International Journal of Reconfigurable Computing

Selected Papers from ReCoSoc 2008

Guest Editors: Michael Hübner, J. Manuel Moreno, Gilles Sassatelli, and Peter Zipf

Selected Papers from ReCoSoc 2008

International Journal of Reconfigurable Computing

Selected Papers from ReCoSoc 2008

Guest Editors: Michael Hübner, J. Manuel Moreno, Gilles Sassatelli, and Peter Zipf

Contents

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 894059, 2 pages
doi:10.1155/2009/894059

Editorial
Selected Papers from ReCoSoC 2008

Michael Hübner,[1] J. Manuel Moreno,[2] Gilles Sassatelli,[3] and Peter Zipf[4]

[1] *Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany*
[2] *Technical University of Catalonia (UPC), 08028 Barcelona, Spain*
[3] *Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), University of Montpellier, 34090 Montpellier, France*
[4] *Universität Kassel, 34109 Kassel, Germany*

Correspondence should be addressed to Michael Hübner, michael.huebner@kit.edu

Received 23 December 2009; Accepted 23 December 2009

The fourth edition of the Reconfigurable Communication-centric Systems-on-Chip workshop (ReCoSoC 2008) was held in Barcelona, Spain, from July 9 to 11, 2010.

ReCoSoC is intended to be a periodic annual meeting to expose and discuss gathered expertise as well as state-of-the-art research around SoC-related topics through plenary invited papers and posters. In this event the years before, ReCoSoC had several keynotes given by internationally renowned speakers as well as special events like tutorials. ReCoSoC is a 3-day long event which endeavours to encourage scientific exchanges and collaborations. ReCoSoC aims to provide a prospective view of tomorrow's challenges in the multibillion transistor era, taking into account the emerging techniques and architectures exploring the synergy between flexible on-chip communication and system reconfigurability.

This special issue covers actual and future trends on reconfigurable computing given by academic and industrial specialists from all over the world. The papers presented in this special issue were selected from all ReCoSoC 2008 submissions and were peer reviewed for the final publication in this journal.

The topics of the special issue cover all levels of abstraction in the field of applications, tools, and design methodology. The paper by Imran Quadri et al. "High level modelling of dynamic and reconfigurable FPGAs" shows one approach to hide the complexity of the novel paradigm of reconfigurable computing from the developer. Also "vMAGIC—automatic code generation for VHDL" by Christopher Pohl et al. Underlines the fact that novel design tools are urgently required to handle the huge design space provided by reconfigurable hardware architectures. Additionally to this, the papers by Thilo Piontek, Diana Göhringer, Damien Picard, and Kurt Franz Ackermann contribute to the very important research topic of novel tools, new design methodologies, and new paradigms. A new taxonomy is introduced by Göhringer and the coauthors with the paper "A taxonomy of reconfigurable single-/multiprocessor systems-on-chip." It shows the complexity of the new degrees of freedom in terms of run-time adaptivity and presents a solution to classify the different approaches provided by academics and industry.

The novel trends in self organizing and bioinspired hardware are described in the paper by Jim Harkin et al. where a reconfigurable and biologically inspired paradigm for computation is presented. Christophe Bobda and the coauthors show the exploitation of self organization in networked systems in his paper.

All techniques for reconfigurable hardware need to be implemented on real physical hardware. The papers which present low-level methods for reconfigurable hardware by Christian Schuck, Stephane Chevobbe, Zied Marrakchi, and their coauthors describe techniques and methods which are used to manipulate reconfigurable hardware on signal level while design and run-time.

Last but not least, system-on-chip architectures and algorithms and certainly multiprocessor systems became a very important topic. Topics like efficient task mapping and message passing are described in the paper by Peter Zipf et al. and also in the paper by Gabriel Marchesan. The paper

"A system on a programmable chip architecture for data-dependent superimposed training channel estimation" by Fernando Martín del Campo describes a novel approach for channel estimation in wireless communication systems using reconfigurable hardware.

Michael Hübner
J. Manuel Moreno
Gilles Sassatelli
Peter Zipf

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 408605, 15 pages
doi:10.1155/2009/408605

Research Article

High level modeling of Dynamic Reconfigurable FPGAs

Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser

INRIA Lille Nord Europe, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Centre national de la recherche scientifique (CNRS), University of Lille, 59650 Lille, France

Correspondence should be addressed to Imran Rafiq Quadri, imran.quadri@lifl.fr

Received 31 December 2008; Accepted 26 March 2009

Recommended by J. Manuel Moreno

As System-on-Chip (SoC) based embedded systems have become a defacto industry standard, their overall design complexity has increased exponentially in recent years, necessitating the introduction of new seamless methodologies and tools to handle the SoC codesign aspects. This paper presents a novel SoC co-design methodology based on Model Driven Engineering and the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) standard, permitting us to raise the abstraction levels and allows to model fine grain reconfigurable architectures such as FPGAs. Extensions of this methodology have enabled us to integrate new features such as Partial Dynamic Reconfiguration supported by Modern FPGAs. The overall objective is to carry out system modeling at a high abstraction level expressed in a graphical language like Unified Modeling Language (UML) and afterwards transformation of these models automatically generate the necessary code for FPGA synthesis.

1. Introduction

Since the early 2000s, System-on-Chips (SoCs) have emerged as a new methodology for designing embedded systems in order to target data parallel intensive processing (DIP) applications. While rapid evolution in SoC technology permits to increase computation power, by doubling the number of integrated transistors on chip approximately every two years, the targeted application domains such as multimedia video codecs, software-defined radio, and radar/sonar detection systems are becoming more sophisticated and resource consuming. However the gap between hardware and software evolution is rapidly increasing due to issues such as reduction of product life cycles, increase in design time, and budget limitations. System reliability and verification are also the main hurdles facing the SoC industry and are directly affected by the design complexity. An important challenge is to find efficient design methodologies which raise the design abstraction levels to reduce overall complexity while effectively handling issues such as accurate expression of system parallelism.

For SoC conception, currently High-Level Synthesis (HLS) approaches are utilized: the behavioral description of the system is refined into an accurate register-transfer level (RTL) design for SoC implementation. An effective HLS flow must be *adaptable* to cope with the rapid hardware/software evolution and *maintainable* by the tool designers. The underlying low-level implementation details are hidden from users and their automatic generation reduces time to market and fabrication costs as compared to hand written Hardware Description Languages (HDL) based implementations. However in reality, the abstraction level of the user-side tools is usually not elevated enough to be totally independent from low-level implementations. Each particular implementation of the system (application/architecture) requires a particular specification which is usually in SystemC [1] or a similar language resulting in several disadvantages. Immediate recognition of system information such as related to hierarchy, data parallelism, and dependencies is not possible; differentiation between different concepts is a daunting task in a textual description and makes modifications complex and time consuming.

Model Driven Engineering [2] (MDE) is an emerging domain and can be seen as a *High- Level Design Flow* for SoC and an effective solution for resolving the above mentioned issues. The advantage of MDE is that the

complete system (application and architecture) is modeled at a high specification level allowing several abstraction stages, thus a system can be viewed globally or from a specific point of view of the system allowing to separate the system model into parts according to relations between system concepts defined at different abstraction stages. This *Separation of Views* (SoV) allows a designer to focus on a domain aspect related to an abstraction stage thus permitting a transition from solution space to problem space. MDE's Unified Modeling Language (UML) graphical nature increases system comprehensibility and allows users to provide high abstraction level descriptions of systems in order to easily identify the internal concepts (task/data parallelism, data dependencies, and hierarchy). The graphical nature of these specifications allows for their reuse, modification, maintenance, and extension.

Partial Dynamic Reconfiguration [3] (PDR) is an emerging feature supported by modern FPGAs allowing specific regions of an FPGA to be reconfigured on the fly, hence introducing the notion of *virtual hardware* with the advantage of time-sharing the available hardware resources for executing multiple tasks. PDR allows task swapping depending upon application needs, hardware limitations, and Quality-of-Service (QoS) requirements (power consumption, performance, execution time, etc.). Currently only Xilinx FPGAs fully support this feature.

Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [4] is an industry standard proposal of the Object Management Group (OMG) for model-driven development of embedded systems. It adds capabilities to UML allowing to model software, hardware, and their relations, along with added extensions (e.g., performance and scheduling analysis). This standard although while rich in concepts, unfortunately lacks tools to move to execution platforms and is insufficient for FPGA modeling.

GASPARD [5, 6] is a MARTE compliant SoC co-design framework dedicated specially towards parallel hardware and software and allows to move from high-level MARTE specifications to an executable platform. It exploits the parallelism included in repetitive constructions of hardware elements or regular constructions such as application loops.

The main contribution of this paper is to present part of a novel design flow using an extended version of MARTE for general modeling of FPGAs. Our methodology allows us to introduce PDR in MARTE for modeling all types of FPGAs supporting our chosen PDR flow. Finally using the MDE model transformations, the design flow can be used to bridge the gap between high-level specifications and low implementation details to automatically generate the code required for the creation of bitstream(s) for FPGA implementation.

The rest of this paper is organized as follows. An overview of MDE is provided in Section 2 while Section 3 summarizes our MARTE compliant GASPARD framework. Section 4 describes PDR while Section 5 gives a summary of related works. Section 6 illustrates our methodology related to implementing PDR supported FPGAs. This paper finishes with a case study in Section 7 followed by a conclusion.

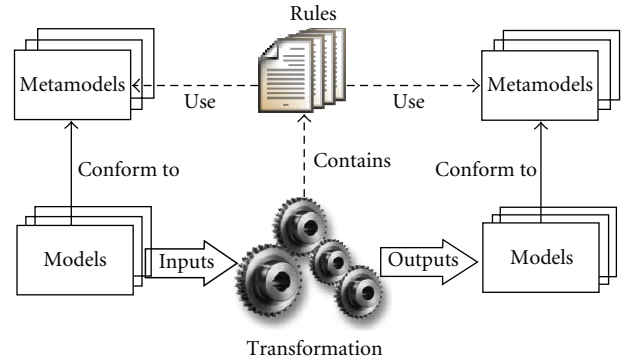


Figure 1: An overview of model transformations.

2. Model Driven Engineering

MDE is centered around three focal concepts. *Models*, *Metamodels,* and *Transformations*. A model is an abstract representation of some reality and has two core elements: *concepts* and *relations*. Concepts represent "things" and relations are the "links" between these things in reality. A model can be observed from different abstract point of views (views in MDE). A metamodel is a collection of concepts and relations for describing a model using a model description language and defines syntax of a model. This relation is analogous to a text and its language grammar. Each model is said to *conform* to its metamodel at a higher definition level.

Models in MDE are not only used for communication and comprehension but using model transformations [7], produce concrete results such as a source code. A model transformation as shown in Figure 1 is a compilation process that transforms a *source* model into a *target* model and allows to move from an abstract model to a more detailed model. The source and target models each conform to their respective metamodels thus respecting *exogenous* transformations. A model transformation is based on a set of *rules* (either declarative or imperative) that help to identify concepts in a source metamodel in order to create enriched concepts in the target metamodel. This separation allows to easily extend and maintain the compilation process. New rules extend the compilation process and each rule can be independently modified. Model transformations carry out refinements moving from high abstraction levels to low levels for code generation. At each intermediate level, implementation details are added to the compilation process. The advantage of this approach is that it allows to define several model transformations from the same abstraction level but targeted to different lower levels, offering opportunities to generate several implementations from a specification. The model transformations can be either unidirectional (modification of source model only: targeted model generated automatically) or bidirectional (target model is also modifiable) in nature. In the second case, this could lead to a model synchronization issue [8]. OMG has proposed the Meta Object Facility (MOF) Query/View/Transformation (QVT) [9] standard for model query and transformations.

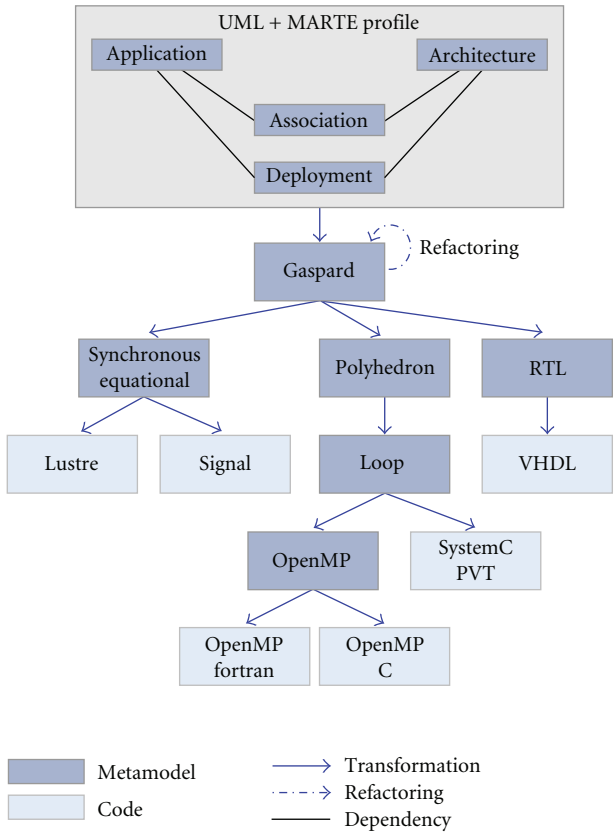


Figure 2: The existing GASPARD framework with deployment added at the MARTE modeling abstraction level.

3. GASPARD Co-Design Framework

GASPARD [5, 6] is a MDE oriented SoC co-design framework and a subset of the MARTE standard currently supported by the SoC industry. In GASPARD as in MARTE, a clear *separation of concerns* exists between the hardware/software models as shown in Figure 2. GASPARD integrates the MARTE allocation mechanism (*Alloc* package) that permits to link the independent hardware and software models (e.g., mapping of a task or data onto a processor or a memory, resp.). The concept used to specify an allocation is called an *Allocate*. An allocation can represent either a *spatial* or a *temporal* placement. Up till now GASPARD only supported spatial placement but we have also integrated the temporal placement allocation in order to implement systems supporting PDR.

GASPARD has contributed in MARTE conception with the *Repetitive Structure Modeling (RSM)* package. RSM is based on a Model of Computation (MoC) known as ArrayOL [10] which describes the potential parallelism in a system and is dedicated to intensive multidimensional signal processing (ISP). RSM allows to describe the regularity of a system's structure (composed of repetitions of structural components interconnected in a regular connection pattern) and topology in a compact manner. GASPARD uses the RSM semantics to model large regular hardware architectures (such as multiprocessor architectures) and parallel applications. GASPARD currently targets *control and data flow oriented* ISP applications (such as multimedia video codes, high-performance applications, anticollision radar detection applications). The applications targeted in GASPARD are widely encountered in SoC domain and respect ArrayOL semantics [10]. Although MARTE is suitable for modeling purposes, it lacks the means to move from modeling specifications to execution platforms. GASPARD bridges this gap and introduces additional concepts and semantics to fill this requirement for SoC co-design.

The first addition relates to the semantics of modeled applications. In MARTE, nearly all kinds of embedded applications can be specified but their behavior cannot be entirely defined. It is up to the designer/programmer to determine the precise behavior. As GASPARD deals with ISP applications based on a specific MoC, we only use the UML concept of *Component* (in order to define an application component) and MARTE *FlowPort* type (to define all port types in both the application and the architecture).

GASPARD also benefits from the notion of a *Deployment* model level [11] which is related to the specification of elementary components (basic building blocks of all other components). To transform the high abstraction level models to concrete code, detailed information must be provided. The Deployment level links every elementary component to an existing code for both the hardware and the application hence facilitating Intellectual Property (IP) reuse. Each elementary component can have several implementations, for example, an application functionality can either be optimized for a processor (written in C/C++) or written in hardware (HDL) for implementation as an hardware accelerator. Hence this level is able to differentiate between the hardware and software functionalities independent from the compilation target. It provides IP information for model transformations to form a compilation chain to transform the high abstraction level models (application, architecture, and allocation) for different domains (formal verification, simulation, high-performance computing, or synthesis). This concept is currently not present in MARTE and is a potential extension of the standard to allow a complete flow from model conception to automatic code generation. It should be noted that the different transformation chains (simulation, synthesis, verification, etc.) are currently unidirectional in nature.

Once GASPARD models are specified in a graphical environment, MOdel to MOdel Transformation Engine (MOMOTE) tool which has been developed internally in the team and is based on EMFT QUERY [12], takes these models as input. MOMOTE is a Java framework that allows to perform model to model transformations. It is composed of an API and an engine. It takes source models as input and produces target models with each conforming to some metamodel.

MOdels to CODe Engine (MOCODE) is another GASPARD integrated tool for automatic code generation which is based on EMF Java Emitter Templates (JET) [13]. JET is a generic template engine for code generation purposes. The JET templates are specified by using a JavaServer Pages (JSP)

like syntax and are used to generate Java implementation classes. Finally these classes can be invoked to generate user customized source code, such as Structured Query Language (SQL), eXtensible Markup Language (XML), Java source code or any other user specified syntax. MOCODE offers an API that reads input models, and also an engine that recursively takes elements from input models and executes a corresponding JET Java implementation class on them.

We are also in process of modifying the deployment level into a *controlled deployment* model to integrate the control aspect of PDR which is an offshoot of the works being done in the synchronous domain in the GASPARD framework [14]. This model will allow to link an elementary component with several IPs (allowing several possible final implementations) as compared to the current approach where an elementary component is only linked finally with one IP among several. This has allowed the concept of *configurations*: an elementary component can have different implementations in different configurations respecting the semantics of partial bitstreams. The control aspect in the deployment level allows to convert the semantics of the new deployment level into a control-mode automata-based component approach and afterwards via model transformations, convert this control aspect into the state machine code to be implemented in the reconfigurable controller in the FPGA automatizing part of the reconfiguration management. However, this aspect is out of scope of this paper as here we only focus on the modeling approach.

4. Basic PDR Related Concepts

Currently PDR is only supported by Xilinx FPGAs. Xilinx initially proposed two methodologies (difference-based and module-based) [15, 16] followed by the *Early Access Partial Reconfiguration (EAPR)* [17] flow. The EAPR flow allows static nets to cross the reconfigurable region boundaries and supports 2D reconfigurable module shapes, thus resolving the drawbacks present in the earlier modular design methodology. The idea is that part(s) of the FPGA remains static, while another part(s) is dynamically reconfigurable at run-time. *Bus macros* BMs are used to ensure proper routing between the static and dynamic parts during and after reconfiguration. The *Internal Reconfiguration Access Port (ICAP)* [18] is an integral component that permits to read/write the FPGA configuration memory at run-time. The ICAP is present in nearly all Xilinx FPGAs ranging from the low-cost Spartan-3A(N) to the high-performance Virtex-5 FPGAs [19]. For Virtex-II and Virtex-II Pro series, the ICAP furnishes 8-bit input/output data buses while with the Virtex-4 Series, the ICAP interface has been updated with 32-bit input/output data buses to increase its bandwidth. In combination with the ICAP, a *Reconfiguration controller* (either a PowerPC or a Microblaze) can be implemented inside the FPGA in order to build a self controlling dynamically reconfigurable system [18].

Virtex devices also support the feature of *glitchless dynamic reconfiguration*, if a configuration bit holds the same value before and after reconfiguration, the resource controlled by that bit does not experience any discontinuity in operation, with the exception of LUTRAMs and SRL16 primitives [3]. This limitation was removed in the Virtex-4 family. With the introduction of EAPR flow tools, this problem has also been resolved for Virtex-II/Pro FPGAs.

5. Related Works

ROSES [20] is an environment for Multiprocessor SoC (MPSoC) design and specification however it does not conform to MDE concepts and as compared to our framework, starts from a low-level description equivalent to our deployment level. Reference [21] provides a simulink-based graphical HW/SW co-design approach for MPSoC but the MDE concepts are absent. In contrast, reference [22] uses the MDE approach for the design of a Software-Defined Radio (SDR), but they do not utilize the MARTE standard as proposed by OMG and utilize only pure UML specifications. While works such as [23, 24] are focused on generating VHDL from UML state machines, they fail to integrate the MDE concepts for HW/SW co-design and are not capable of managing complex ISP applications. MILAN [25] is another project for SoC co-design benefiting from the MDE concepts but is not compliant with MARTE. Only the approach defined in [26, 27] comes close to our intended methodology by using the MDE concepts and the MARTE standard for SoC co-design. Yet the disadvantage is that in reality it only generates the ISP application part to be implemented as a hardware accelerator in an FPGA. Hence there is no hardware description of FPGA at the high design level. MOPCOM [28] uses MDE and MARTE but is not oriented towards PDR. In [29], the authors present a design flow to manage partially reconfigurable regions of an FPGA automatically using SynDEx. A complete system (application/architecture) can be modeled and implemented, however the MDE concepts are strikingly absent. Similarly [30] present an HLS flow for PDR, yet it still starts from a lower abstraction level as compared to MDE.

