, and configures the ISEF automatically [146,325].

Despite all the previous effort, no acceptable solution exists to the problem of fully automatic application analysis, reconfigurable ASIP instruction set customization, customized ASIP platform construction, and mapping of applications on such a customized platform. Since this problem is of high scientific interest and practical relevance, it represents a very hot research topic, and numerous research results related to this topic have been published recently [23,25,101,120,181,239,255,296–305,321,362,363,374,377,378,382–389,390–417]. An adequate full automation of the above processes is necessary due to many factors, including the growing complexity and requirements of application, designer productivity gap and short time to market requirement, as well as NP-hard character of the problems to be solved and complex tradeoffs to be resolved by the selection of an optimized custom instruction set from a usually huge set of candidate instructions and by the application mapping.

As already mentioned, instruction set customization is usually performed when using a graph-based (e.g. DFG, CDFG, HCDG) representation of the application. Before starting the actual
instruction set customization, various transformation and parallelization techniques are often applied to the graph-based application representation (e.g. [411,412,414]). Instruction set customization is usually performed in two steps, namely: custom instruction identification and custom instruction selection. In the literature related to reconfigurable ASIPs, instruction set customization is usually limited to instruction set extension, although in general, it should consider the elimination of less useful instructions and of a related hardware as well, of course, if this is at all possible.

5.5.2. Custom instruction identification

Custom instruction identification consists of analysis of a graph-based application representation to find some critical and repeating sub-graphs (operation patterns) that are good candidates to be converted into single custom instructions. The search for the candidate patterns (instructions) is usually guided by quality metrics. The quality metrics should express the effectiveness and efficiency of a pattern after its realization as hardware of the corresponding application-specific instruction. The aim of these metrics is to facilitate the identification of the most promising candidate patterns in the application graph, as well as to support pruning of the set of found candidate patterns during the instruction identification, if this set is becoming too large. The metrics are usually quite complex, and involve hard constraints imposed by the type of architecture used, implementation technology or guarantee of a proper scheduling (e.g. related to the number of inputs and outputs, operation convexity, operation type, etc.), as well as some optimization objectives related to various characteristics of the hardware (instruction) implementing a given operation pattern, as the execution time, power/energy consumption, hardware resources used, etc. Unfortunately, in the instruction identification methods of the published research, proxy metrics are used commonly that often do not well express the actual pattern effectiveness and efficiency required. For instance, in [99,359–361,372,373,418–420] instruction identification methods have been proposed that aim at maximum spatial reuse of patterns and mainly based on the number of pattern occurrences in the application graph. This often results in quite small and simple patterns that do not guarantee a sufficient execution speedup. Several works showed that larger and more complex patterns often result in higher speedups (e.g. [361,382,412,422]) and different metrics have to be used to achieve them. For higher speedups, not the spatial reuse of patterns, but rather their temporal reuse should be maximized. To reduce the average execution time, the most frequently executed patterns should be identified, as e.g. patterns on the most frequently executed paths [100,421]. To reduce the worst-case execution time and satisfy hard real-time constraints, the patterns most frequently occurring at the critical and near-critical paths should be extracted [402,422]. In [402] the worst-case execution time is used as a metrics to guarantee satisfaction of the real-time constraints. In [386,387,403] dynamic reconfiguration cost is accounted for, and in [410] the extra clock cycles to move data between the register files and hardware units implementing the new instructions, as well as hardware sharing among the hardware units. Some other approaches [99,100,418,419] use specific heuristics to identify some promising patterns while discarding some less promising ones. The method proposed in [100] and several other methods find possible custom instruction candidates, while pruning the search space based on the input or output constraint violation by the candidate sub-graphs, operation type, convexity, etc. The methods proposed in [16] and [382] incrementally grow patterns when observing performance gains and penalties related to the input or output constraint violation.