In the domain of runtime reconfiguration, Xilinx initially proposed two design flows in [15, 16] termed as the *Modular-based* and *Difference-based* approaches. The difference-based approach is suitable for small changes in a bitstream but is inappropriate for a large dynamically reconfigurable module necessitating the use of the modular approach. However, both approaches were not very effective leading to new alternatives.

Sedcole et al. [31] presented a modular approach that was more effective than the initial Xilinx methodologies and were able to carry out 2D reconfiguration by placing hardware cores above each other. The layout (size and placement) of these cores was predetermined. They made use of reserved static routing in the reconfigurable modules which allowed the signals from the base region to pass through the reconfigurable modules allowing communication between modules by using the principle of glitchless dynamic reconfiguration.

Becker et al. [32] implemented 1D modular reconfiguration using a horizontal slice-based BM. All the reconfigurable modules that stretched vertically to the height of the device

were connected with the BM for communication. They followed by providing 2D placement of modules of any rectangular size by using routing primitives that stretch vertically throughout the device [33]. A module could be attached to the primitive at any location, hence providing arbitrary placement of modules. The routing primitives are LUT-based and need to be reconfigured at the region where they connect to the modules. A drawback of this approach is that the number of signals passing through the primitives are limited due to the utilization of LUTs. This approach has been further refined in [34].

In March of 2006, Xilinx introduced the *Early Access Partial Reconfiguration (EAPR)* [17] design flow along with the introduction of CLB-based BMs which are pre-routed IP cores. The concepts introduced in [31, 32] were integrated in this flow. The restriction of full column modular PDR was removed allowing reconfigurable modules of any arbitrary rectangular size to be created. The EAPR flow also allows signals from the static region(s) to cross through the partially reconfigurable region(s) without the use of BMs. Using the principle of glitchless reconfiguration, no glitches will occur in signal routes as long as they are implemented identically in every reconfigurable module for a region. The only limitation of this approach is that all the partial bitstreams for a module to be executed on a reconfigurable region must be predetermined hence making it *semipartial dynamic* in nature.

Works such as [19, 35] focus on implementing softcore internal configuration ports on Xilinx FPGAs such as the pure Spartan-3 which do not have the hardware ICAP core rendering dynamic reconfiguration impossible via traditional means. In [35] a soft ICAP known as JCAP (based on the serial JTAG interface) is introduced for realizing PDR while [19] introduces the notion of a PCAP (based on the parallel SelectMAP interface) providing improved reconfiguration rates as compared to the JTAG approach. However this approach is only suitable to reconfigure very small regions of FPGA and since the design is not an embedded one, it is impossible to retrieve bitstreams from an external memory. This issue has been addressed in [36], where a complete reconfigurable embedded design on a Spartan-3 board has been implemented using a reconfigurable coprocessor. The user application can map to a number of potential coprocessors and the reconfiguration controller can order the self-reconfiguration of the system for the reconfigurable coprocessor resulting in loading of the partial bitstream related to a potential coprocessor. The results show that this achieves a compromise between the works presented in [19, 35].

In [37], a new framework is introduced for implementing PDR by the utilization of a PLB ICAP. The ICAP is connected to the PLB bus as a master peripheral with direct memory access (DMA) to a connected BRAM (as compared to the traditional OPB-based approach). This provides an increased throughput of about 20 percent by lowering the process load. Reference [38] provides another flavor of a PDR architecture by attaching a Reconfigurable Hardware accelerator to a Microblaze Reconfiguration controller via a Fast Simplex Link (FSL) [39]. Works such as [40] use ICAP to connect

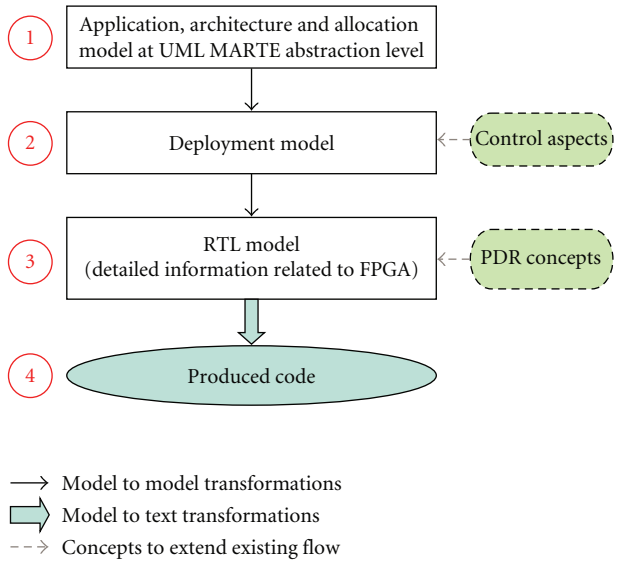


Figure 3: The complete design flow.

with Network on chip (NoC) to allow distributed access to speed up reconfiguration time. However the Read-modify-write (RMW) [18] mechanism is not supported which is an important factor to speed up reconfiguration times. This limitation has been resolved in [41] where an ICAP communicates with an NoC using a light-weight RMW method in order to reduce reconfiguration time.

For our implementation purposes, we have focused mainly on the Xilinx EAPR flow methodology [3] as it is openly available and can be adapted to other PDR architecture implementations. Our contribution does not relate to creating a new PDR architecture methodology per se at the RTL level, but is based on how the methodology can be raised to a higher abstraction level for (a) reducing design complexity, and (b) to create a generic PDR approach for implementing all ISP applications supporting our MoC. This approach can then be taken as an input for the designers who contribute to the PDR domain at the RTL level. While there are lots of related tools, works and projects; we have only detailed some and have not given an exhaustive summary. To the best of our knowledge, only our methodology takes into account the following domain spaces: SoC HW/SW co-design, ISP applications, MDE, MARTE standard, and PDR which is the novelty of our design flow.

6. Modeling of Partially Dynamically Reconfigurable FPGAs

We first present our design flow to model and implement PDR supported fine grain reconfigurable architectures (FPGAs) as shown in Figure 3 which is an extension of the design flow present in [27]. As described before, this paper only focuses on the first layer of our design flow (application, architecture, and allocation modeling) which is the most abstract in nature. The 2nd layer deals with the Deployment layer with integrated control aspects for determining the

configuration aspects for static/partial bitstreams. This layer serves as an input to the PDR-RTL layer where detailed transformation rules related to targeted application and FPGA in general (clock/reset signals, interface creation, constraint file among others) are present. This layer uses the control aspects in layer 2 for generating part of the reconfiguration controller and is responsible for partial FPGA layout for accelerator placement. Each part of these model levels/layers correspond to its respective metamodel. Finally using MOCODE, it is possible to convert the models to source code. Once the source code for the application (implemented as a hardware accelerator) and the reconfigurable controller is obtained, usual synthesis flow can be invoked using commercial tools such as Xilinx ISE [42] for final implementation. Our aim is not to replace the commercial tools but to aid them in the conception of a system. While tools like PlanAhead [43] are capable of estimating the FPGA resources required for a reconfigurable module, it is finally up to the user to decide the best placement depending on QoS requirements. Also as our work deals with dynamic partially reconfigurable FPGAs and currently only Xilinx FPGAs support this feature, our modeling methodology revolves around the Xilinx reconfiguration flow as it is openly available and flexible enough to be modified. While this does make the architectural aspects of our design flow restricted to Xilinx-based technologies, it is an implementation choice as currently no other FPGA vendor supports this feature. It should be noted that our methodology can be used as a building block to support other non standard PDR implementations based on Xilinx FPGAs (use of Soft ICAP cores, e.g.).

6.1. MARTE Hardware Concepts Overview. The hardware concepts in MARTE are grouped in the *Hardware Resource Model (HRM)* package. HRM consists of several views, a functional view (*HwLogical* subpackage), a physical view (*HwPhysical* sub-package) or a merge of the two. The two sub-packages derive certain concepts from the *HwGeneral* root package in which *HwResource* is a core concept that defines a generic hardware entity. An HwResource can be composed of other HwResource(s) (e.g., a processor containing an ALU). This concept is then further expanded according to the functional or physical specifications. The functional view of HRM defines hardware resources as either *computing*, *storage*, *communication*, *timing,* or *device* resources. The physical view represents hardware resources as physical components with details about their shape, size, and power consumption among other attributes. GASPARD currently only supports the functional view, but we have also integrated the physical and merged views for modeling PDR featured architectures. The HRM also exploits the Nonfunctional Properties (NFP) MARTE package that introduces a value specification language (VSL) which supports complex expressions for specifying nonfunctional properties and quantitative annotations with measurement units. The NFP package provides a rich library of basic types like *Data size*, *Data Transmission Rate,* and *Duration*.

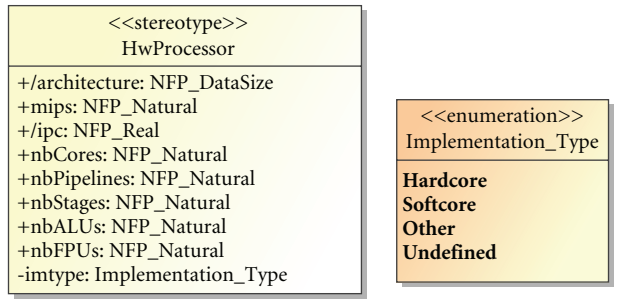


Figure 4: Modified version of the HwProcessor concept.

6.2. MARTE Modifications for PDR Concepts. In order to model PDR supported FPGAs, the HRM package was examined and we found it to be lacking in certain aspects. The *HwComputing* sub-package in the HRM functional view defines a set of active processing resources pivotal for an execution platform. An *HwComputingResource* symbolizes an active processing resource that can be specialized as either a processor (*HwProcessor*), an ASIC (*HwASIC*), or a PLD (*HwPLD*). An FPGA is represented by the HwPLD stereotype, it can contain a RAM memory (*HwRAM*) (as well as other HwResources) and is characterized by a technology (SRAM, Antifuse, etc.). The cell organization of the FPGA is characterized by the number of rows and columns, but also by the type of architecture (Symmetrical array, row-based, etc.). These concepts are partly sufficient enough for high-level abstract FPGA description but do not integrate all aspects (such as interfaces for IP cores, processor implementation type, etc.) and need a detailed modeling for representing a complete real heterogeneous FPGA. Also the concepts related to representing a processor are not sufficient for a complex SoC on FPGA design in which a processor can either be implemented as a softcore IP or integrated as a hardcore IP. We thus add the attribute **imtype** (Implementation_Type) that is flexible enough to define a processor implementation as either **Hardcore** or **Softcore** and adaptable using the **Other** and **Undefined** types. The last two types have been added for extension purposes. The **Other** type is denoted for other existing technologies which are not actually specified at the time of modeling (in the case of processor implementation, this type is set to false) and **Undefined** for future evolution in hardware and to allow easy modification of existing models. They can be viewed as having equivalent purposes but are created to avoid ambiguity. Figure 4 shows only the simplified modeling description of the modified HwComputing sub-package related to a processor implementation.

The second modification relates to the physical *HwLayout* sub-package as shown in Figure 5. The core concept of this package is *HwComponent* which is an abstraction of any real hardware entity based on its physical attributes. HwComponent can be specialized as either *HwChip* (e.g., a processor), *HwChannel* (e.g., a bus), *HwPort* (e.g., an interface), *HwCard* (e.g., a motherboard), or an *HwUnit* (a hardware resource that does not fall into the preceding four categories). As a PDR featured architecture consists of

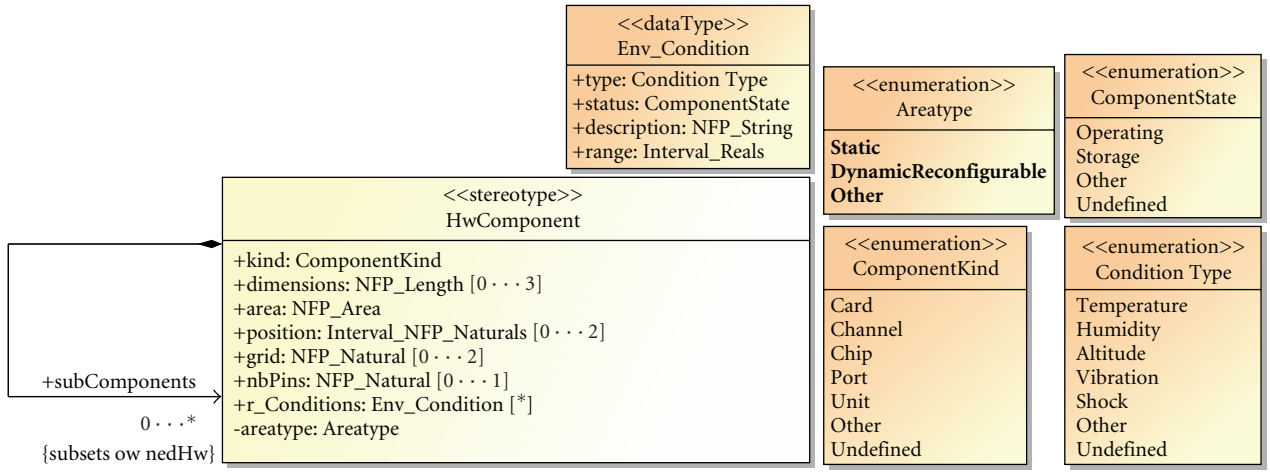


Figure 5: Modified version of the HwComponent concept.

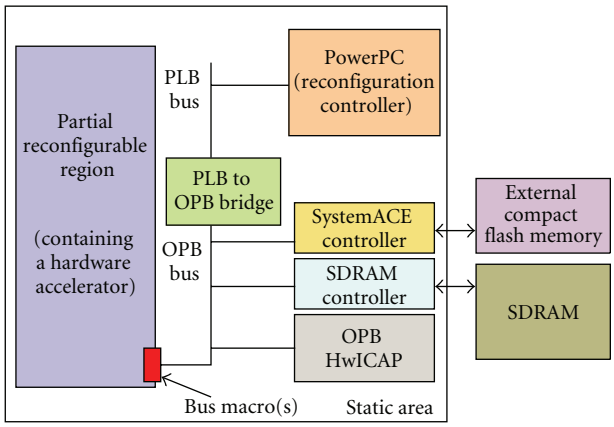


Figure 6: Block Diagram of the architecture of our reconfigurable system.

either static or dynamically reconfigurable region(s), we have introduced the attribute **areatype** (Areatype) which can be either **Static**, **DynamicReconfigurable,** or typed as **Other** for extension purposes. This concept has been introduced in the MARTE physical concepts as the area properties for a hardware component are usually expressed in the physical sub-package of the HRM. Figure 5 thus shows only the simplified overview of our modified HwComponent concept.

These are the 2 added extensions of the MARTE standard. These concepts are specifically added to the high level in order to generally benefit other frameworks and system descriptions and they could be easily extended. While these modifications seem trivial in nature, they make a definite impact in the corresponding model to model transformations for the final implementation. We now present the specific concepts related to FPGA and PDR in our methodology.

In Figure 6 we present an example of a PDR-supported Xilinx FPGA that we have implemented in reality. We have used the Virtex-II Pro XC2VP30 on an XUP Board [44] as a reference as it seems to be a popular choice for implementing PDR. We have implemented a Reconfiguration Controller (a PowerPC in this case) connected to the high-speed 64-bit PLB bus and links with the slower slave peripherals (connected to the 32-bit OPB bus) via a PLB to OPB Bridge. The buses and the bridge are a part of the IBM Coreconnect technology [45]. The OPB bus is attached to some peripherals such as A SystemACE controller (for accessing the partial bitstreams placed in an external onboard Compact Flash (CF) card). An SDRAM controller for a DDR SDRAM present onboard (permits the partial bitstreams to be preloaded from the CF during initialization for decreasing the reconfiguration time). An ICAP is present in the form of an OPB peripheral (OPBHwICAP) and carries out partial reconfiguration using the read-modify-write (RMW) mechanism. The static (base) portion of the FPGA is connected to a Reconfigurable Hardware Accelerator (RHA) via BMs. Although the RHA can be placed with the fast PLB bus, it is an implementation choice to connect it with the OPB bus to make the system more diverse at the cost of reconfiguration time. The concepts such as PowerPC, PLB, and OPB buses, PLB to OPB Bridge, CF and SDRAM memories can be defined using the current MARTE HRM concepts. However the peripherals, BMs, ICAP, and RHA require an extended and more detailed conception. An internal memory can also be used to store the partial bitstreams depending upon the application size. Since our targeted applications cannot be placed inside the internal memory, we have used an external memory.

The *HwCommunication* sub-package in the HRM functional view represents the concepts for all hardware communications. *HwMedia* is the central concept that defines a communication resource capable of data transfer with a theoretical bandwidth. It can be controlled by *HwArbiter*(s) and connected to other HwMedia(s) by means of an *HwBridge*. An *HwEndpoint* defines a connection point of an HwResource and can be defined as an interface (e.g., pin or port). *HwBus* illustrates a specific wired channel with

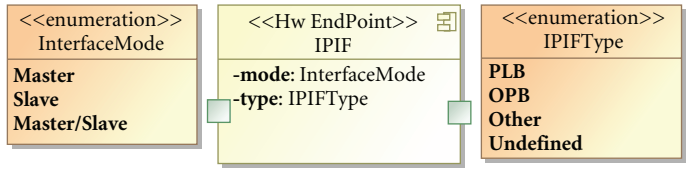


Figure 7: Modeling of the IPIF hardware wrapper.

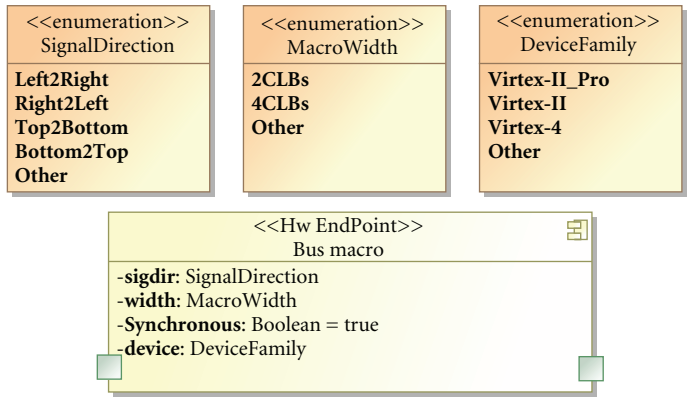


Figure 8: Modeling of a BM.

particular functional attributes. These concepts are sufficient and abstract enough to define all kinds of communication resources. Some of the other common HRM concepts that we utilize for PDR are *HwComputingResource* (to describe a general computing resource) from the *HwComputing* package, *HwRAM* and *HwROM* from the *HwMemory* package (for RAM and ROM concepts), *HwStorageManager* from the *HwStorageManager* package (for a memory controller), *HwClock* from the *HwTiming* package (to specify a clock), and *HwIO* from the *HwIO* package (for an I/O resource).

Xilinx provides the notion of an Intellectual Property Interface (IPIF) module which acts as a hardware bus wrapper specially designed to ease IP core interfacing with the IBM Coreconnect buses using IPIC connections. It can also be used for other purposes such as connecting the OPB bus to a DCR bus [45] (another bus of the Coreconnect technology). As all peripherals in our architecture consist of the IPIF module and an IP core, this is a vital modeling concept and has permitted us to model all peripherals which are themselves hierarchically composed. The abstract IPIF module has two basic attributes: a **mode** which can be either **Master**, **Slave,** or **Master/Slave**, and **type** that determines the protocol of IPIF adapted for a particular bus. It can be either **PLB**, **OPB,** or extensible using **Other** or **Undefined** types. We avoided adding detailed information related to the options and protocols offered by IPIF (software registers, FIFOs, etc.) to simplify its definition at the high abstraction level. The IPIF is typed as *HwEndpoint* to illustrate that it is a hardware wrapper module providing an interface to the actual IP core. This approach can be adapted to model customized wrappers for customized user IPs. Figure 7 shows the IPIF design.

The second modeling concept is of BMs. Although the EAPR flow allows static nets in the base design to pass through the reconfigurable region(s) without the use of BMs, they are still essential in order to ensure the correct communication routing between the static and dynamic regions. Being CLB-based in nature, they provide a unidirectional 8-bit data transfer. BMs have been modeled having four attributes. The **sigdir** attribute determines the communication direction that can be **Left2Right,** or **Right2Left** (for Virtex-II and Virtex-II Pro devices), as well as **Top2Bottom**, **Bottom2Top** or **Other** for Virtex-IV and other future PDR supported devices. The **width** attribute determines the CLB width of the BM (**2CLBs** or **4CLBs** width making it either a narrow or wide BM or use of **Other** for a user specified width). The **Synchronous** attribute determines if the BM is a synchronous one or not. We have assigned a default value of **true** to this attribute (as recommended by Xilinx). The final attribute **device** determines the targeted FPGA device family (either **Virtex-II**, **Virtex-II Pro**, **Virtex-4** or a newer device such as Virtex-5 using the **Other** type). The BM (*Busmacro*) (as shown in Figure 8) is typed as *HwEndpoint* to illustrate that it is a communication medium between the static and dynamically reconfigurable modules of the FPGA.

Modeling of the OPB_HWICAP peripheral is then carried out as shown in Figure 9. It consists of an IPIF (**ic2opb**) connected to the HWICAP core (**hwicap**) (typed as *HwComputingResource*) and is itself defined as an *HwComputingResource*. The HWICAP core is itself composed of three subcomponents: an ICAP controller (**icapctrl**) and ICAP Primitive (**icap**) both typed as *HwComputingResource*(s), and a BlockRAM (**bram**) defined as *HwRAM* for storing a configuration frame of FPGA memory. The BlockRAM contains a port having a multiplicity of 2 indicating that it is repeated two times (dual port RAM). We have used

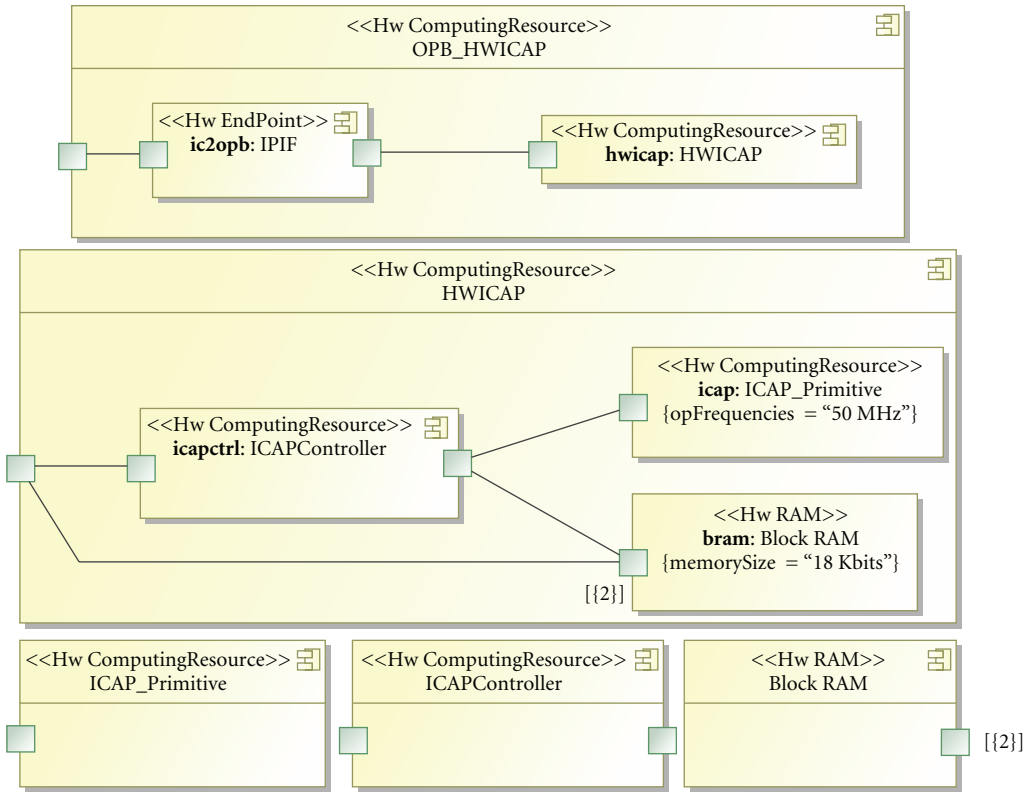


Figure 9: Modeling of the OPB HWICAP peripheral.

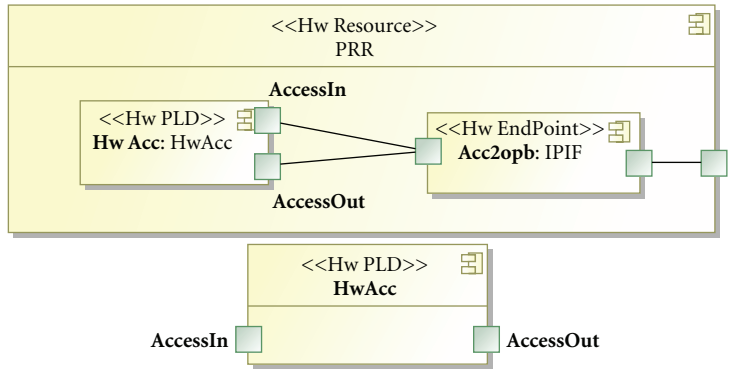


Figure 10: A reconfigurable hardware accelerator.

the notion of a *Reshape* connector [10] (as defined in the MARTE RSM package and in our MoC) in order to link the sub components of the HWICAP. The Reshape allows to represent complex link topologies in a simplified manner. In Figure 9, the Reshape connectors permit to specify accurately which port (either the port of the ICAPController or the single port of the HWICAP itself) is connected to which repetition of the port of the BlockRAM. The sub components of HWICAP also have specific attributes (such as BlockRAM having a 18 Kbit memory) related to actual architectural details of the targeted FPGA. We refer the reader to [18] for a detailed description related to HWICAP.

Figure 10 represents the modeling of the Reconfigurable Hardware Accelerator (RHA). The Partial reconfigurable region (PRR) consists of a RHA (**HwAcc**) defined as *HwPLD* having ports **AccessIn** and **AccessOut** and an IPIF module (**Acc2opb**). The PRR is typed as the generic *HwResource* type in order to illustrate that the partially reconfigurable region can be either generic or have a specific functionality. The RHA is typed as *HwPLD* as it is reconfigurable, as compared

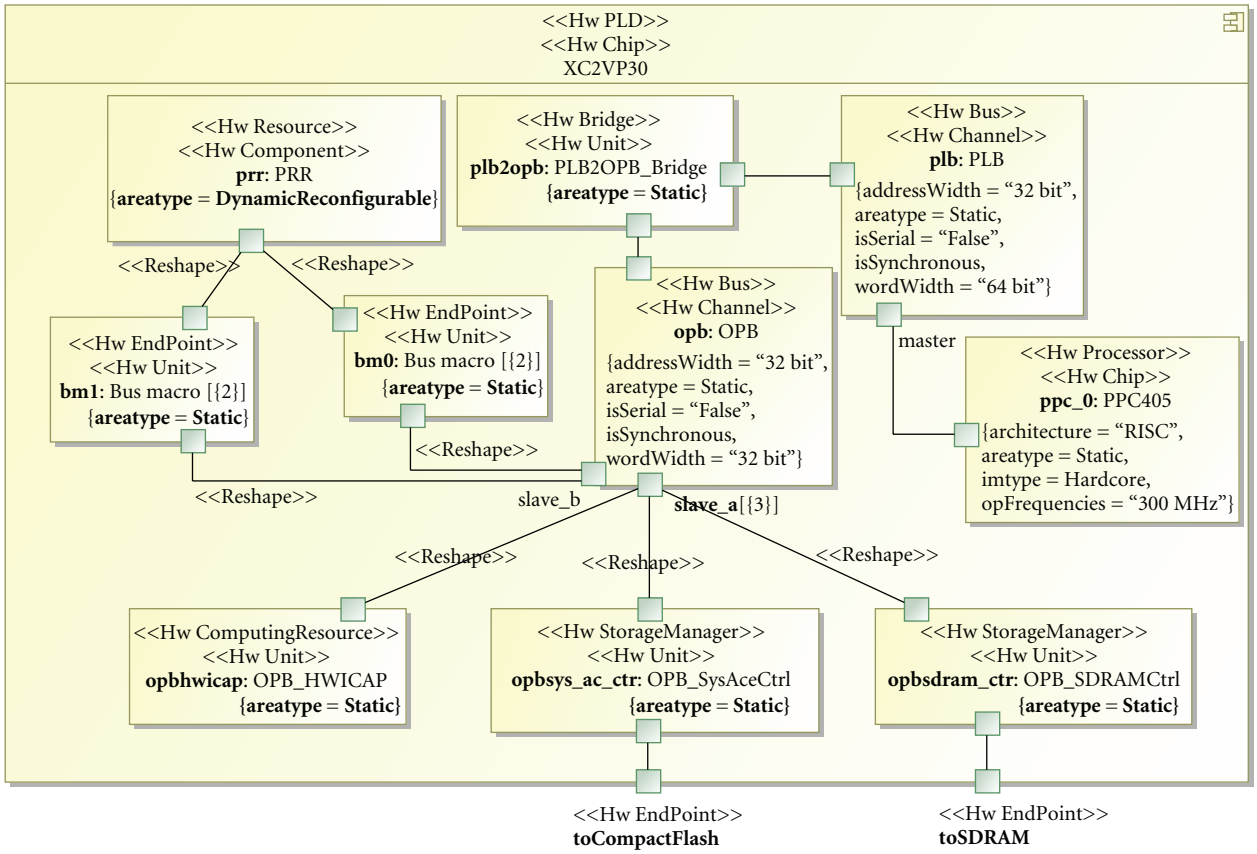


Figure 11: Modeling of our PDR architecture.

to a typical hardware accelerator in a large-scale SoC design which can be seen as a *HwASIC* (after fabrication) depending upon the designer's point of view.

Figure 11 finally shows our reconfigurable architecture (An XC2VP30 Virtex-II Pro chip) using our proposed concepts in a merged functional/physical view to express all the necessary attributes related to the corresponding physical/logical stereotypes. Every hardware component has two type definitions (the first being the functional and the second representing the physical one). The XC2VP30 chip consists of a PowerPC PPC405 (**ppc_0**) connected to the slave peripherals: the OPB_SysAceCtrl (**opbsys_ac_ctr**), the OPB_HWICAP (**opbhwicap**), the OPB_SDRAMCtrl (**opbsdram_ctr**), and the PRR (**prr**) via the PLB (**plb**) and OPB (**opb**) buses. The PLB2OPB_Bridge (**plb2opb**) connects the two buses, while Bus macro(s) (**bm0** and **bm1** having types Left2Right and Right2Left resp.) connect the OPB bus to the PRR. Each of the BMs is instantiated two times (multiplicity of 2 on both **bm0** and **bm1,** resp.). The OPB bus has a **slave_a** port with a multiplicity of 3 to allow the bus to connect to the peripherals (opbhwicap, opbsys_ac_ctr, and opbsdram_ctr). Reshape connectors are used to determine which peripheral is connected to which repetition of the slave port. Similarly Reshape connectors are used to determine the accurate connections between the BMs and the ports of OPB and PRR. Although a single slave port can be used on OPB with an appropriate multiplicity to include the topology of BMs, this is avoided to reduce the design complexity. Finally, the XC2VP30 contains two HwEndPoint(s) interfaces, **toCompactFlash** and **toSDRAM** to connect **opbsys_ac_ctr** and **opbsdram_ctr** to the external Compact Flash and SDRAM memories, respectively. The OPB arbiter is not modeled as it is considered to be a part of the OPB Bus. It should be noted that this is a top level view only and nearly each component is itself hierarchically composed. Note that the new attributes and those by default in the HRM package of MARTE allow the designer to specify general attributes of each component at the highest abstraction level (e.g., ppc_0 having a frequency of 300 MHz).

7. Case Study: A GASPARD Application Mapped on our PDR Architecture

A case study of a complete SoC model is presented here to illustrate our modeling methodology. The modeled application *MainApplication* is an academic grayscale 4×4 pixel image filter application (producing 8-bit images) respecting our MoC. It consists of three tasks (application components): An image sensor PictureGen (**pg**), the main image filter task Flux (**tasks**) (Figure 12), and an output PictureRead (**pr**). The Flux component is comprised of a Filter component (**filter**) (repeating infinitely as shown by the

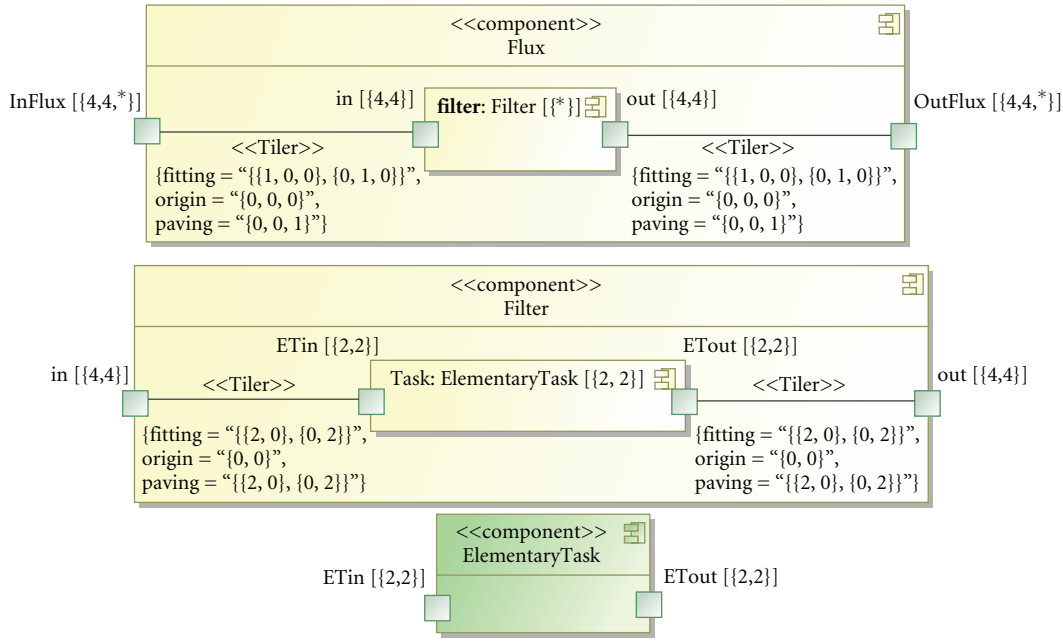


Figure 12: Model of an image filter task.

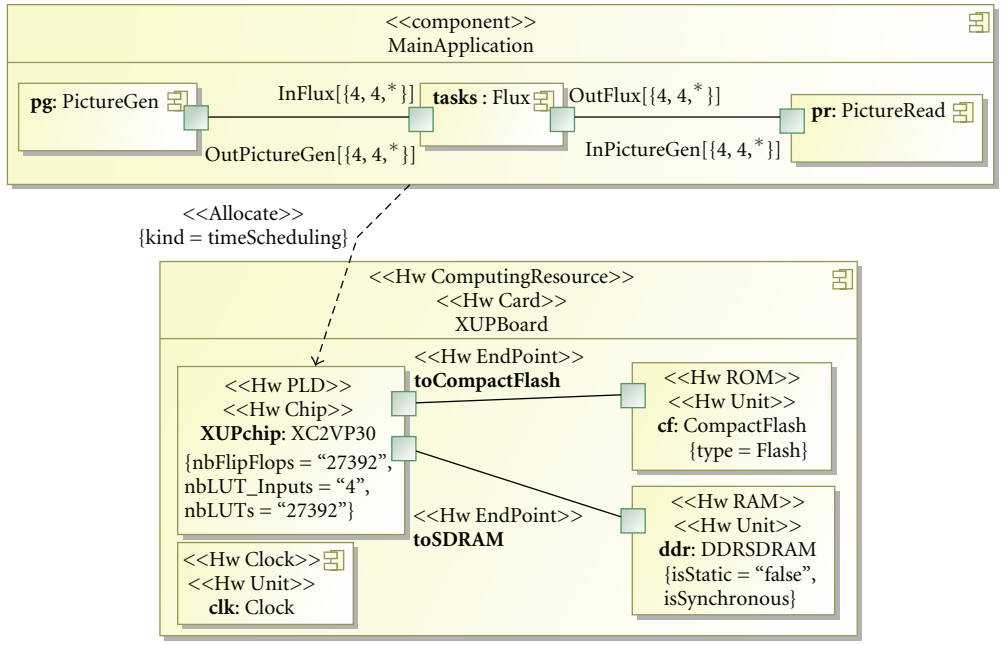


Figure 13: Allocation: Level 1.

multiplicity of $*$). The Filter component itself contains an elementary application component ElementaryTask (**Task**) being repeated four times (having a multiplicity of 2,2). The Tiler connectors are used to describe the tiling of produced and consumed arrays by a pattern mechanism [10]. The elementary component can have several implementations and the controlled deployment layer can create different configurations for the reconfigurable hardware accelerator and this information is thus passed to the PDR-RTL layer.

We then illustrate the different levels of allocation of the application onto the architecture. In Figure 13, the model of the whole application is shown allocated to the XC2VP30 chip (**XUPchip**) on an XUPBoard using the *Allocate* type allocation. Currently GASPARD only supports spacial placement (static scheduling at compilation time due to the nature of targeted applications), however due to the nature of PDR and related applications; we have integrated the temporal placement: *timeScheduling* (dynamic

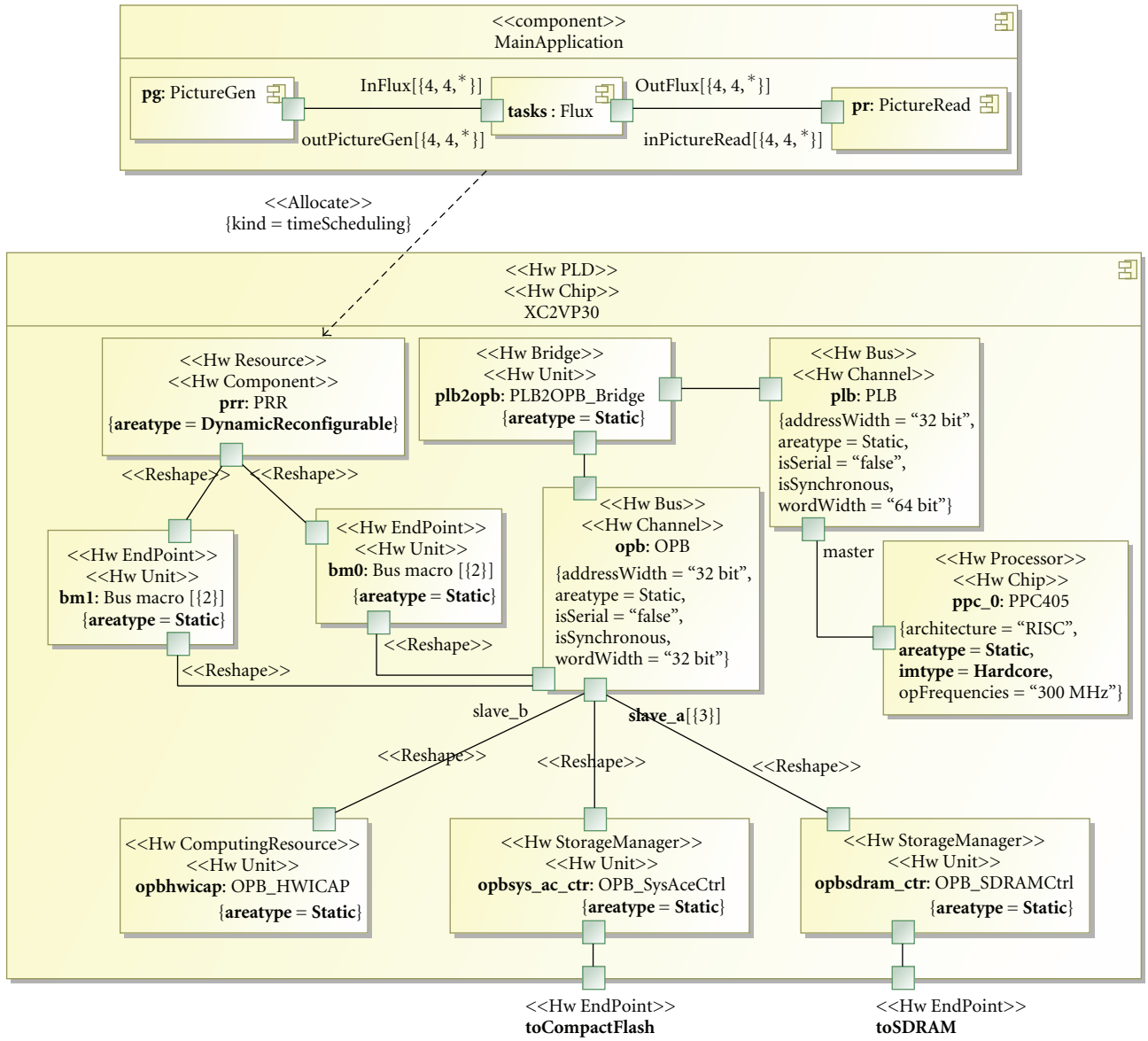


Figure 14: Allocation: Level 2.

scheduling of a set of tasks spatially allocated to the same platform resource) notion of allocation as defined in MARTE standard. Figure 14 presents a detailed view of the allocation illustrating the mapping of the application onto the PRR reconfigurable portion. Due to space limitations we have not presented the last level of allocation in which the application is finally placed on the hardware accelerator **HwAcc** for execution. The XUPBoard also contains a global Clock (**clk**) and the CompactFlash (**cf**) and DDR SDRAM (**ddr**) memories. The concepts introduced in our approach can be modified and extended to manipulate other PDR supported architectures such as introduced in [37, 38] and can be adapted to serve new emerging technologies such as explained in [19, 35].