A commonly used concept in the custom instruction identification is this of template. Template is an operation pattern known or assumed to be a promising candidate for a custom instruction. Custom instruction identification can be performed as template matching or template generation. Template matching assumes the existence of a template library and consists of finding the number of occurrences in the application graph or the number of repetitive executions of particular existing templates from the template library (e.g. [355,382,423]). The most frequent templates are then implemented as custom instructions. This problem is similar to the sub-graph isomorphism problem [424–426], and it is known that the directed sub-graph isomorphism problem is NP-complete [117]. Template generation consists of creating new templates (e.g. [356,359,360,363,370,371,374,391,393,394–402,404–410,412,425–428]). Usually it starts with selection of a particular node or a larger pattern to be a seed, and gradually grows the seed through absorbing some neighbouring nodes, when observing the influence of the pattern growth on its parameters included in the constraints and objectives of the pattern quality metrics that guide the search. The pattern optimizing the quality metrics is accepted as a new template. After constructing one or more new templates, the number of template occurrences in the application graph or the number of their repetitive executions is checked to prune the less frequent templates or to accept the most frequent once. Some approaches that combine the template matching with generation have also been proposed (e.g. [359,360]).

In general, the problem of custom instruction identification is of exponential complexity, because the set of possible new custom instructions grows exponentially with the number of the application graph nodes, as a graph with $n$ nodes contains $2^n$ sub-graphs. In the past, exhaustive enumeration, several dynamic program-based algorithms (e.g. [359]), and Integer Linear Programming or ILP-based algorithms (e.g. [362,393]) have been proposed to solve the problem, but these approaches are not efficient for larger general problem instances [100]. Consequently, to effectively and efficiently solve this problem for large instances, only some easier to process specific application graphs and/or sub-graphs should be considered, or adequate heuristic algorithms have to be used.

The solution difficulty of the instruction identification problem depends on the kind of application graphs and sub-graphs (templates, operation patterns, instructions) considered. In particular, since cyclic graphs cannot be easily sorted, acyclic graphs are considered in most cases. Although a cyclic graph can be transformed into an acyclic one (e.g. through unrolling the cycles), this significantly increases the graph complexity. Also only connected graphs are considered in most cases (e.g. [101,359,360,382,429]), and disconnected graphs are processed in parts through processing their connected components, despite the fact that the direct consideration of the disconnected graphs makes possible a more effective parallelism exploitation (e.g. [361,363,382,429]). Also consideration of multi-output templates and overlapping templates during the custom instruction generation and selection is difficult [100,429]. During instruction generation the disjoint templates are usually considered, and the nodes absorbed into a template are immediately removed from the application graph. Although the overlapping templates consideration can potentially produce better results, their consideration drastically increases the problem difficulty, and additionally, the costs related to replication of the common nodes of the overlapping templates may sometimes exceed the performance gains due the overlapping template consideration.