This point is validated as we present another PDR architecture as shown in Figure 15. The figure shows the merged functional/physical modeling of a PLB ICAP-based PDR architecture as defined in [37]. We have omitted some of the high level attribute specifications and type definitions in the figure in order to respect the space limitations. However, the modeling clearly illustrates that the PDR modeling methodology that we have proposed can be used as a building block. The model to model transformation rules can be extended by addition of new rules, hence it is possible to implement other existing and future PDR architectures.

Our modeling methodology can also be extended by integrating the MARTE *HwPhysical* arrangement notation which provides rectangular grid-based placement mechanisms in order to bridge the gap between UML diagrams and actual physical layout and topology of the targeted architecture. Unfortunately, due to the current functional limitations of

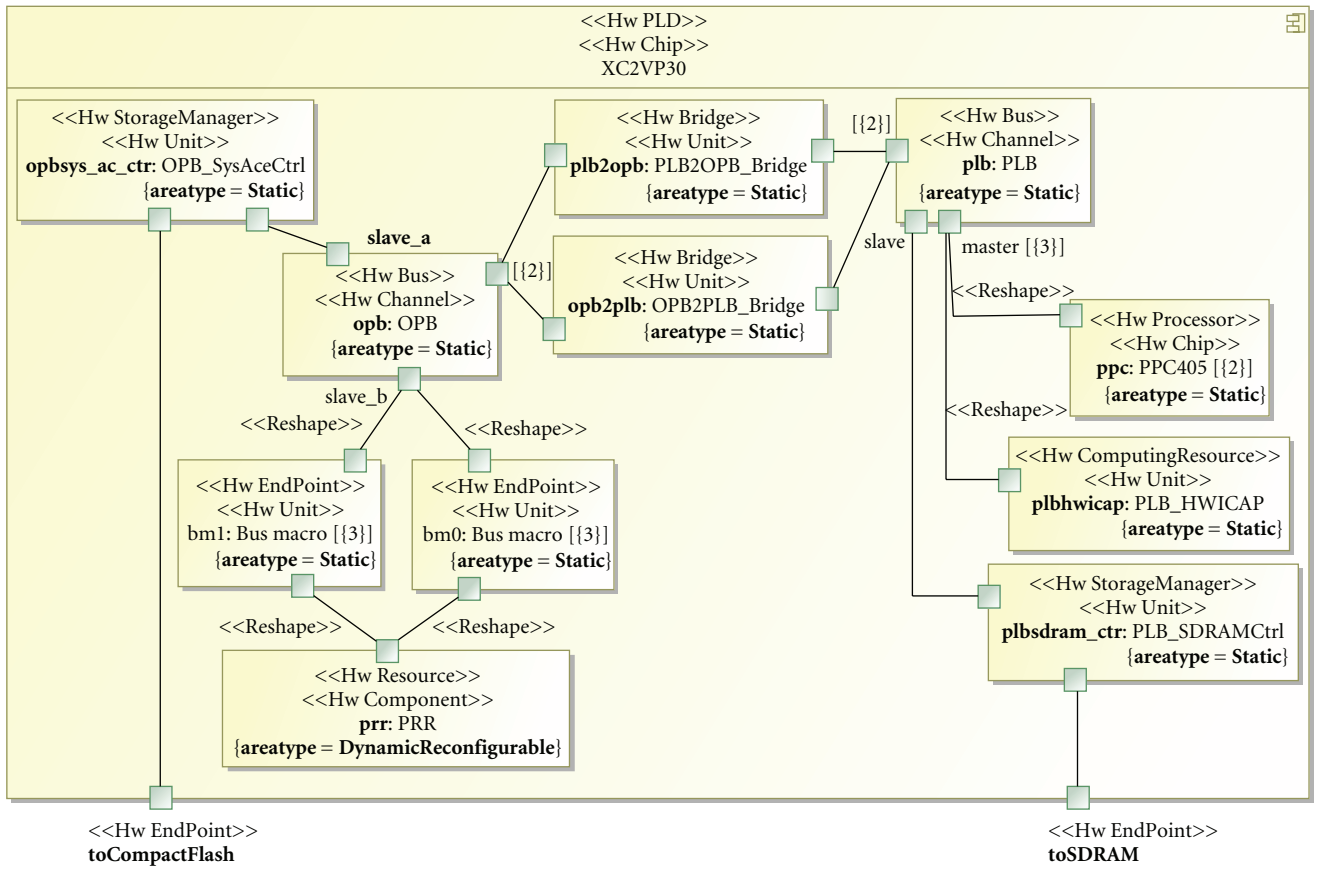


Figure 15: Modeling of a PLB ICAP based reconfigurable architecture.

the modeling tools (Papyrus: http://www.papyrusuml.org/, MagicDraw: http://www.magicdraw.com/), it is not possible to express this view. However, this view could be a potential additional aid to commercial PDR tools such as PlanAhead [43]. Designers can specify the FPGA layout at the MARTE specification level. At the simulation level, designers can accurately estimate if the layout is feasible and determine the number of consumed FPGA resources. Finally using these simulation results, the high-level models can be modified resulting in an effective Design Space Exploration Strategy (DSE) for PDR-based FPGA implementation.

8. Conclusion

This paper presents a novel methodology to implement FPGAs based on an MDE approach using the MARTE standard. For this purpose, modifications have been made to the MARTE specifications to resolve the current limitations for FPGA modeling. This paper introduces notions in the MARTE standard such as those of peripherals and hardware wrappers, which can be adapted to new versions of the standard. These modifications make a direct impact to the corresponding model transformations in order to move from model level specifications to an executable FPGA platform. Further more, they allow us to model a complete SoC on an FPGA. Afterwards we integrate the aspects of Partial Dynamic Reconfiguration using the modified version of the standard. Currently we adhere to the Xilinx-based PDR design flow due to its availability and extendable nature. However our PDR-based methodology can be used as a template in order to model and implement other existing or future PDR-based fine grain reconfigurable architectures. Coarse grain reconfigurable architectures can also be addressed using the GASPARD framework and our design flow. By modeling a complete system (application and architecture) we have defined the first stage of our design flow. In future works, we will detail the controlled deployment level which will allow to link an elementary component with several unique IPs thus creating the concept of configurations, and hence creating part of the reconfigurable controller responsible for managing the self reconfiguration. Finally the enriched RTL level (the level which details the abstract FPGA concepts modeled above) will be able to take the upper model levels as inputs and generate the necessary code required for PDR implementation. The code can then be used as input for commercial tools for final FPGA synthesis.

References

[1] Open SystemC Initiative, "SystemC," 2007, http://www.systemc.org/.

[2] Planet MDE, "Portal of the Model Driven Engineering Community," 2007, http://www.planet-mde.org/.

[3] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–6, Madrid, Spain, August 2006.

[4] Object Management Group, "OMG MARTE Standard," 2007, http://www.omgmarte.org/.

[5] The DaRT Team, "GASPARD Design Environment," 2008, http://gforge.inria.fr/projects/gaspard2.

[6] A. Gamatié, S. Le Beux, É. Piel, et al., "A model driven design framework for high performance embedded systems," Tech. Rep. 6614, INRIA, Sophia Antipolis, France, August 2008, http://hal.inria.fr/inria-00311115/en.

[7] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[8] P. Stevens, "A landscape of bidirectional model transformations," in *Proceedings of the Generative and Transformational Techniques in Software Engineering II (GTTSE '07)*, vol. 5235 of *Lecture Notes in Computer Science*, pp. 408–424, Braga, Portugal, July 2007.

[9] Object Management Group Inc., "MOF QVT Final Adopted Specification," November 2005, http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[10] P. Boulet, "Array-OL Revisited, Multidimensional Intensive Signal Processing Specification," Tech. Rep. 6113, INRIA, Sophia Antipolis, France, 2007, http://hal.inria.fr/inria-00128840/en.

[11] R. Ben Atitallah, E. Piel, S. Niar, P. Marquet, and J.-L. Dekeyser, "Multilevel MPSoC simulation using an MDE approach," in *Proceedings of the IEEE International SOC Conference (SOCC '07)*, pp. 197–200, Hsin Chu, Taiwan, September 2007.

[12] Eclipse, "Eclipse Modeling Framework Technology (EMFT)," http://www.eclipse.org/emft/.

[13] Eclipse, "EMFT JET," http://www.eclipse.org/modeling/m2t/?project=jet.

[14] H. Yu, A. Gamatié, É. Rutten, and J.-L. Dekeyser, "Safe design of high-performance embedded systems in an MDE framework," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 215–222, 2008.

[15] Xilinx, "Two flows for partial reconfiguration: module based or difference based," *Xilinx Application Note XAPP290*, Version 1.1, November 2003.

[16] Xilinx, "Two flows for partial reconfiguration: module based or difference based," *Xilinx Application Note XAPP290*, Version 1.2, May 2004.

[17] Xilinx, "Early Access Partial Reconfigurable Flow," 2006, http://www.xilinx.com/support/prealounge/protected/index.htm.

[18] B. Blodget, S. McMillan, and P. Lysaght, "A lightweight approach for embedded reconfiguration of FPGAs," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, vol. 1, pp. 399–400, Munich, Germany, March 2003.

[19] S. Bayar and A. Yurdakul, "Dynamic partial self-reconfiguration on spartan-III FPGAs via a parallel configuration access port (PCAP)," in *Proceedings of the 2nd HiPEAC Workshop on Reconfigurable Computing (HiPEAC '08)*, pp. 1–10, Goteborg, Sweden, January 2008.

[20] W. Cescirio, A. Baghdadi, L. Gauthier, et al., "Component-based design approach for multicore SoCs," in *Proceedings of the 39th Design Automation Conference (DAC '02)*, pp. 789–794, New Orleans, La, USA, June 2002.

[21] Y. Atat and N.-E. Zergainoh, "Simulink-based MPSoC design: new approach to bridge the gap between algorithm and architecture design," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pp. 9–14, Porto Alegre, Brazil, March 2007.

[22] G. Gailliard, E. Nicollet, M. Sarlotte, and F. Verdier, "Transaction level modelling of SCA compliant software defined radio waveforms and platforms PIM/PSM," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '07)*, pp. 1–6, Nice, France, April 2007.

[23] R. Damasevicius and V. Stuikys, "Application of UML for hardware design based on design process model," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, pp. 244–249, Taipei,Taiwan, January 2004.

[24] W. E. McUmber and B. H. C. Cheng, "UML-based analysis of embedded systems using a mapping to VHDL," in *Proceedings of the 4th IEEE International Symposium on High Assurance Software Engineering (HASE '99)*, pp. 56–63, Washington, DC, USA, November 1999.

[25] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/Scopes '02)*, pp. 18–27, Berlin, Germany, June 2002.

[26] S. Le Beux, P. Marquet, A. Honoré, and J.-L. Dekeyser, "A model driven engineering design flow to generate VHDL," in *Proceedings of the International Workshop on Model Driven Design for Automotive Safety Embedded Systems (ModEasy'07)*, pp. 15–22, Barcelona, Spain, September 2007.

[27] S. Le Beux, *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modeles*, Ph.D. dissertation, LIFL/USTL, Lille, France, 2007.

[28] A. Koudri, D. Aulagnier, D. Vojtisek, et al., "Using MARTE in a co-design methodology," in *Proceedings of the Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML Profile Workshop Co-located with DATE '08*, Munich, Germany, March 2008.

[29] F. Berthelot, F. Nouvel, and D. Houzet, "A flexible system level design methodology targeting run-time reconfigurable FPGAs," *EURASIP Journal of Embedded Systems*, vol. 2008, Article ID 793919, 18 pages, 2008.

[30] M. Boden, T. Fiebig, M. Reiband, P. Reichel, and S. Rulke, "GePaRD—a high-level generation flow for partially reconfigurable designs," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI '08)*, pp. 298–303, Montpellier, France, April 2008.

[31] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, and T. Becker, "Modular partial reconfiguration in virtex FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 211–216, Tampere, Finland, August 2005.

[32] J. Becker, M. Hübner, and M. Ullmann, "Real-time dynamically run-time reconfiguration for power-/cost-optimized virtex FPGA realizations," in *Proceedings of the International Conference on Very Large Scale Integration of System-on-Chip (VLSI-SoC '03)*, pp. 129–134, Darmstadt, Germany, December 2003.

[33] M. Hübner, C. Schuck, M. Kiihnle, and J. Becker, "New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits," in *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, pp. 97–102, Karlsruhe, Germany, March 2006.

[34] C. Schuck, M. Kuhnle, M. Hübner, and J. Becker, "A framework for dynamic 2D placement on FPGAs," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08)*, pp. 1–7, Miami, Fla, USA, 2008.

[35] K. Paulsson, M. Hübner, G. Auer, M. Dreschmann, L. Chen, and J. Becker, "Implementation of a virtual internal configuration access port (JCAP) for enabling partial self-reconfiguration on Xilinx Spartan III FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 351–356, Amsterdam, The Netherlands, August 2007.

[36] E. Cantó, M. López, F. Fons, et al., "Self reconfiguration of embedded systems mapped on Spartan-3," in *Proceedings of the 4th Reconfigurable Communication-Centric Systems-on-Chip Workshop (ReCoSoC '08)*, pp. 117–123, Barcelona, Spain, July 2008.

[37] C. Claus, F. H. Müller, J. Zeppenfeld, and W. Stechele, "A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS '07)*, pp. 1–7, Long Beach, Calif, USA, March 2007.

[38] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "A self-reconfigurable implementation of the JPEG encoder," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '07)*, pp. 24–29, Montreal, Canada, July 2007.

[39] Xilinx, "Fast Simplex Link Channel (FSL)," 2004.

[40] R. Koch, T. Pionteck, C. Albrecht, and E. Maehle, "An adaptive system-on-chip for network applications," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, pp. 1–8, Rhodes Island, Greece, April 2006.

[41] C. Schuck, B. Haetzer, and J. Becker, "An interface for a decentralized 2D-reconfiguration on Xilinx virtex-FPGAs for organic computing," in *Proceedings of the 4th Reconfigurable Communication-Centric Systems-on-Chip Workshop (ReCoSoC '08)*, Barcelona, Spain, July 2008.

[42] Xilinx, "ISE Foundation Software," 2008.

[43] N. Dorairaj, E. Shiflet, and M. Goosman, "PlanAhead software as a platform for partial reconfiguration," *Xcell Journal*, no. 55, pp. 68–71, 2005.

[44] "The Xilinx XUP-V2Pro Board," http://www.xilinx.com/univ/xupv2p.html.

[45] IBM, "The CoreConnect Bus Architecture," white paper, IBM, 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 205149, 9 pages
doi:10.1155/2009/205149

Research Article

vMAGIC—Automatic Code Generation for VHDL

Christopher Pohl, Carlos Paiz, and Mario Porrmann

Heinz Nixdorf Institute, University of Paderborn, Fürstenallee 11, D - 33102 Paderborn, Germany

Correspondence should be addressed to Christopher Pohl, pohl@hni.upb.de

Received 28 November 2008; Accepted 26 March 2009

Recommended by Michael Huebner

Automatic code generation is a standard method in software engineering, improving the code reliability as well as reducing the overall development time. In hardware engineering, automatic code generation is utilized within a number of development tools, the integrated code generation functionality, however, is not exposed to developers wishing to implement their own generators. In this paper, VHDL Manipulation and Generation Interface (vMAGIC), a Java library to read, manipulate, and write VHDL code is presented. The basic functionality as well as the designflow is described, stressing the advantages when designing with vMAGIC. Two real-world examples demonstrate the power of code generation in hardware engineering.

1. Introduction

The notion *automatic code generation* (ACG) envelopes a number of different techniques aimed at simplifying the task of writing a code. Apart from specific implementation details these techniques differ in the level of abstraction exposed to the developer: a very low level of abstraction is given by template-based techniques such as code completion or code insertion. These allow for the generation of code structures with a low complexity (e.g., getters/setters), which are inserted into the code by the user on an explicit (calling an editor function) or implicit (the editor recognizes the beginning of a construct and completes it) basis. This is a very general approach that can be applied in every programming language and in any kind of desired application. Code transformation represents a higher level of abstraction, where a piece of code is translated from a source language into a target language. This is useful if domain specific language (DSL) exists to describe features of a very specific kind of application, which will be implemented in another, more general language. This approach can obviously save a lot of time, as the definition of an application in the very specific and abstract DSL typically takes much less time than implementing it in the specific target language.

Code generators such as the aforementioned exist for various types of applications in computer science, for example, parser generators, database generators, or unified modeling language (UML) tools, rapidly generating production code and saving development costs. Apart from saving time by generating code which would otherwise have to be implemented manually, (correct) generators deliver correct code; additionally, all developmental iterations (with alterations to the specification) can be handled in the source language, again saving time. While these principles and methods are largely applied in the area of software development, hardware developers are supported by very few specific tools like IP-Core Generators or C-to-hardware compilers, covering only a very small area of what could be done with ACGs. Examples for C-based or graphical tools can be found in [1–4]. Each of these tools utilizes code generators for some hardware description language in their specific area, however, this functionality is not exposed to a developer wishing to implement own code generator.

VHDL Manipulation and Generation Interface (vMAGIC) has been developed to fill in this gap by providing a basis for all kinds of VHDL code generators by implementing three important basic tasks:

(i) reading of existing code,

(ii) manipulation of existing code, generation of new code,

(iii) writing of manipulated and/or generated code.

vMAGIC is not a code generator by itself, but a homogenous framework for implementing code generators (and other applications, cf. Section 2). Using vMAGIC accelerates design processes whenever uniform tasks can be automated and reused many times. vMAGIC is free software under LGPL 3 and can be obtained via http://vmagic.sourceforge.net/.

The general concepts and the user interface to the vMAGIC API are discussed in Section 2, as are a very small example and a number of possible applications beyond code generators. In Sections 3 and 4, two examples (HiLDE and HiLDEGART) are provided to demonstrate the power of vMAGIC. This paper concludes with a summary the benefits of the presented approach, and the work in progress is shown.

2. Automatic Code Generation with vMAGIC

The description of the vMAGIC API starts with the implementation of the main functionality of parsing, modifying, and writing VHDL code. The next section is devoted to the API comprising of a set of so-called metaclasses. The description of vMAGIC concludes with an example and an outlook on further use cases for vMAGIC.

2.1. Functionality. The vMAGIC API is a Java library compatible with runtime environments 1.5 and later. Therefore, it is platform independent and usable in command line tools, graphical user interfaces, and scripts. The full functionality is given in the API documentation [5], here the implementation of the basic functionality is described.

(i) Parser. vMAGIC implements a VHDL'93 compliant parser to transform VHDL code into a more convenient internal representation called Abstract Syntax Tree (AST). An AST in general contains all information from the code, while redundancy (parenthesis, semicolons, and so on are implicitly included in the tree structure) is removed; this AST however is shaped in a way optimally suited for the manipulations described next.

The vMAGIC parser was generated using ANTLR 3.1 [6], a powerful parser generator; parsers generated with ANTLR support certain error recovery strategies. This implies that the vMAGIC parser can correct certain unambiguous syntactical errors while parsing VHDL source code, such as additional semicolons in a port or component.

(ii) Modification and Generation of Code. The AST by itself is a tree structure, which is not well suited for human interaction. To hide this structure behind a simple API, a set of so-called metaclasses was defined. Metaclasses combine the functionality to generate or modify specific VHDL constructs and the knowledge how to interact with the AST, such that the developer is using a homogenous API with intuitive functions. For example, `Signal.getIdentifier()` returns the identifier of a signal, `new Process()` creates a new VHDL process.

Objects of metaclasses are created either by the user, defining a VHDL design from scratch, or using a VHDL template. The template is parsed into an AST, which is then parsed by a tree parser generating the metaobjects and discarding the original tree. This approach is, again assuming that the tree grammar is correct, another means of ensuring that the generated code is correct regardless of coding style or context. The metaobjects implicitly define a descendible tree structure, beginning with a `VhdlFile` object with members for `Entity` and `Architecture` objects and so on. User programs work on this metatree rather than on the AST, allowing for intuitive software development for hardware generation or analysis purposes.

There are two different levels of abstraction represented by metaclasses: the so-called low-level classes represent basic VHDL constructs such as signal declarations or processes; the high-level metaclasses combine several low-level classes such as to create complex functionality like registers or state machines. The use of high-level classes implies a higher level of abstraction and therefore an improved coding speed.

(iii) VHDL Writer. To generate VHDL code from an AST, a VHDL Writer based on ANTLR's String Template system was developed. Again, a tree parser is used to analyze the tree and templates are used to generate VHDL code constructs. These templates are defined in a single text file in a very simple format, such that the developers preference in coding style (e.g., the use of lower case or upper case letters for keywords, or using optional identifiers at the end of a process or entity) are implementable by changing this text file.

The vMAGIC designflow, as depicted in Figure 1, follows the three steps as described above.

2.2. vMAGIC API. The complete API functionality of vMAGIC is contained in the metaclasses, accessible via member functions. The most important functions not related to VHDL are `public String toVhdlString()` implemented by all metaclasses and the `public static VhdlFile parse()` function in the VhdlFile class. The `toVhdlString()` function generates VHDL code from every possible VHDL element, such that either the complete design or parts of the design can be viewed as VHDL code (cf. II-C, where VHDL is generated for an architecture). The `parse()` function is an overloaded function, allowing for the creation of VhdlFile objects from various sources. The usage of both functions is demonstrated in Example 1. All other members are VHDL related and can be found in the vMAGIC API-Documentation on the vMAGIC website.

Another important feature of the API is the possibility to process so-called vMAGIC-tags, beginning with "$-*$" in the source code. This is a means to pass additional information from the source code to a vMAGIC application, for example, categorizing files or processes, or giving instructions on what to do with certain parts of the design.

Extracting information from a VHDL-file in vMAGIC is based on high-level functions accessing the AST: generating new code means creating and joining metaobjects. Using the vMAGIC API ensures syntactically correct code. In the next section, the use of metaclasses is demonstrated by means of an example.

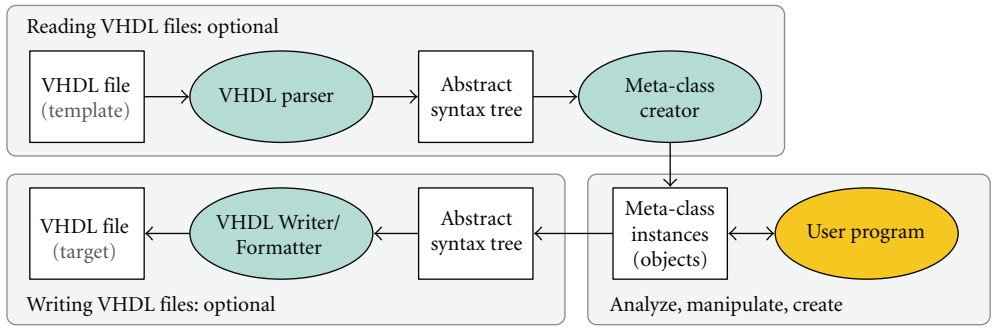


Figure 1: vMAGIC Designflow, reading and writing VHDL is optional. As such a vMAGIC application can be a pure VHDL generator or analyzer.

2.3. Example. The code listed in Algorithm 1 constructs a very simple wrapper file for an entity in the VHDL file *test.vhd*.

In lines 1 et seq. the input file is parsed, the entity is extracted and an output file with the same entity is generated. Lines 5–10 create an architecture *beh* with the component declaration of the entity and the instantiation of and add it to the output file. The `for` loop in lines 11–25 generates registers for all signals in the entity of : line 12 et seq. generate and declare an internal signal of the same type as the port signal, which is then connected to a new register based on the mode of the port signal. If the port signal is an input to (see line 15), the register connects the appropriate signal of the wrapper entity with the intermediate signal (line 16), which is then connected to the instance of (line 17). The registers for output ports are generated in line 18 et seq. After the loop has finished by adding the register to the architecture (line 22), the reset and clock signals are generated and added to the wrapper entity, and the code is printed out.

The output for an entity called , containing one 8 Bit input and one 8 Bit output signal, is given in Algorithm 2. For the tiny entity , creating this very simple code generator takes about twice as much time as creating the wrapper file manually. However, the wrapper generator can be reused on every possible entity, thus saving a lot of time even for this very simple case.

2.4. Advantages and Application Areas. Using a code generator always implies spending additional time on the code generator rather than programming an application manually. However, if the code generator is versatile, and this degree of versatility cannot be reached using methods native to the target language (e.g., Generics in VHDL), then it is very likely that the time spent on the generator is less than the time saved by using it. In VHDL, that point is easily reached due to limitations of VHDL as a computer language: everyday objects such as multiplexers with a generic number of inputs or busses with a generic number of devices cannot be described efficiently in standard synthesizable VHDL. Apart from these limitations, the implementation of DSLs can greatly improve coding speed as shown in the next section.

Apart from code generation, vMAGIC provides important mechanisms to extract information such as signal names, design hierarchies, or generic values from VHDL code. On top of this functionality, any kind of analyzer tool can be implemented. Optimization algorithms can be implemented on the level of VHDL code rather than by accessing netlists. IP-Cores and Algorithms specified with vMAGIC are portable, such that they can be applied to any design.

In the following sections we present a versatile and platform-independent approach to FPGA-based testing, which makes extensive use of the vMAGIC library (available online on the vMAGIC web site). The basic principles of this approach are similar to those of Hardware-in-the-Loop (HiL) simulations, where a real Design under test (DUT) is interacting with a simulated environment. In this case, the DUT resides on an FPGA while the environment is simulated on a host computer, resulting in a high reliability of test results and, in many cases, in a speedup for the simulation itself.

3. HiLDE: A Designflow for FPGA Based Testing

Hardware-in-the-Loop Development Environment (HiLDE) is a cycle-accurate testing framework for performing FPGA-in-the-Loop simulations. HiLDE utilizes vMAGIC to encapsulate a DUT into a *hardware wrapper*, such as to enable the connection to and synchronization with a simulation tool such as MATLAB/Simulink [7], ModelSim [8] or CAMeL-View [9]. In the following, a brief description of the HiLDE wrapper and the use of vMAGIC is given, the basic concept of HiLDE has been published in [10].

There are two main challenges in the creation of HiL simulations: the synchronization of DUT and simulation on the one hand, a high-speed data transfer between DUT and simulation on the other hand. Section 3.1 gives an overview of the synchronization mechanism in HiLDE, while recent and unpublished developments in the HiLDE communication system are described in Section 3.2.

3.1. DUT Access. The HiLDE synchronization system utilizes the properties of synchronous logic to slow down the DUT execution to match the speed of the environmental simulation. This is a very special case, as typical HiL frameworks (such as [11]) must speed up the simulation to

```
1  VhdlFile inFile = VhdlFile.parse("test.vhd");    // parse a VHDL file
2  Entity inEntity = inFile.getEntity("test");      //get the input entity ("test")
3  VhdlFile outFile =newVhdlFile();                 // create a new VHDL file as output
4  outFile.add(inEntity.clone());                   // add a copy of the entity to the out file
5  Architecture arch =new Architecture("beh", outFile.getEntity("test"));
6  outFile.add(arch);                               // create a new architecture in the output file
7  Component comp =newComponent(inEntity);          // create a component from test
8  arch.addDeclaration(comp);                       // declare the component in the out file
9  ComponentInstantiation inst =newComponentInstantiation("inst", comp);
10 arch.addStatement(inst);                         // instantiate the component in the out file
11 for(Signal s : comp.getPort().getSignals()){    // for all signals in the component
12   Signal wire =newSignal(s.getIdentifier() + "_int", s.getType());
13   arch.addDeclaration (wire);
14   Register reg = null;
15   if(s.getMode() == Signal.Mode.IN){            // create an in- or out-register
16    reg =newRegister("regp_" + s.getIdentifier(), s, wire);
17     inst.connect(s.getIdentifier(), wire);
18   } else if(s.getMode() == Signal.Mode.OUT){
19     reg =newRegister("regp_" + s.getIdentifier(), wire, s);
20     inst.connect(s.getIdentifier(), wire);
21   } else { /* handle other modes (INOUT, BUF, ...)*/ }
22   arch.addStatement(reg);                        // and add that register to the out file
23 }
24 Signal rst =newSignal("LRESET_N");
25 rst.setMode(Signal.Mode.IN);                    // create and add reset and clock signals
26  Signal clk =newSignal("clk");
27  clk.setMode(Signal.Mode.IN);
28  inEntity.getPort().addSignal(rst);
29  inEntity.getPort().addSignal(clk);
30  System.out.println(outFile.toVhdlString());
```

Algorithm 1: Java program to generate a wrapper file for the entity which registers inputs and outputs.

real-time level, because the DUT's behavior would change at lower speeds, or it cannot be executed at a lower clock rate at all. This is especially the case when the DUT cannot be isolated from, that is, analog interfaces or other timing critical devices, such as in a production ready controller module. Under the premises of pure synchronous logic, however, the synchronization of DUT and simulation can be solved with the following interface: the hardware interface for HiLDE (see Figure 2) comprises of a bus interface [12] to the host PC, which is connected to the DUT's input and output ports, and the so-called synchronizer. The synchronizer can switch the clocks of th DUT on and off on a "per clock cycle" basis; the user can adapt the number of clock cycles the DUT should run before synchronizing with the simulation. The integration of a DUT into a software simulator such as MATLAB/Simulink is depicted in Figure 3. The simulation itself follows four steps:

(1) read the DUTs outputs and propagate to the simulated environment (as inputs),

(2) read the environments outputs and propagate to the DUTs inputs,

(3) execute a predefined number of clock cycles (acoording to the DUTs I/O data rates),

(4) return to step 1) or end simulation.

3.2. Communication Optimization. In the simulation flow as described above, all I/O data have to be transferred at every clock cycle, resulting in redundant I/O operations where data have not changed. To decrease this overhead, two further concepts were integrated in HiLDE: *event based communication* and *transactors*:

(i) Event-Based Communication. To reduce the number of redundant I/O operations, only data that actually changes has to be transferred. While this is straightforward to be implemented in software (Simulink provides appropriate functions), the hardware wrapper has to be extended. The register of every output port is extended with a mechanism to detect changes at the output. For n_o output ports an additional register with n_o bits stores the results of these detectors, and thus indicates which values must be read by the host computer. The number of additional read operations to retrieve this information is dependent on the word width of the bus to the host computer, resulting in an overall number of read accesses $\tilde{n}_r$:

$$\tilde{n}_r = \Delta(\text{out}) + \left\lceil \frac{n_o}{\text{wordwidth}} \right\rceil, \tag{1}$$

where $\Delta(\text{out})$ is the number of output ports with a new value. Given that n_r denotes the number of read operations in the

```
1  ENTITY test IS
2        PORT (
3              din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
4              dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
5              LRESET_N : IN std_logic ;
6              clk : IN std_logic
7              );
8  END;
9
10  ARCHITECTURE beh OF test IS
11      COMPONENT test IS
12                  PORT (
13                            din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
14                            dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
15                        );
16        END COMPONENT;
17        SIGNAL din_int : STD_LOGIC_VECTOR(7 DOWNTO 0);
18        SIGNAL dout_int : STD_LOGIC_VECTOR(7 DOWNTO 0);
19  BEGIN
20      inst : test
21             PORT MAP (
22                            din => din_int,
23                            dout => dout_int
24                       );
25      regp_din : PROCESS (clk, LRESET_N)
26      BEGIN
27               IF LRESET_N = '0' THEN
28                        din_int <= "00000000";
29               ELSIF clk'event AND clk = '1'THEN
30                        din_int <= din;
31               END IF;
32        END PROCESS;
33        regp_dout : PROCESS(clk, LRESET_N)
34        BEGIN
35                 IF LRESET_N = '0' THEN
36                          dout2 <= "00000000";
37                 ELSIF clk'event AND clk = '1' THEN
38                        dout <= dout_int;
39                 END IF;
40        END PROCESS;
41  END;
```

Algorithm 2: VHDL output of the example Java program. The indentation and the notation of keywords is governed by the String Template file and can be changed to fit the developers needs.

standard HiLDE wrapper, the benefit $n_r/\tilde{n}_r$ is dependent on the relation of I/Os with regularly changing values to the overall number of I/Os in the DUT. In general DUTs with irregularly changing I/Os will benefit from this technique.

(ii) Transactors. Whenever the sequence of events (value changes) is predefined, such as in communication protocols, the number of I/O operations can be reduced even further by implementing adaptors for the simulation and for the FPGA. The amount of savings here is dependent on the complexity of the protocol: instead of transferring all control-signals or control-signal changes, the adaptors detect protocol activity and transfer only the necessary data, such as address and data, the actual protocol handling is processed in the adaptors in the simulation environment and in the FPGA. While the functionality of the HiL simulation is not affected by this method, the amount of I/O operations for a protocol as described in [12] can be reduced by over 90%.

3.3. HiLDE and vMAGIC. The generation of the HiLDE hardware wrapper is a very uniform procedure, usually varying only in the number and width of I/Os, or in the transfer mode as described earlier. The following steps are completed by a Java program utilizing vMAGIC:

(1) parse the DUT and a special template file,

(2) create an instance of the host communication bus and connect the synchronizer to the bus,

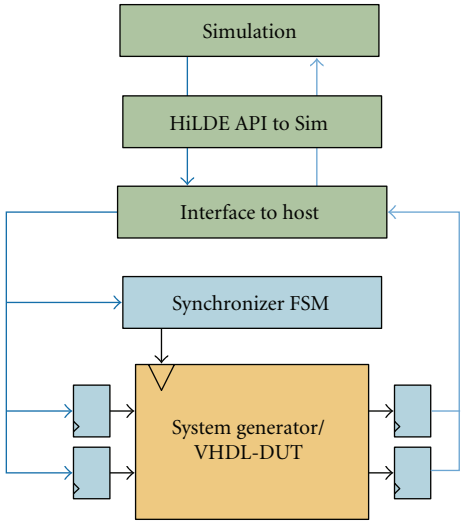


Figure 2: HiLDE hardware wrapper controlling the DUT-IOs and clock.

(3) declare and instantiate the DUT in the template,

(4) create registers for ever I/O port and connect them to the DUT instance,

(5) add all registers to the bus,

(6) generate configuration files for different simulators (currently Simulink, ModelSim and CamelView [13]).

While the manual (error prone) implementation of the wrapper can take hours, the HiLDE Wrapper Generator takes seconds at most. A demo of the generator is available at the vMAGIC project website.

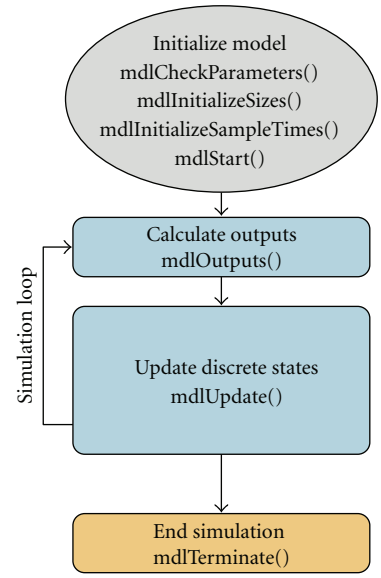
4. HiLDEGART

A logical step after performing a cycle-accurate functional design verification with a simulated environment is to realize a real-time verification of the DUT. The requirements for such a real-time framework are as follows.

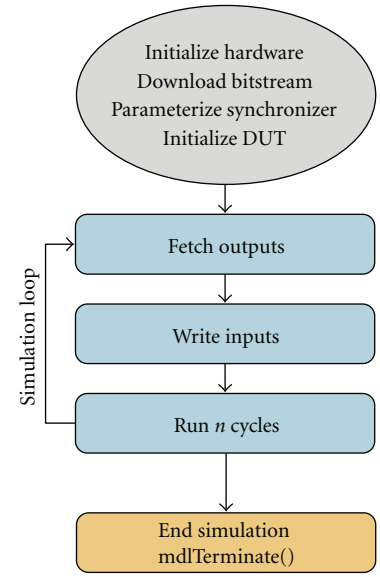
(i) Monitoring of inputs and outputs of an FPGA-based system which is connected to a real testbed. To limit the data that is transferred to the host, resampling and data-based triggering of data recording must be possible, while not influencing the DUT's functionality or timing. Downsampling the data allows for a trade-off between monitoring accuracy and required communication infrastructure.

(ii) In addition to monitoring abilities, parameterization of the DUT must be possible. Switching between different parameter sets during run-time should be possible based on inputs or outputs of the DUT.

(iii) The triggering subsystem as described in what follows, should allow triggers based on boolean operations on inputs and outputs of the DUT as well as combinations of these.



(a) Simulink simulation steps.



(b) HiLDE simulation steps.

Figure 3: For Hilde simulations the standard Simulink S-Function (a) has been extended (b).

One approach is to use real-time verification tools such as ChipScope from Xilinx [14]. However, aside from its limited allowable monitoring time, this kind of tools do not permit an interaction with a DUT. Another approach is logic analyzers, which are expensive and it is very time consuming to set up a test environment. For this purpose, HiLDE for Guided Active Real-Time test (HiLDEGART) was developed. Our approach can be implemented with a standard PC, and it allows the automatic integration of an already functionally verified design to be tested in real-time. In the following section, the concept and realization of HiLDEGART is presented, focusing on the communication between the DUT and the host computer.

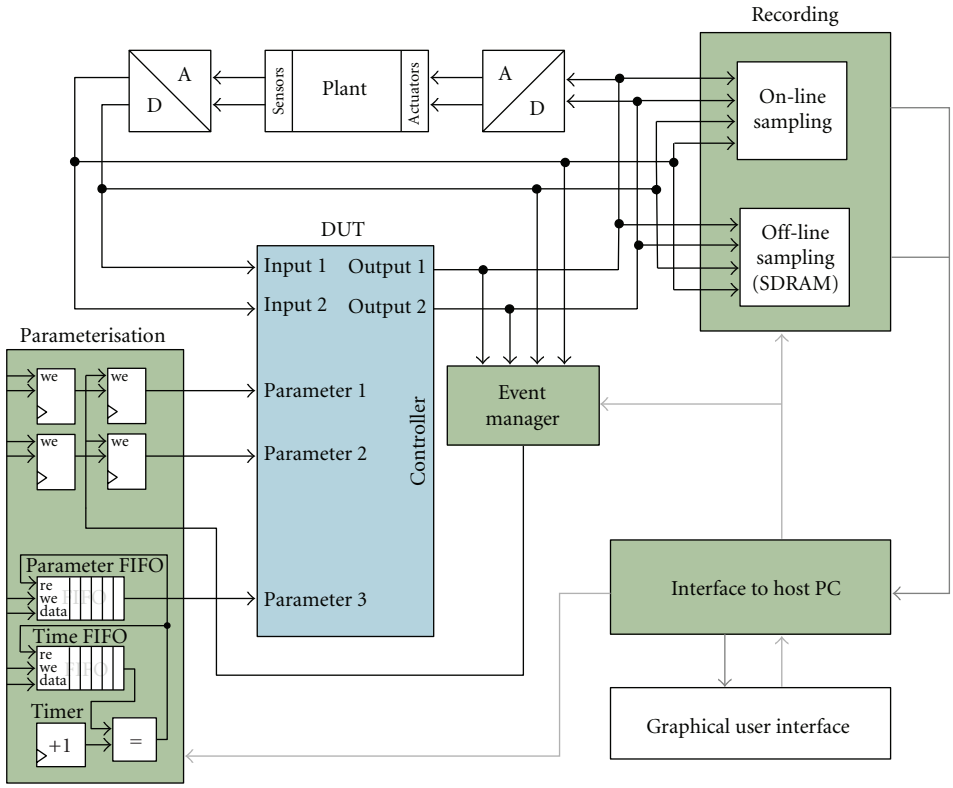


Figure 4: Structure of a real-time FPGA-in-the-Loop scenario utilizing HiLDEGART.

4.1. DUT Access. Figure 4 shows the basic concept of the presented Hardware-in-the-Loop (HiL) framework. The design under test (DUT), a controller, is implemented on an FPGA. The testbed consists of a plant to be controlled and an analog/digital interface. There are three main components surrounding a DUT to be tested with HiLDEGART.

(i) The *Interface to Host-PC* enables the communication between the host PC and the DUT. It works very similar to the interface described in Section 3.1 The main difference is the use of embedded FIFOs and external SDRAM memory to assure meeting the required sampling rates, as explained in what follows.