Regarding the number of outputs of a sub-graph (template, operation pattern, instruction) the following two types of sub-graphs can be distinguished: multiple input single output (MISO)
and multiple input multiple output (MIMO). MISO sub-graphs of maximal size are called MAXMISO. The type of patterns or instructions considered directly relates to the instruction identification problem complexity. The exhaustive enumeration of MISO patterns is exponential, as it is strictly related to the sub-graph enumeration problem which is known to be exponential [100,369]. However, the exhaustive enumeration of MAXMISO patterns is linear in the number of nodes [427], because the intersection of MAXMISO patterns is empty. In [386] the enumeration of patterns is based on the number of inputs, outputs, area and convexity, but is limited to the connected sub-graphs. This limitation is eliminated in [429]. In [363] the identification of convex MIMO instructions is presented through clustering of MAXMISO instructions to maximally exploit the MAXMISO-level parallelism. In this algorithm, the convexity is guaranteed by construction. Through extension of this algorithm, a heuristic linear complexity algorithm has been constructed for identification of convex MIMO instructions [370]. Since direct implementation of single MAXMISO instructions does not result in significant performance gains, this algorithm is based on combining MAXMISO instructions per levels, i.e. several MAXMISO instructions of the same level of a reduced graph are combined into one convex MIMO instruction. Further results based on this idea are presented in [369,371] and a framework for the automatic generation and selection of convex MIMO instructions in [428]. Another research on MIMO instructions is presented in [394]. Some other papers present sub-graph enumeration algorithms limited to only the so-called legal patterns which are the convex sub-graphs that satisfy some architectural constraints, as number of I/O operands, pipeline depth, and other constraints [146,391,392,396,418,419,421,430]. While the number of all sub-graphs is exponential, the number of legal sub-graphs is polynomial [146]. In [400] it has been demonstrated that the number of different prevalent data-flow patterns in popular multimedia benchmarks is very limited (approximately 10 patterns). In [395,396] this has been experimentally proven for the second time. It has been demonstrated that a relatively small number of predefined templates, called morphable structures, is needed for a near-optimal instruction set customization for the relatively narrow multimedia application class, and a rapid custom instruction generation method is presented based on this fact. The same is proven for the third time in [397]. A similar idea of the custom instruction generation speedup through only considering the major blocks of CDFG is presented in [398]. In [373] an instruction generation algorithm is proposed for hybrid reconfigurable systems. It uses pattern appearance frequency in the application graph to guide the instruction generation, iteratively absorbs the edges to templates, and finds the most frequent templates irrespective of their size. Its extension to parallel templates is discussed in [360]. Another algorithm for finding the convex patterns resulting in the maximal speedup is proposed in [431]. It identifies the patterns with any specified maximum number of outputs, and any set of disconnected sub-graphs satisfying some input, output and other constraints can be identified for their parallel execution. Identification and selection are combined. In [17] a scalable method of custom instruction identification is proposed. The search program is sub-divided into several smaller programs corresponding to smaller search spaces. The solutions for the smaller search spaces are subsequently combined into the total solution. The area consumed by particular instructions is reduced through sharing the usage of hardware modules by several instructions. The template area is computed and only the templates satisfying the area constraints are selected. Another scalable custom instructions identification method is discussed in [429]. A fast iterative improvement-based instruction generation technique using principles of the well-known Kernighan–Lin min-cut heuristic is proposed in [399,400]. In [419] several exact and approximate algorithms for the instruction set extension are discussed.

From this brief overview it should be clear that the above-discussed algorithms for instruction identification are efficient due to imposing extra constraints on the application graphs and/or sub-graphs considered, and in consequence, only considering their specifics makes it easier to process sub-sets, or due to exploiting some approximate heuristics. This, however, results in exploration of only a sub-set of possible solutions which may not include the optimal or even near-optimal solutions. Consequently, the algorithm effectiveness improvement can be obtained through a careful relaxation of some of the constraints to a reasonable degree or development of better heuristic methods. One of the first works that proposed control-flow constraint relaxation through instruction candidate identification across basic blocks was the paper by You and Mitra [100]. Also memory operations limit the size and kind of instructions in the previously discussed approaches. In [388,404] introduction of local memory to the custom-processing units implementing new instructions is proposed to relax these constraints. In [389] combination of register file access serialization with pipelining has been proposed to relax the register-file port constraints, and in result the sub-graph I/O constraints, and it is exploited in a new instruction identification method [405]. The same idea is exploited in [406,408,410], and a similar idea using data forwarding in [407], and using shadow registers in [409]. Also a new promising method for instruction identification that is based on constraint programming has been proposed recently [425].

5.5.3. Custom instruction selection

Custom instruction selection consists in selecting the most promising sub-set of custom instructions from the set of custom instructions constructed in the process of custom instruction identification. This is realized using some quality metrics involving evaluation of an instruction sub-set in terms of performance, area, power consumption, etc. after its realization in hardware. These quality metrics are similar to those used for the instruction identification, but at this time the metrics should express the effectiveness and efficiency of an instruction set and not the quality of particular instructions. Similarly as in the case of the instruction identification methods, in the instruction selection methods of the published research, some proxy metrics are used that often do not well express the actual pattern set effectiveness and efficiency required.