(ii) By utilizing the *Recording*-Block, the user has the choice to select a specific sampling rate for each port of a DUT using this module. There are two kinds of sampling mechanisms, real-time and offline sampling. Real-time sampling enables the visualisation of the selected signals at run time—the amount of signals that can be visualized in real-time is limited by external factors (e.g., I/O-bandwidth). For offline sampling, an SDRAM memory directly attached to the FPGA is used for buffering the data, allowing for very high sampling rates. The buffered data, as opposed to real-time data, is transferred to the host for visualization after the simulation has finished.

(iii) The *Event Manager* allows basic compare operations to generate events. These events may be combined by boolean operators to form conditions like $(A > \widetilde{A}) \wedge \neg(B = \widetilde{B})$, where A and B are the ports and $\widetilde{A}$ and $\widetilde{B}$ are values defined by the user at run time. With the resulting events, either the changing of the sampling rates or the changing of the parameters of the design can be triggered in real-time. Additionally, the events can be used to start or stop recording.

The presentation of I/O values and all configuration tasks are controlled via a GUI, which has been implemented using Trolltech's platform-independent programming environment Qt [15] in combination with QWT [16]. The project files describe the hardware interface including addresses and number representations (e.g., fix point/binary configuration). They are generated by a vMAGIC application based on user annotations (vMAGIC-tags, cf. Section 2) in the VHDL code. The GUI is automatically generated based on the interface description, including graphs, LCD-like displays for current values, and input boxes for parameters, as can be seen in Figure 5.

The automatic generation of the HiLDEGART hardware wrapper using vMAGIC is similar to the process as described in Section 3.3. The typical tool-flow for HiLDE and HiLDEGART as well as a test-case is described in the following section.

5. vMAGIC Toolflow and Example

Generating the hardware wrappers for HiLDE and HiLDEGART is an application of the complete vMAGIC designflow

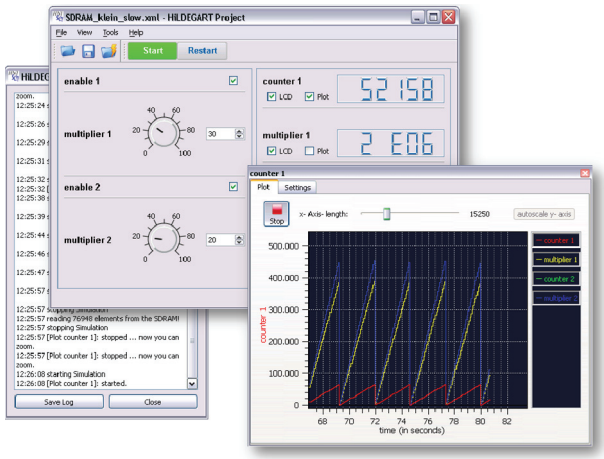


FIGURE 5: Main-, Log-, and Plot-Window of HiLDEGART. The GUI is generated from an XML file generated by a vMAGIC application.

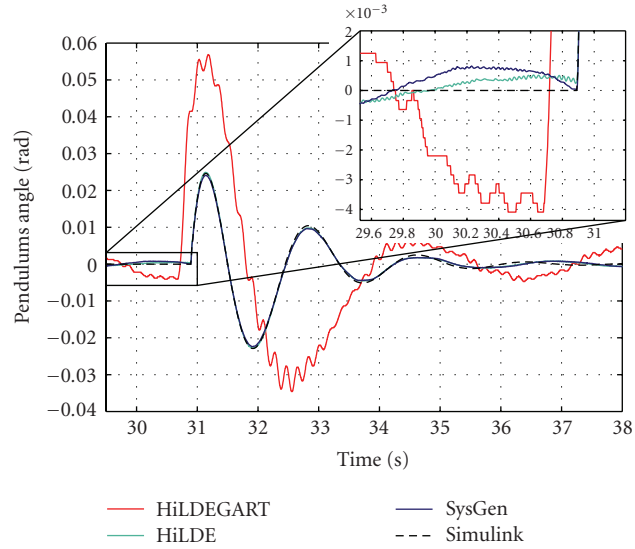


FIGURE 6: Inverted pendulum controller: angle of the pendulum.

as depicted in Figure 1. The starting point of the flow is a VHDL file containing the DUT's entity definition (if no internal signals should be monitored the DUT itself can be described in any HDL), which is then analyzed by vMAGIC. The user program generates the DUT-specific wrappers according to the specifications described in Section 3 respectively, 4 and generates the configuration files for a HiLDE simulation or HiLDEGART. These configuration files contain information regarding hardware addresses, sampling rates and number formats (e.g., fixpoint position). As the number of formats and sampling rates cannot be deduced from the hardware interface, they are supplied via vMAGIC-tags in the source code or directly in the GUI. This completes the vMAGIC specific part; after this, the wrapper and design files have to be synthesized using vendor specific tools. After the FPGA bitstream has been generated, the simulation is conFigurered using the configuration files and the simulation can be started.

As a case study, an inverted pendulum controller was designed using Xilinx' Simulink-based System Generator. First, a model of the pendulum and a controller to balance this pendulum are created using Simulink. The controller is then reimplemented in hardware blocks using the System Generator Toolbox. Figure 6 shows the difference between the continuous Simulink controller and the (time- and value-) discretized hardware controller (SysGen). After the System Generator model has been simulated it can be tested in real hardware using the HiLDE-flow, still using the software model of the pendulum; the result is depicted as *HiLDE*. There are very small differences between the simulation and real hardware due to the internal number representation in the system generator. The last step is to use HiLDEGART to monitor the controller in the real control loop, depicted as *HiLDEGART*. The differences between the HiLDE and the HiLDEGART simulations are due to the inaccurate modelling of the pendulum (e.g., A/D conversion effects and plant dynamics).

6. Conclusions and Outlook

In this paper, vMAGIC, a Java library for automatic code generators for VHDL has been presented. Its functionality and the associated design flow have been shown alongside with examples for vMAGIC's analysis and generation capabilities. The application areas and advantages of a vMAGIC-based designflow have been described.

The vMAGIC API has been released under LGPL 3 on sourceforge.net and can be freely downloaded and used for personal research or commercial uses. It is very usable and reliable, but by no means complete, as many useful features have not been implemented yet. We keep adding functionality to the library and we are planning to create a library on top of vMAGIC that will be able to do semantic operations as well.

Acknowledgments

This work was developed in the course of the "Collaborative Research Center 614 - Self-Optimizing Concepts and Structures in Mechanical Engineering," University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

References

[1] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: delftworkbench automated reconfigurable VHDL generator," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 697–701, Amsterdam, The Netherlands, August 2007.

[2] S. McCloud, "Catapult C Synthesis-Based Design Flow: Speeding Implementation and Increasing Flexibility," Mentor Graphics White Paper, 2004.

[3] *Synplify Users Guide*, Synopsys, Mountain View, Calif, USA, 3rd edition, 2008.

[4] *System Generator Users Guide*, Xilinx, San Jose, Calif, USA, 10th edition, 2008.
[5] C. Pohl and R. Fuest, *vMAGIC API Documentation*, Heinz Nixdorf Institute, Paderborn Germany, 2008.
[6] T. J. Parr and R. W. Quong, "ANTLR: a predicated-*LL*(*k*) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.
[7] *Simulink Users Guide*, The Mathworks, Natick, Mass, USA, 2008.
[8] *ModelSim Users Guide*, Mentor Graphics, Wilsonville, Ore, USA, 6th edition, 2008.
[9] *CAMeL-View Users Guide*, iXtronics GmbH, Paderborn, Germany, 6th edition, 2008.
[10] C. Paiz, C. Pohl, and M. Porrmann, "Reconfigurable hardware in-the-loop simulations for digital control design," in *Proceedings of the 3rd International Conference on Informatics in Control, Automation and Robotics (ICINCO '06)*, pp. 39–46, Setubal, Portugal, August 2006.
[11] H. Hanselmann and F. Schutte, "Control system prototyping productionizing and testing with modern tools," in *Proceedings of the 38th International Intelligent Motion Conference*, pp. 9–16, Intertec International, Nurnberg, Germany, June 2001.
[12] H. Kalte, M. Porrmann, and U. Rückert, "A prototyping platform for dynamically reconfigurable system on chip designs," in *Proceedings of the IEEE Workshop Heterogeneous Reconfigurable Systems on Chip (SoC '02)*, Hamburg, Germany, April 2002.
[13] CAMeL-View, "CAMeL-View Virtual Engineering Workbench Reference Guide," iXtronics GmbH, Paderborn, Germany, 2004.
[14] *ChipScope Users Guide*, Xilinx, San Jose, Calif, USA, 10th edition, 2008.
[15] Trolltech, "Qt—cross-platform application framework," http://trolltech.com/.
[16] U. Rathmann, "Qwt—Qt Widgets for Technical Applications," http://qwt.sourceforge.net/.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 942930, 10 pages
doi:10.1155/2009/942930

Research Article

A Design Technique for Adapting Number and Boundaries of Reconfigurable Modules at Runtime

Thilo Pionteck, Roman Koch, Carsten Albrecht, and Erik Maehle

Institute of Computer Engineering, University of Lübeck, 23538 Lübeck, Germany

Correspondence should be addressed to Thilo Pionteck, pionteck@iti.uni-luebeck.de

Received 30 November 2008; Accepted 17 May 2009

Recommended by Michael Huebner

Runtime reconfigurable system-on-chip designs for FPGAs pose manifold demands on the underlying system architecture and design tool capabilities. The system architecture has to support varying communication needs of a changing number of processing units mapped onto diverse locations. Design tools should support an arbitrary placement of processing modules and the adjustment of boundaries of reconfigurable regions to the size of the actually instantiated processing modules. While few works address the design of flexible system architectures, the adjustment of boundaries of reconfigurable regions to the size of the actually instantiated processing modules is hardly ever considered due to design tool limitations. In this paper, a technique for circumventing this restriction is presented. It allows for a rededication of the reconfigurable area to a different number of individually sized reconfigurable regions. This technique is embedded in the design flow of a runtime reconfigurable system architecture for Xilinx Virtex-4 FPGAs. The system architecture will also be presented to provide a realistic application example.

1. Introduction

Runtime partially reconfigurable FPGA devices like those of the Xilinx Virtex-4 and Virtex-5 series allow system designers to reuse hardware resources over time. Individual modules can be replaced at runtime so that only currently needed processing units (PUs) have to be instantiated in the FPGA. System designs making use of this feature typically comprise only one or few reconfigurable regions (PR regions) with fixed sizes, and locations, allowing for only a fixed number of reconfigurable PUs. For such systems, no special system architectures or design approaches are needed. The system structure is static while only PUs with fixed links are exchanged at runtime. Design tools such as Xilinx early access partial reconfiguration (EAPR) tools [1] and recent versions of the PlanAhead [2] floorplanning tool provide reasonable support for such designs.

Yet, as soon as the number, sizes, and locations of PR regions in the reconfigurable system partition are not static anymore, system layout becomes complicated. With varying communication needs of a changing number of PUs of different sizes mapped onto diverse locations, the system architecture has to provide mechanisms to back this flexibility. This feature is not provided by commonly used system architectures based on buses or point-to-point connections for the communication network. These kinds of communication infrastructures require that all possible combinations of PUs have to be determined at design time so that the communication infrastructure can be dimensioned to support all possible scenarios, resulting in a huge hardware overhead. In addition, current design tools put tighter restrictions on system layout than imposed by the actual hardware. The number, sizes, and locations of reconfigurable regions have to be fixed at an early stage in the design process. There are no provisions for adapting this spacial partitioning later. As a consequence, each PR region has to be dimensioned according to the footprint of the largest PU. Such an approach results in a potential waste of logic resources and is not satisfactory as it is technically well feasible, for example, to move the border between two neighbouring regions in order to provide more space for the one module if the other is currently not needed or requires less resources.

To solve these issues, a system architecture for runtime reconfigurable designs based on Xilinx Virtex-4 FPGAs is

proposed. The reconfigurable system partition is divided into tiles which can be replaced individually at runtime. PUs mapped onto these tiles are connected by a topology adaptive network-on-chip (NoC). The boundaries of the tiles within the reconfigurable partition can be adjusted at runtime, allowing an area efficient mapping of PUs of unequal complexity. While the basic principles of this system architecture were already presented in [3], the design methodology for exploiting the features of the system architecture has not been available till now. This contribution closes this gap by presenting a self-contained technique that allows to change the boundaries and number of PR regions using the available tools, yet circumventing their restrictions. The technique will be demonstrated by means of a comprehensible example, and it will also be shown how this technique facilitates the design of adaptive runtime reconfigurable systems.

The rest of the paper is structured as follows. Section 2 discusses related work in the area of runtime reconfigurable system architectures for designs with a variable number of exchangeable PUs. A brief overview of the proposed system architecture is given in Section 3 and provides the motivation and context for the design technique presented in Section 4. Section 5 demonstrates the advantages when applying the design technique to runtime reconfigurable systems, and, finally, Section 6 summarises the contribution of this paper.

2. Related Work

In [4, 5] an architectural template for runtime reconfigurable systems is presented. The idea bases on algorithmic skeletons which provide a separation of the structure of the computation from the computation itself. Algorithmic skeletons were first introduced in the 1980s by [6] and were extended to dynamic reconfigurable computing in [4]. Here, they act as a kind of abstraction layer on top of an FPGA making the concept independent from a specific FPGA. The reconfigurable fabric is divided into tiles of equal or unequal sizes which are used by a skeleton dispatcher for the hardware realisation of algorithmic skeletons. Algorithmic skeletons are written in VHDL and are enfolded with a wrapper that handles all communications. Mapping the algorithmic skeletons to tiles is done manually by a pattern designer. The pattern designer also has to guarantee that enough communication resources between tiles are available so that point-to-point communication between skeletons is possible as well as communication with a central storage unit. The partitioning of the FPGA in tiles is done at design time and cannot be adapted at runtime.

The idea of adapting the size of PR regions to the size of the actually implemented PU was also picked up in [7]. At design time, a Virtex-II FPGA is partitioned into fixed sized configuration slots, each adjacent to a routing channel. The configuration slots are used at runtime to implement the PUs. In case of a PU not occupying the complete height of a configuration slot, additional PUs with the same width may be mapped onto the same slot. This is done by online adapting the addresses of the configuration data of the PUs at the granularity of CLBs. Connections between PUs and the static part of the system are provided by routing channels. Each routing channel is connected to the static system and consists of vertical links. In case a PU has to be connected to the communication network, the vertical communication links at the position of the PU interface are exchanged by special communication primitives providing horizontal and vertical links. In order to ease online routing, each PU has an LUT-based communication interface at a predefined position, for example, in the lower right corner of the PU. The design flow required to build up such a system is described in [8] and bases on Xilinx JBits class library [9].

A design methodology for generating on-chip communication infrastructures for partially reconfigurable FPGAs with a variable number of PUs is also presented in [10]. The PR region is partitioned into several homogeneous tiles which all provide identical links to the communication network. This allows the mapping of PUs onto any set of tiles in the PR region. The mapping itself is determined either at design time or at runtime by system specific placement algorithms. Address mapping of PUs to tiles of the PR regions is handled transparently so that a position-independent communication can be guaranteed. The communication infrastructure is divided into five abstraction layers, simplifying system adaption to the resources and reconfiguration capabilities of a specific FPGA.

In [11, 12] COMMA, a methodology for automatically generating a communication infrastructure for systems targeting Virtex-4 FPGAs, is introduced. The system layout consists of fixed sized PR regions surrounded on the left and right side by slots reserved for the communication infrastructure. The communication infrastructure is determined at compile time by analysing the communication requirements between PUs based on a given application. An ILP-based approach is used to assign PUs to PR regions so that the number of wires in the routing channels is reduced as well as the communication delay. Communication between adjacent PUs in the same column is done without accessing the communication network. PUs may span adjacent PR regions, and it is also possible to implement several smaller PUs on one PR region. Reconfigurable data ports are used to connect the I/O pins of the PUs to the communication infrastructure at reconfiguration time.

An early work providing a system infrastructure and design methodology to set up runtime reconfigurable systems with an arbitrary number of PUs to be exchanged at runtime is presented in [13]. Targeting the Xilinx Virtex-E FPGA series, the system is divided into a static infrastructure and a number of so-called Dynamic Hardware Plugins (DHPs). The static infrastructure is routed in such a way that no nets cross any DHP regions. This is achieved by a modified version of the router. DHPs are implemented in separate designs by using a gasket interface which on the one side provides fixed interconnect points to communicate with the static infrastructure and on the other side prevents logic from being placed and nets from being routed outside a dedicated area. A special tool called PARBIT is used to extract the configuration code for the DHP from a bitfile and to retarget it into a similar sized region of the FPGA in the final system.

The main difference between the related work and the system architecture and design technique presented in this paper is that here tiles and, hence, PR regions can either be used to implement PUs or components of the communication network. There is no need to provide dedicated routing channels next to PR regions, and PR regions can directly adjoin each other. System connectivity is provided at a level of tiles by mapping components of the communication network onto tiles. In addition, the initial partitioning of the reconfigurable system partition into PR regions can be adapted at runtime. Thus, the presented architecture offers two degrees of freedom for the system layout: mapping of PUs onto different PR regions and resizing of PR regions at runtime.

3. System Architecture

The basic structure of the runtime reconfigurable system architecture is depicted in Figure 1. This architecture takes into account the reconfiguration capabilities and physical restrictions of the Xilinx Virtex-4 FPGA series. Yet, its principal ideas are device independent. A detailed description of the architecture is given in [3]. Here, a brief summary is given in order to provide the background and motivation for the proposed design technique.

The FPGA logic area is divided into a static and a reconfigurable partition which in turn is subdivided into a grid of equally sized tiles. The reconfiguration control logic including the internal configuration port (ICAP), I/O controller, system controller, and all static PUs resides in the static partition. PUs to be exchanged at runtime as well as the communication network to interconnect them are realised onto the grid of tiles in the reconfigurable partition. For the communication network a topology adaptive packet-based NoC called Configurable Network on Chip (CoNoChi) [3, 14] is used. CoNoChi comprises 16-bit wide virtual cut-through switches with four full-duplex links. Routing is done by means of locally stored routing tables supplied by a global control unit. The NoC protocol supports different priority classes for sending routing tables, control messages, and data.

Four different types of basic building blocks can be mapped onto the grid of tiles in the reconfigurable partition: horizontal and vertical communication links, network switches, and application specific blocks, for example, PUs. Each block provides, at the same position, communication links to adjacent blocks so that each block can be connected with the other. The communication links consist of so-called bus macros which are communication lines crossing the border between two blocks with endpoints routed to LUTs on either side of the border. At runtime, blocks mapped onto tiles can arbitrarily be exchanged, allowing to set up a random NoC structure and arrangement of PUs. Each tile consists of 12×16 Configurable Logic Blocks (CLBs) and is aligned to the reconfiguration capabilities of the Xilinx Virtex-4 FPGA series. While this size fits well with the resource requirements of a CoNoChi switch of about 120 CLBs, PUs may require to combine the resources of several tiles. From the hardware point of view, this is feasible. However, the design flow described in [1] to implement partially reconfigurable designs thwarts this approach. Based on [1], the boundaries of any individually reconfigurable region have to be fixed exactly once at design time. Thus, an adaption of the boundaries of tiles is not possible afterwards, in particular, not at runtime. The design technique described in the next section circumvents this constraint.

4. Adapting the Number of PR Regions

The design technique bases on Xilinx EAPR tools and also uses the PlanAhead floorplanning tool. Compared to the former design techniques for building runtime reconfigurable systems [15], the EAPR tools significantly ease the design process and impose fewer restrictions on the system layout. In the context of this paper, the main advantages of the EAPR tools are that PR modules no longer have to cover the whole column of an FPGA and that signals of the static components may cross PR regions without the use of bus macros. The first feature allows with some restrictions to define PR regions of almost arbitrary shape and, thus, an adaptation of the PR region to the size of a PU. As I/O signals of the PR region have to be routed through bus macros spanning the boundary between PR region and adjacent region, one PR region cannot be resized without adapting the adjacent region. In order to omit online placement and routing, all possible scenarios of PR locations and sizes for a given system have to be identified at design time. Relating to the system architecture described before, this results in the necessity to generate separate configuration files for all different tile locations and tile types. In the EAPR flow, the boundaries constraining the PR regions must be defined once at the beginning. Hence, the entire flow has to be run again when number, size, or locations of PR regions are changed. Moreover, different numbers of PR regions require different numbers of bus macros and, thus, result in different top-level netlists. Therefore, the proposed merging and separating of PR regions at runtime makes it necessary to combine the results of different runs of the EAPR tool flow. This leads to an issue with static nets through PR regions which might be routed in different ways in multiple, independent runs of the flow. Functional failures or even short-circuits may then occur when merging or separating PR regions by reconfiguration. Thus, a technique has to be set up which guarantees that these static routes are consistent across various runs of the tools.

4.1. Technical Background. The actual implementation process for partially reconfigurable systems is divided into several phases. Initially, rectangular regions in the FPGA grid are defined for the PR modules in order to constrain routing and logic into the corresponding region. After this, bus macros are locked to positions on region boundaries to allow boundary-crossing communication. A *static design implementation* is then built from the top level design while leaving out the PR modules. Subsequently, the PR modules are implemented individually, again including the top level design. Finally, a desired set of PR module implementations

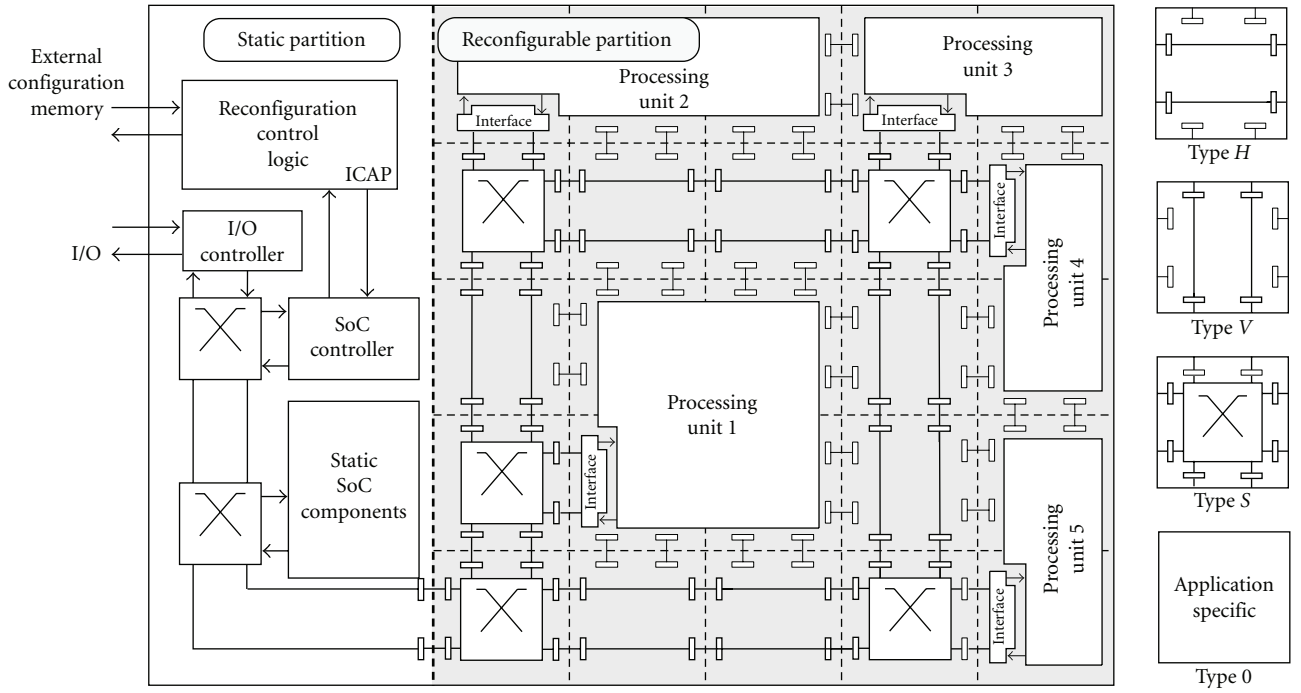


Figure 1: System architecture [3].

is merged into the static design implementation. Configuration bitstream files are generated for the complete design as well as for partially reconfiguring individual PR modules.

Ideally, any static logic and routing is kept out of the PR regions. In practice, however, there can be externally given constraints that make it necessary to place, for example, an input buffer for a signal entering the FPGA inside such a region. The EAPR tool flow manages these situations by keeping track of which FPGA resources are occupied during the static design implementation phase. These resources are then prevented from being used again by PR modules so that there are no conflicts when it comes to merging static and reconfigurable implementations.

These precautions taken by the tools, however, are only effective within a single scenario. Let S_k be a scenario characterised by a certain combination of a top-level design, static modules, and a floorplan of PR regions. Changing the number or sizes of PR regions means switching from S_k to S_l ($k \neq l$) and, thus, requires the top-level design and the static modules to be reimplemented. The resulting static design implementation of S_l may use different resources within PR regions than does its counterpart from S_k. Hence, the PR modules built within S_l are likely to be incompatible with S_k. Checks within the tools prevent straight-forward approaches like trying to reuse the resource exclusion information from S_k for S_l from being successful, and so a different solution must be found.

4.2. Proposed Design Technique. The proposed technique for exactly matching static routes in PR regions of different projects for the same system is illustrated in Figure 2. Assuming two or more modular PR designs that share the same static elements and define the same interface between the static and the reconfigurable parts, the basic idea is to first implement the common static elements in the context of one design and then to transfer the result into a *hard macro* for replacing the static elements defined in the other design. A hard macro is a completely prerouted and preplaced block of logic circuits and interconnects. A hard macro can be instantiated at HDL level multiple times and can be locked to different places inside the FPGA. Hard macros allow parts of a design to be reproduced in exactly the same way throughout different versions of the design.

As the static elements of the different scenarios were functionally identical before and as all static elements are resembled by the hard macro, this approach keeps the affected design consistent. The technique comprises five steps and operates on netlists of the modular PR designs created with standard tools. Each step will be explained in the following by means of an illustrative example implemented on a Xilinx Virtex-4 FX60 FPGA. The example is shown in Figure 3 and consists of two different scenarios. The static part of the design comprises external I/O connections, circuitry for receiving and sending data, and a control unit. In addition, there are four different types of PUs: A *NULL* module just forwards the incoming data, an *XOR* module calculates an exclusive-or on the data, a *DES* encryption module gathers 64 bit of data and then performs a DES encryption, and an *AES* module which collects and encrypts 128 bit. As the AES module requires more resources than the other modules, the design comes in two variants. The dark shaded modules in each PR region of Figure 3 make up a scenario S_1 while the lighter shaded modules correspond

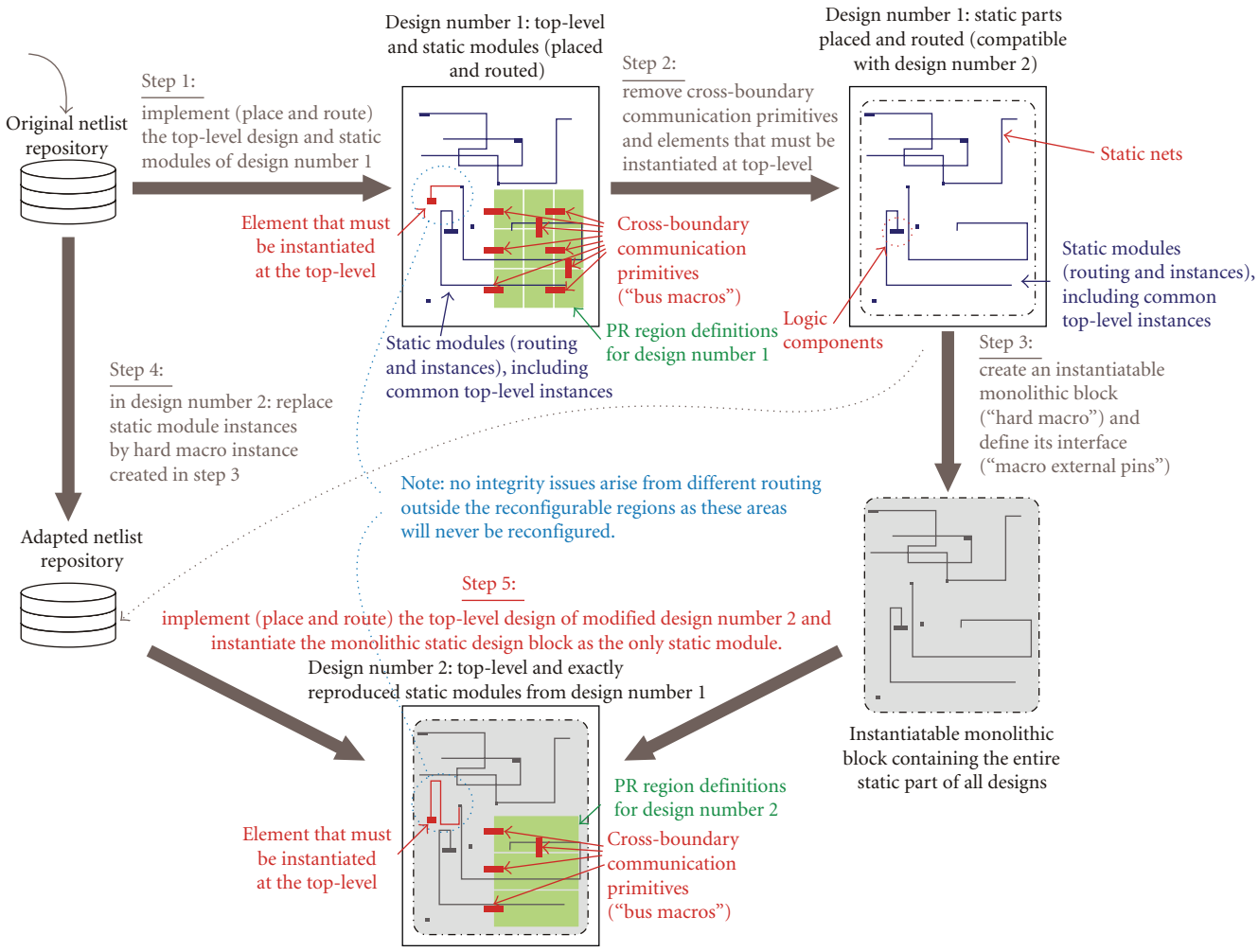


Figure 2: Exact cross-project reproduction of static design artifacts within PR regions.

to scenario S_2. Thus, the first scenario (S_1) defines four equally sized PR regions, and the second (S_2) defines three PR regions. In S_2 the two upper regions are combined to one in order to provide space for the larger AES module implementation. The corresponding floorplans created with PlanAhead are shown in Figure 4.

Step 1 (implementing the static parts of scenario S_1). Seen from the outside, the top-level design provides all the external ports necessary for communicating with the design, including a clock input. Internally, the top-level design instantiates a single static module which contains all the static components of the design. At the same hierarchy level as this static module, the four PR modules of scenario S_1, several bus macros, I/O buffers for external connections, and clocking primitives are instantiated. As, later on, the static part of the design will be extracted as a hard macro that has to be connected to the bus macros for the communication with the PR regions, precautions are taken not to "loose" the pins of the static module. All signals between the static module and bus macros are routed through explicitly instantiated FPGA primitives—1:1 look-up tables (LUT1) programmed with the identity function. These LUT1 instances are constrained to exact locations close to the bus macros. So, signals from and to static modules become available at exactly known locations in the FPGA. Figure 5 gives a close look on the actual routing between the static design and bus macros residing on the boundary to a PR region. One end of the corresponding nets is connected to the bus macros while the other end is connected to LUT1s which are part of static design. The implemented design is shown in Figure 6. It does not include any PR module implementation as these modules were considered as yet unimplemented black boxes.

Step 2 (removal of primitives instantiated at top-level). In order to turn the static design into a re-instantiatable hard macro, the static design has to be adapted in FPGA Editor. This tool is first used to save routing information of nets which must be included in the hard macro, for example, clock nets. As hard macros cannot contain these routing

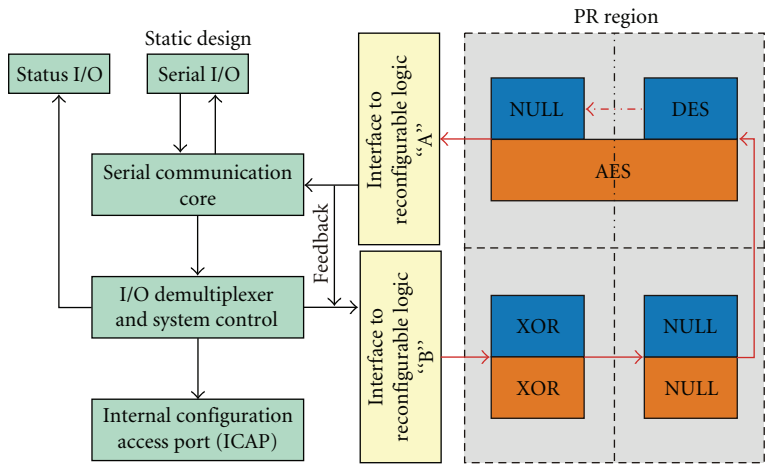
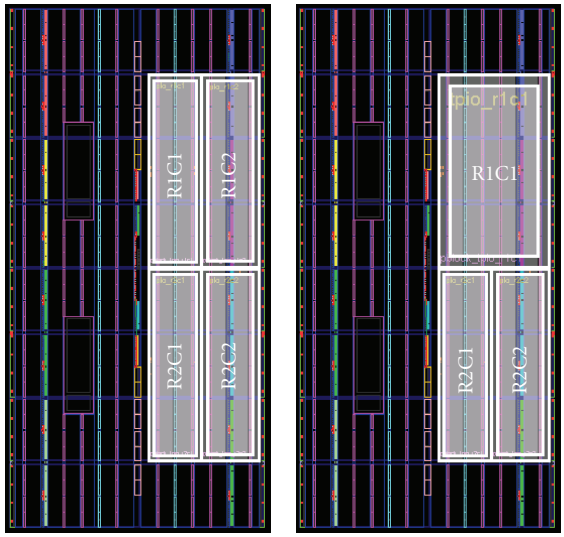


Figure 3: Example design.



(a) scenario S_1: four equally sized PR regions

(b) scenario S_2: upper two regions combined to one

Figure 4: Two scenarios (PlanAhead floor-plans).

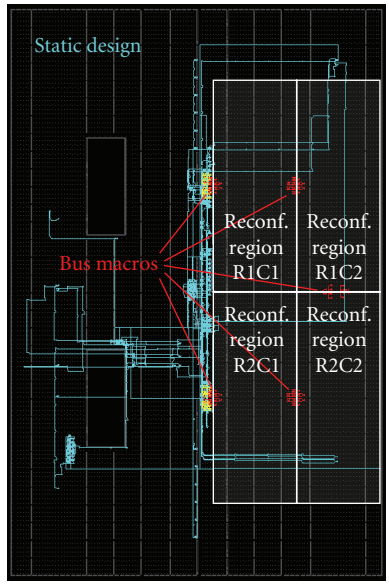


Figure 6: Static design intersecting PR regions.

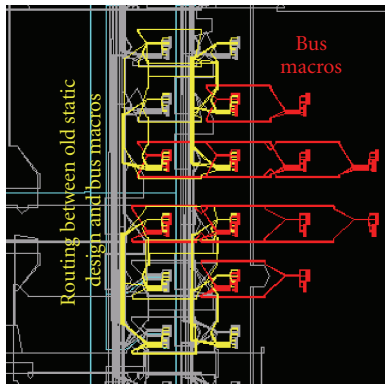


Figure 5: Routing between static design and bus macros.

information, data have to be saved in a separate file to reproduce routing of affected nets later during the design process. In a next step, top-level elements such as bus macros and clocking primitives are removed from the design. These elements also must be immediate children of the top-level of scenario S_2 and, thus, must not be added again to scenario S_2 when including the hard macro. An example of such an element is given in Figure 2. The element encircled in the floorplan after Step 1 is removed in Step 2. As the net connected to this element crosses the PR region and, thus, has to be part of the hard macro, the net is segmented by routing it through an explicitly instantiated FPGA primitive which is part of the hard macro.

Step 3 (creating a static hard macro). The remaining design consists of modules and nets of the static part of scenario S_1. These elements are also part of scenario S_2. In order

to replace these elements later in scenario S_2, the design is converted into a hard macro, and "external pins" are assigned for the clock input and for the known locations of signals to and from the PR regions. In the following, this hard macro will be referred to as the *static hard macro*.

Step 4 (implementing the static parts of scenario S_2). In contrast to the top-level design for S_1 the top-level design for S_2 is heavily thinned out; for example, the only external port of the new design is a clock input. There are no other external ports, because all I/O buffers of the system are incorporated into the static hard macro. Besides the static hard macro, the new top-level design instantiates bus macros as needed for scenario S_2, three—instead of four—PR modules, and mandatory top-level elements.

Step 5 (merging scenario S_2 and static hard macro). In the final implementation phase, the saved routing information from scenario S_1 is utilised to reproduce routing, for example, of clock nets. The resulting design is shown in Figure 7. Note that the routing of nets from elements which have to be instatiated at top-level may differ between scenario S_1 and scenario S_2. As these nets have been segmented before by routing through explicitly instantiated FPGA primitives, scenario S_1 and scenario S_2 provide identical links to these nets. Hence, it does not matter whether the routing from these fixed links to the elements differs in both designs. Only one version will be implemented in the final design, and the static area will never be reconfigured.

Finally, PR modules are merged with their respective static designs. Partial bitstreams are created for individually reconfiguring the PR regions. Most importantly, it is considered safe to dynamically load PR modules of either scenario into an FPGA initialised with the other. Note, however, that it might be necessary to blank a region beforehand in order to avoid short circuits. Figure 8 shows both scenarios fully implemented. For switching between the scenarios only PR regions R1C1 ($S_{1,2}$) and R1C2 (S_1) need to be reconfigured.

4.3. Pros and Cons. Although the proposed technique intercepts the standard design flow multiple times, it completely bases on Xilinx tools and does not require any additional tools manipulating the configuration bitfiles. One drawback of the proposed technique is that the placed and routed design has to be edited manually in FPGA Editor. Yet, once the static hard macro is created it can be reused easily in numerous scenarios that share the same static design. In principle, there also exist two alternative approaches for resizing the boundaries of PR regions, though each approach has significant drawbacks compared to the proposed technique.

4.3.1. Avoidance of Static Nets in PR Regions. Static routing that intersects PR regions is most often caused by I/O buffers residing inside these regions. In order to *avoid* such routing, the I/O buffers could alternatively be instantiated inside the PR modules instead of the top-level or static module [16, 17]. The signals would then have to be passed through additional

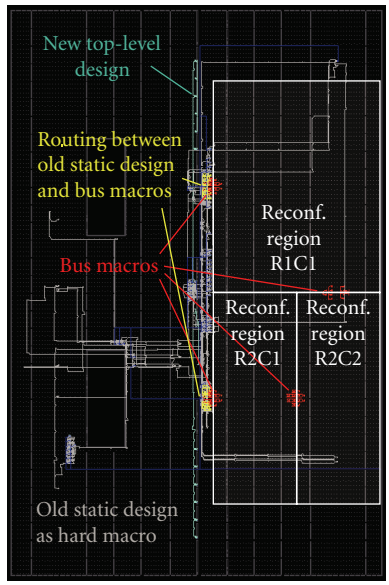


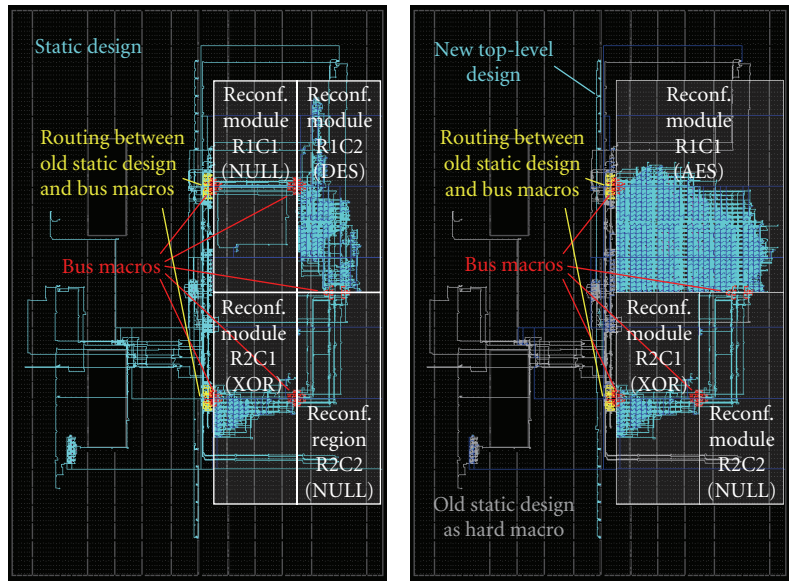
Figure 7: Static design of S_2 with static design from S_1 instantiated as a hard macro.

bus macros, potentially through multiple PR regions. Under the tool flow aspect, this approach is very clean. The insertion of additional bus macros, however, is likely to turn out even more cumbersome than creating a static hard macro as proposed in this paper. Furthermore, owing to the fact that signals are routed in PR modules, a single signal may be routed in various ways through the same PR region and, thus, show varying propagation delays. Even worse, such a signal can be interrupted during reconfiguration. Most important is, however, that it can hardly be guaranteed that really no static routing at all intersects PR regions. Consequently, this approach is not safe to rely on.