For instance, in [99], an algorithm is presented that aims at selection of a minimal set of instructions that maximizes the number of covered nodes in the application graph. An interesting observation of the experimental research of this work is that increasing the number of different instructions (patterns) used to covering results in a significant increase of the number of nodes covered only up to a certain level, above which, usage of more templates does not substantially influence the number of nodes covered. Consequently, a reasonably small number of well-designed and adequately selected custom instructions can result in a significant performance gain. Other methods aim at minimizing the number of distinct instructions used [356], maximizing the number of pattern occurrences [382,423] or execution frequencies [101,402,421,422], or account for the occurrence of specific nodes [16,360] or for resource sharing [432–434].

For an effective and efficient custom instruction identification and selection, effective and efficient estimation methods are necessary for estimation of the basic characteristics of the accelerator hardware related to delay, area, power consumption,
etc. For this purpose, the same or similar estimation methods and tools can be used, as for instance those presented in [98,101–112,114–116,435].

The instruction selection problem is a specific graph coverage problem that is known to be NP-hard [117]. Consequently, to solve this problem for large and complicated instances, some heuristic algorithms have to be used. In the past, several exact LP-based algorithms have been proposed [168,362,363,390] for instruction selection and some branch-and-bound-based algorithms [410], as well as some branch-and-bound-based algorithms for general covering problems [434,437,438] and some effective and efficient heuristic algorithms for similar coverage problems [439]. Also, some heuristic methods have been proposed for instruction selection (e.g. [364,366,419]). Especially the method discussed in [364], which is based on constraint programming and performing the final instruction selection during scheduling and mapping, seems to be very promising. Moreover, some methods for instruction set customization for various specific domains have been proposed, as e.g. for video processing [414,415], multimedia [416] or network applications [417].

Currently, new instruction set customization methods for hybrid reconfigurable systems [374,413], synthesis methods for multi-pipeline heterogeneous (re-)configurable multi-processors [18,50], automatic tools for soft-core customization [440], as well as integrated platforms [441] and complete fully automatic tool-chains [436] for heterogeneous embedded multi-processor systems involving (re-)configurable embedded processors and reconfigurable hardware accelerators are under research and development.

**5.5.4. Re-targetable compilers**

A very convenient way to generate code for a modified processor is to use a re-targetable compiler [442], i.e. a flexible compiler that has been especially designed for its easy modification to allow code generation to various specific processor architectures (targets). Usually an architecture description language is used to implement this flexibility. One of the best known examples of a re-targetable compiler is the GNU compiler collection (GCC) that supports several HLLs and very many computing platforms [443]. Examples of known re-targetable compilers that generate code for different instances of parameterised and/or extendable ASIPs for embedded hardware/software systems include: HiveCC [316], CHESS [343,344], AVIV [338], and RECORD [444]. An interesting easy-targetable and highly flexible compiler development system is CoSy from ACE associated compiler experts [445]. It serves for development of target-specific optimizing compilers and cross-compilers for a broad spectrum of targets and processor architecture exploration. It enables efficient construction of high-quality compilers and exploration of compilation effects due to architecture variations, enabling this way HW/SW co-design. Since the typical use of the re-targetable compilers is compilation for new (sometimes still under development) platforms or for embedded systems that involve several different or modified different processors, or in which it is impossible or inconvenient to perform compilation, the most of them are cross-compilers, i.e. compilers that generate code for a different platform than the compiler itself is run. In embedded system compilers, in parallel to re-targetability, an adequate support for satisfaction of real-time constraints and other physical constraints and objectives is of primary importance, with power and energy consumption minimization receiving an increasing attention recently. HW/SW partitioning, instruction selection and scheduling, instruction and data encoding, memory traffic and many other factors can have a substantial impact on the power/energy consumed. Power minimization in compilers requires development of new modelling, analysis and code generation techniques. Additional information on re-targetable compilers for embedded systems can be found in [442,444].

**6. Conclusion**

As discussed in the initial part of this paper, many new opportunities, but also many new difficulties to solve issues have been created through introduction of the nano-dimension CMOS technology. Moreover, increasingly complex and sophisticated information-processing systems are required to reliably perform real-time computations to extremely tight schedules with energy and area efficiency never demanded before. There is a general consensus that the progress in microelectronic technology alone cannot guarantee satisfaction of the growing computational demands, physical requirements and economical challenges of most modern applications. To effectively exploit the opportunities created by the modern technology, new efficient application-specific system architectures and circuit implementations have to be used, exploiting more adequate concepts of computation, storage and communication. The reconfigurable system approach exploiting a mixture of the traditional CPU-centric instruction-stream-based processing with the decentralized parallel application-specific data-dominated processing and application-specific (re-)configuration can significantly help in resolving some of the difficult issues and in satisfying the stringent requirements of the modern applications.

The area of reconfigurable systems is very promising, but quite new area that has been under development for only somewhat more than 15 years. New opportunities have opened for the developments in this area through introduction of the SoC technology, and a big progress has been made in the recent years. Important research results related to many different kinds of reconfigurable systems have been published, and many different reconfigurable devices and computers became commercially available [1]. Nevertheless, multiple aspects of the RC systems, and especially of their development and their supporting tools still belong to the open research or development topics. In this paper, we overviewed the recent developments and development trends in the design methods and synthesis tools for reconfigurable systems. We discussed the need of enabling development technology for RC systems and introduced the platform-based RC system development approach. Subsequently, we focused on the platform-based RC system architecture synthesis, when discussing the issues of the application analysis and restructuring, as well as, macro- and micro-architecture exploration and synthesis. We devoted a special attention to the important and currently hot issues of run-time reconfiguration, hardware accelerator design, and (re-)configurable ASIC design. Finally, we also commented the important technology of re-targetable compilers. From the overview it should be clear that a great progress has been made in the area of RC system development methodology and tools in only the recent years. It should, however, also be clear that much more research and development is needed in this field.

Although some of the current major drivers and requirements direct the research and development in the reconfigurable system area towards the coarse-grain and heterogeneous architectures, and the run-time reconfigurable systems, the fields of fine-grain architectures and statically reconfigurable systems remain important research and development areas. Many of the modern applications naturally require the fine-grain or hybrid-grain reconfiguration, and do not require any run-time reconfiguration. Moreover, the overheads of run-time reconfiguration and growing on-chip resources often make the run-time reconfiguration less attractive even for the applications that could potentially profit from it. Nevertheless, the further development and sophisticated
usage of the 3D integration technology in the RC system area, as well as, of the run-time reconfiguration methods and supporting tools may change this picture in the future. Due to the development of SoC technology, the tightly coupled architectures become more and more attractive, because it is easier to achieve substantial performance gains with them, they can implement more substantial parts of computations and adequately serve more applications with the growing on-chip resources. Moreover, their programming model can remain basically the same as in the CPU-centric processors. This causes that growing attention of the RC system and EDA-tool researchers and developers is devoted to the tightly coupled approach, and to the development of an adequate design methodology and tools for the tightly coupled architectures, as for instance the (re-)configurable ASPS. Nevertheless, the loosely coupled or mixed RC systems will continue to play an important role for many applications for which large and/or massively parallel reconfigurable resources are required, and communication of the reconfigurable resources with the host processor remains limited. New opportunities have been opened for the developments in the heterogeneous RC system area through introduction of the SoC technology that enables efficient implementation of complex heterogeneous systems, and specifically an efficient communication between their sub-systems. Heterogeneous (re-)configurable multi-processor architectures (e.g. [2,18,158,161,169,170]), enabled by the SoC technology and plenty of on-chip resources, are also more and more required due to the growing complexity and diversity of the modern computationally intensive applications, mixed granularity of their data, their substantial parallelism at various levels, and the necessity of decreasing the fabrics interconnects, energy usage, power and size, and increasing performance. To adequately tackle various combinations of these issues, hierarchical multi-grain fabrics are needed involving various kinds of design-time, compile-time and run-time (re-)configurable blocks, as well as highly optimized hardwired blocks for “standard” computations of a given application field, and more “general purpose” (customizable) processor cores. In consequence, an increased research and development effort related to the methodology and tools for heterogeneous hierarchical multi-grain RC systems, as well as, software reconfiguration in multi-processor systems can be expected in the future. It should be stressed that the requirements of high communication and storage efficiency, as well as, low-energy consumption and high reliability are common to virtually all kinds of fabrics and RC systems, and many various embedded applications. Therefore, a substantial research and development effort in the RC system development methods and tools is expected to be devoted to these important issues.

Acknowledgements

The authors are indebted to Fondo Nacional de Desarrollo Científico y Tecnológico (Fondeczy), Chile, for the Fondeczy Grant for International Cooperation #7080216, which partly supported preparation of this paper. The second author is grateful to FAPERJ (Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro, http://www.faperj.br) and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico, http://www.cnpq.br).

All product names, etc. mentioned in this text are trademarks or registered names of their respective owners.

References


Lech Jóźwiak is an Associate Professor, Head of the Section of Digital Circuits and Formal Design Methods, at the Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands. He is an author of a new information driven approach to digital circuit synthesis, theories of information relationships and measures and general decomposition of discrete relations, and methodology of quality driven design that have a considerable practical importance. He is also a creator of a number of practical products in the fields of application specific embedded systems and EDA tools. His research interests include system, circuit, information and design theories and technologies, decision theory, artificial intelligence, embedded systems, re-configurable and parallel computing, dependable computing, multi-objective circuit and system optimization, and system analysis and validation. He is an author of more than 150 journal and conference papers, some book chapters, and several tutorials at international conferences and summer schools. He is an Editor of “Microprocessors and Microsystems,” “Journal of Systems Architecture” and “International Journal of High Performance Systems Architecture”. He is a Director of EUROMICRO; co-founder and Steering Committee Chair of the EUROMICRO Symposium on Digital System Design; Advisory Committee and Organizing Committee member in the IEEE International Symposium on Quality Electronic Design; and program committee member of many other conferences. He is an advisor and consultant to the industry, Ministry of Economy and Commission of the European Communities. He recently advised the European Commission in relation to Embedded and High-performance Computing Systems for the purpose of the Framework Program 7 preparation. In 2008 he was a recipient of the Honorary Fellow Award of the International Society of Quality Electronic Design for “Outstanding Achievements and Contributions to Quality of Electronic Design”. His biography is listed in “The Roll of Honour of the Polish Science” of the Polish State Committee for Scientific Research and in Marquis “Who is Who in the World” and “Who is Who in Science and Technology”.

Nadja Nedjah is an associate professor in the Department of Electronics Engineering and Telecommunications at the Faculty of Engineering, State University of Rio de Janeiro, Brazil. Her research interests include embedded systems and reconfigurable hardware design as well as cryptographic hardware. Nedjah received her Ph.D. in computation from the University of Manchester-Institute of Science and Technology (UMIST), England, her M.Sc. in System Engineering and Computation from the University of Annaba, Algeria and her Engineering degree in Computer Science also from the University of Annaba, Algeria.

Miguel Figueroa received a B.Sc., a professional degree and a M.Sc., all in Electrical Engineering, from the University of Concepción in Chile, in 1988, 1991 and 1997, respectively. He received the M.Sc. and Ph.D. degrees in Computer Science and Engineering from the University of Washington, U.S.A. in 1999 and 2005. He is currently an Associate Professor in the Electrical Engineering Department at the University of Concepción. His research interests include low-power VLSI design, and reconfigurable computing.