4.3.2. Directed Placement and Routing of the Static Design. Instead of using a hard macro, the static design could also be exactly reproduced for different scenarios by utilising "directed routing". Following this approach, the static design would also be opened in the FPGA Editor. Complete routing and placement information would then have to be exported in a user constraint file (UCF). This UCF file would be included into subsequent static design implementation runs. The handicaps of this approach are that potentially very large text files have to be handled and that it is necessary to reimplement the whole static design for every scenario.

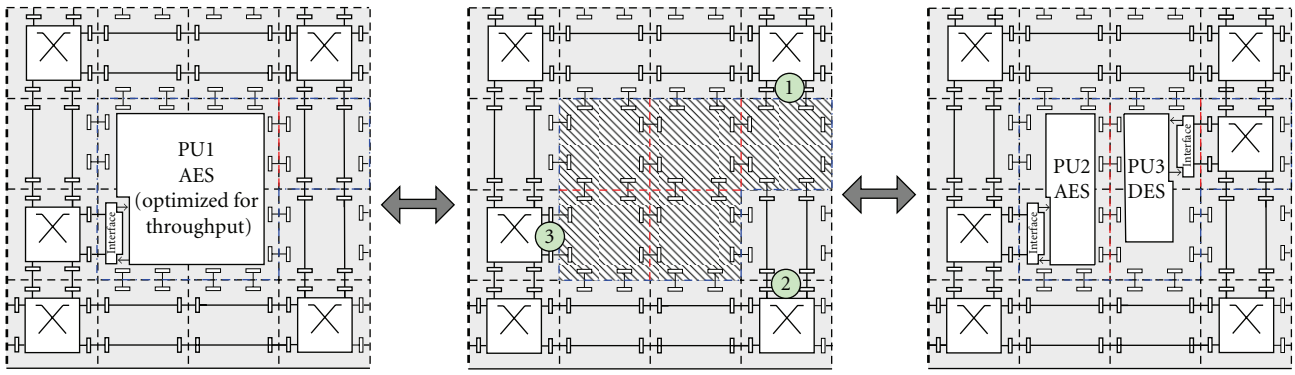
5. Application Example

The design technique was developed in the context of the system architecture presented in Section 3. This architecture will now be used to demonstrate the usefulness of this technique by means of an application oriented example, a coprocessor platform for network processors called Dynamically adaptable Coprocessor based on Reconfiguration (DynaCORE) [18].



(a) scenario S_1 (b) scenario S_2

Figure 8: Complete designs (PR modules merged in static design).



(a) Scenario S_1: one AES module (b) Intermediate step: blank tiles (c) Scenario S_2: two small encryption cores

Figure 9: Application example for changing boundaries of PU regions in the context of DynaCORE.

DynaCORE is an adaptable hardware accelerator for increasing the performance of network processors when deep packet processing is required. Typical deep packet processing tasks comprise compression, network-intrusion detection, or encryption/decryption, for example, when virtual private networks are to be established. As network processors are primarily optimised for header processing, the increasing demand in deep packet processing tasks asks for dedicated accelerators. These accelerators have to be flexible as network traffic composition and processing requirements vary throughout the day. DynaCORE provides this flexibility by autonomously determining an optimal set and arrangement of PUs according to the actual traffic profile.

A typical scenario for a dynamic exchange of PUs of different sizes within DynaCORE is depicted in Figure 9. Figure 9(a) shows the mapping of a fast and hardware intensive AES core onto four tiles. These four tiles are combined to one PR region and only provide bus macros at the boundary of the region. The complete logic area inside the PR region is used to implement the AES core. Compared to a partitioning of the AES core into four separate components to be mapped onto four individual PR regions, the problem of partitioning the core into components with only few links between them is avoided, no bus macros are required to link the components among each other, and the cross-tile routing resources of the four tiles can be used for routing. Altogether, this allows more compact PU designs than splitting one PU onto several individual PR regions.

In case DynaCORE detects a significant change in traffic profile, for example, that a considerable amount of data flows require DES encryption which cannot be handled anymore by a software instance, but less data require AES encryption, the throughput optimised AES core of Figure 9(a) may be replaced by two smaller cores, one for AES and one for

DES. This is illustrated in Figure 9(c). The four tiles are grouped into two independent PR regions consisting of two tiles, respectively. As in the initial scenario only one link for connecting one PU mapped onto the four tiles is provided, the communication infrastructure has to be adapted when changing the PU composition as well. In Figure 9 this is illustrated by replacing the vertical communication tile in the upper right corner with a switch tile.

As mentioned in Section 4.2, the adjustment of PR regions should not be done in the same step as mapping PUs onto the PR region. The required intermediate step is shown in Figure 9(b). Each tile is configured as an individual PR region with bus macros at each border. This minimises the risk of short circuits when reconfiguring one tile with neighbour tiles still holding configuration code with nets originally routed to the tile under reconfiguration. Special precautions also have to be taken for the communication infrastructure when elements have links to the PR regions under reconfiguration. In Figure 9(b), these elements are marked with numbers. As interfering signals arise at links to regions under reconfiguration, the corresponding switch ports have to be disabled. Special NoC messages are used to activate/deactivate switch ports. A detailed description of this mechanism is given in [14]. When switching back from two small PUs to one large PU, the steps shown in Figure 9 have to be done in reverse order.

6. Summary

The presented design technique enables the adjustment of the boundaries and number of PR regions in a system at runtime. No online manipulating of configuration data or adapted versions of design tools are necessary. The implemented system does not require additional FPGA logic resources to provide this feature but the link to the ICAP component. The complexity to provide a system with such features is completely moved to design time. The practicability of the design technique is shown by means of two examples. The first example explains the single steps in detail on a low abstraction level. On application level, the second example demonstrates the advantages a system can achieve when using a system architecture designed for this technique. To conclude with, the proposed technique opens the way for partially reconfigurable designs which use PR modules of varying sizes with standard design tools.

Acknowledgment

This work was funded in part by the German Research Foundation (DFG) within priority programme 1148 under Grant reference Ma 1412/5.

References

[1] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 12–17, Madrid, Spain, August 2006.

[2] "PlanAhead the Fastest Route to Better Design," Data sheet, 2005.

[3] T. Pionteck, C. Albrecht, R. Koch, and E. Maehle, "Adaptive communication architectures for runtime reconfigurable system-on-chips," *Parallel Processing Letters*, vol. 18, no. 2, pp. 275–289, 2008.

[4] F. Dittmann, "Algorithmic skeletons for the programming of reconfigurable systems," in *Proceedings of the 5th IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS '07)*, R. Obermaisser, Y. Nah, P. P. Puschner, and F.-J. Rammig, Eds., vol. 4761 of *Lecture Notes in Computer Science*, pp. 358–367, Springer, Santorini Island, Greece, May 2007.

[5] F. Dittmann, S. Frank, and S. Oberthür, "Algorithmic skeletons for the design of partially reconfigurable systems," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, pp. 1–8, Miami, Fla, USA, April 2008.

[6] M. Cole, *Structured Management of Parallel Computing*, Pitman/The MIT Press, Boston, Mass, USA, 1989.

[7] M. Hübner, C. Schuck, and J. Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs," in *Proceedings of the 13th Workshop on Reconfigurable Architectures (RAW '06)*, pp. 1–8, Rhodes, Greece, April 2006.

[8] C. Schuck, M. Kühnle, M. Hübner, and J. Becker, "A framework for dynamic 2D placement on FPGAs," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, pp. 1–7, Miami, Fla, USA, April 2008.

[9] S. Guccione, D. Levi, and P. Sundararajan, "Bits: Java based interface for reconfigurable computing," in *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD '99)*, 1999.

[10] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "A design methodology for communication infrastructures on partially reconfigurable FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 331–338, Amsterdam, The Netherlands, August 2007.

[11] S. Koh and O. Diessel, "COMMA: a communications methodology for dynamic module reconfiguration in FPGAs," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 273–274, Napa, Calif, USA, April 2006.

[12] S. Koh and O. Diessel, "Communications infrastructure generation for modular FPGA reconfiguration," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '06)*, pp. 321–324, Bangkok, Thailand, December 2006.

[13] E. L. Horta, J. W. Lockwood, D. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," in *Proceedings of the 39th Design Automation Conference (DAC '02)*, pp. 343–348, New Orleans, La, USA, June 2002.

[14] T. Pionteck, R. Koch, and C. Albrecht, "Applying partial reconfiguration to networks-on-chips," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 155–160, Madrid, Spain, August 2006.

[15] Xilinx, Inc., "Two Flows for Partial Reconfiguration: Module Based or Difference Based," Xilinx Application Note 290, Version 1.2, March 2005.

[16] Xilinx, Inc., "Answer Record #25018—Partial Reconfiguration—PlanAhead Flow FAQ/Know Issues for the Early Access," Partial PlanAhead Program, May 2008.
[17] Xilinx, Inc., "Early Access Partial Reconfiguration User Guide," UG208, Version 1.2, September 2008.
[18] C. Albrecht, R. Koch, and T. Pionteck, "On the design of a loosely-coupled run-time reconfigurable network coprocessor," in *Proceedings of the 7th Workshop on Media and Streaming Processors (MSP '05)*, November 2005.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 395018, 11 pages
doi:10.1155/2009/395018

Research Article

A Taxonomy of Reconfigurable Single-/Multiprocessor Systems-on-Chip

Diana Göhringer,[1] Thomas Perschke,[1] Michael Hübner,[2] and Jürgen Becker[2]

[1] *FGAN-FOM, Research Institute for Optronics and Pattern Recognition, 76275 Ettlingen, Germany*
[2] *Fakultät für Elektrotechnik und Informationstechnik, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung (ITIV), 76131 Karlsruhe, Germany*

Correspondence should be addressed to Diana Göhringer, dgoehringer@fom.fgan.de

Received 19 December 2008; Accepted 12 May 2009

Recommended by Gilles Sassatelli

Runtime adaptivity of hardware in processor architectures is a novel trend, which is under investigation in a variety of research labs all over the world. The runtime exchange of modules, implemented on a reconfigurable hardware, affects the instruction flow (e.g., in reconfigurable instruction set processors) or the data flow, which has a strong impact on the performance of an application. Furthermore, the choice of a certain processor architecture related to the class of target applications is a crucial point in application development. A simple example is the domain of high-performance computing applications found in meteorology or high-energy physics, where vector processors are the optimal choice. A classification scheme for computer systems was provided in 1966 by Flynn where single/multiple data and instruction streams were combined to four types of architectures. This classification is now used as a foundation for an extended classification scheme including runtime adaptivity as further degree of freedom for processor architecture design. The developed scheme is validated by a multiprocessor system implemented on reconfigurable hardware as well as by a classification of existing static and reconfigurable processor systems.

1. Introduction

The usage of Multiprocessor System-on-Chip (MPSoC) for accelerating performance intensive applications is an upcoming trend in current chip technology. The approach of distributing tasks of an application to several processing units is a well-established method for increasing computational performance in computer science. In 1958, the IBM researchers Cocke and Slotnick discussed the usage of parallelism in numerical calculations for the first time. Four years later the first approach with four discrete parallel working processors is documented in [1]. In this first multiprocessor system, discrete chips were connected on a printed circuit board in a certain topology. Nowadays the System-on-Chip approach allows for integration of multiple devices on one chip die. Many difficulties in hardware technology were resolved. However, achieving a well-balanced workload using an optimized partitioning of the application tasks to the different processor cores remains an unsolved challenge. The partitioning algorithms and methodologies are able to detect inherent parallelism in a dataflow graph (DFG), and a mapping tool is able to distribute the tasks to a certain processing unit. The gap in this procedure is that the tasks can only be optimized to a certain extent to a given (multi)processor architecture. A certain granularity or requirement of a task cannot be further optimized. Exactly this is the problem whereof static multiprocessor hardware architecture suffers from. Changing applications and requirements do not reach an optimized work balance of the processors and the communication infrastructure. A promising approach for bridging this gap is to exploit runtime reconfigurable hardware. The requirements of an application's task can be fulfilled by a flexible hardware, which adapts in terms of computational performance and communication bandwidth.

Therefore, several academic labs all over the world are investigating the use of reconfigurable hardware in single- and multiprocessor systems-on-chip (MPSoC) (e.g., [2–4]). The hardware adaptivity offers a greater flexibility, which is not only beneficial in the global view of an MPSoC, but

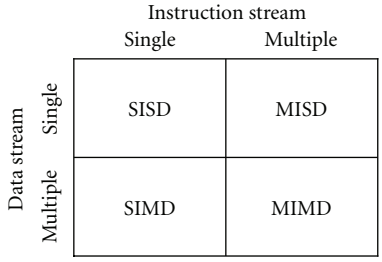


Figure 1: Flynn's taxonomy (1966) [7].

also in the special case of a single processor. This way, a static single processor optimized for a special application (Application Specific Instruction Set Processor (ASIP)) can be transformed into a RISP (Reconfigurable Instruction Set Processor) allowing for optimizations at runtime for a range of applications. Many of the novel reconfigurable MPSoCs consist of several RISPs (e.g., [5, 6]).

The most used classification scheme until today for single-/multiprocessor systems is Flynn's taxonomy [7], which uses four classes separating them in respect to single-/multiple data and instruction streams. Based on this classification scheme a new taxonomy is proposed, where runtime reconfiguration is included to further separate between static and dynamic processor systems, but also to discriminate between the different kinds of runtime reconfigurability (data and instructions).

This paper describes the new taxonomy and validates it by classifying several existing static and dynamic single- and multiprocessor systems. Furthermore, it is validated by a runtime adaptive multiprocessor system-on-chip (RAMPSoC) [6].

The paper is organized as follows: the state of the art will be presented in Section 2. Section 3 describes the new taxonomy for reconfigurable single-/multiprocessor systems-on-chip. Examples for each class of the new taxonomy are given in Section 4. Section 5 describes the RAMPSoC architecture, which forms the superclass of the new taxonomy. Finally the paper is closed by presenting the conclusions and future work in Section 6.

2. State of the Art

Until today, the most known classification scheme for single- and multiprocessor systems is the taxonomy of Flynn introduced in 1966, which is shown in Figure 1.

This taxonomy groups processor systems into four classes depending on their support for single/multiple data and/or instructions streams. At the time, when Flynn developed his taxonomy, only static processor systems existed. Due to their static architecture, they force the user to follow a top-down designflow, in which the application integration has to follow the given hardware architecture. Especially for multiprocessor systems this is a major drawback, because a given application can only be optimized to a certain extend to the static hardware architecture. As the processors and their communication infrastructure are fixed only few applications, for which the architecture was optimized, achieve a well balanced workload. This drawback exists not only at design-time, but also at runtime, as the hardware cannot be adapted on-demand in respect to the application requirements.

Looking at state-of-the-art processor architectures a novel trend toward using runtime reconfigurable hardware for single- and multiprocessor systems can be observed [8]. The systems using runtime reconfigurability can overcome the major drawback of static processor systems, because they can be adapted to the requirements of the application. As these adaptive systems differ in respect to their degree of reconfigurability, they have to be classified. The separation within Flynn's taxonomy is very coarse and does not account for the new degree of flexibility and adaptivity of these novel hardware systems.

There exist previous approaches to classify reconfigurable systems. Sanchez et al. [9] proposed a very coarse classification by dividing the systems into static and dynamic configurable systems. Following their terminology, static configurable systems are configured by an external processor once before executing a task. Dynamic configurable systems can configure themselves to adapt to a changing environment. This classification approach is too coarse for classifying state-of-the-art single- and multiprocessor systems. Also only FPGA-based architectures are taken into account.

Page [10] proposes possible categories, which can be used to develop a taxonomy for reconfigurable processor architectures. These reconfigurable processor architectures are nowadays called RISPs and consist of a single processor and a reconfigurable array. Therefore, his classification is very useful for such architectures. It is also very fine grained, as it classifies these systems depending on their interconnection, on the used memory architecture and memory operation and also on the models used for programming the reconfigurable array. A major drawback is that it does not take into account reconfigurable MPSoC systems or static single- and multiprocessor systems.

Also, Hartenstein [11] describes a classification for RISPs processors, but he only takes into account coarse-grained reconfigurable architectures. He classifies these architectures according to their architecture (mesh-, linear array-, or crossbar-based). Each architecture class is further divided depending on the used granularity, fabrics, mapping, and intended target applications. This results in the same major drawback than described above for [10].

Additionally, Sima et al. [12] describe a taxonomy to classify RISPs, also named Field-Programmable Custom Computing Machines (FCCMs). In contrary to [10, 11], they classify these systems only on architectural criteria and not on implementation details. This results in a coarser taxonomy consisting of four classes. Also here the major drawback is that this taxonomy does not take into account reconfigurable MPSoC systems or static single- and multiprocessor systems.

Radunovic and Milutinovic [13] propose the "Olymp" classification for reconfigurable systems. In a first step, they divide these systems depending on their aim of reconfiguration (increasing performance or fault tolerance). Further categories are the granularity of the reconfigurable

device (fine-, medium, or coarse-grained), the integration (dynamic, static closely coupled, static loosely coupled), and, finally, if the external network is static or reconfigurable. For the integration category, they use the same separation as proposed in [9]. This means, dynamic systems are stand-alone systems, and static systems consist of a host processor that manages the configuration of the reconfigurable array. Out of this, 15 classes are generated. Each of them is named after a Greek god. For six of these classes, examples of existing systems are given. These systems are classified according to their dominating characteristic. The advantage of this classification is that it supports a fine-grained classification for reconfigurable computing systems, which use reconfigurability to achieve a better performance. A disadvantage is that all reconfigurable systems designed for fault-tolerance reasons are grouped together and are not finer classified. An additional drawback is that there is no separation made between static and reconfigurable systems, and also there is no separation between single and multiple data and/or instruction streams and therefore no separation between single- and multiprocessor systems is done by this classification. Finally, using names of Greek gods in such a classification makes it also difficult to remember, which Greek god resembles which class of systems. The naming convention used by the taxonomy of Flynn is straighter forward.

In summary, the major drawback of all these classification approaches is that none of them classify static and reconfigurable single- and multiprocessor systems. Therefore, a new taxonomy, supporting a finer classification of static and dynamic single- and multiprocessor systems, is required and will be described in detail in the next section.

3. The Taxonomy Extension to Flynn

Nowadays, while more and more different implementations of reconfigurable processors and reconfigurable MPSoCs are introduced, Flynn's taxonomy is not sufficient anymore, as it does not differentiate between reconfigurable and static processor systems. Also, as mentioned before, a finer separation with respect to the type of reconfigurability is required. Therefore a new taxonomy for the classification of reconfigurable Single-/Multiprocessor Systems-on-Chip is necessary. These reconfigurable systems differ from each other not only with respect to single or multiple data and instruction streams, but also with respect to the grade of reconfigurability, which they support. This results in the new taxonomy shown in Figure 2.

The form of the classification scheme is based on a Karnaugh diagram with four variables. First, the systems are divided into single (SI) and multiple instruction (MI) stream systems. Second, they are divided further into single (SD) and multiple data (MD) stream systems. These divisions were adopted from Flynn. Furthermore, the systems are separated in static and reconfigurable instruction (RI) stream systems. Reconfigurable instruction stream systems can exchange either their instruction memory or their instruction set, parts of which can also be implemented in

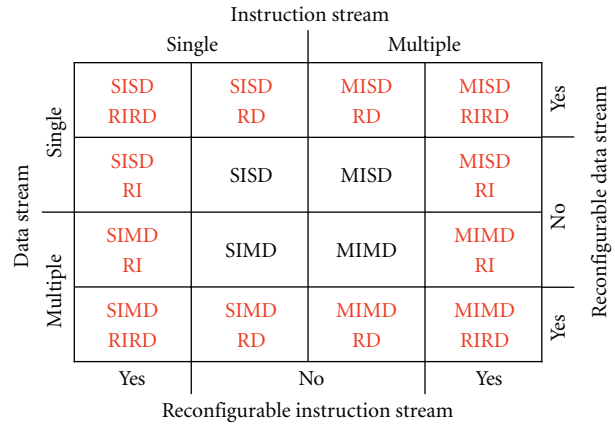


Figure 2: The new taxonomy.

reconfigurable hardware accelerators. Finally, the systems are divided with respect to static and reconfigurable data (RD) streams. Processor systems with reconfigurable data streams can either exchange their data memory or modify the data paths or both. As can be seen in Figure 2, the four classes in the middle define the static processor systems known from Flynn's taxonomy. The reconfigurable classes are grouped around them.

For this taxonomy the following terminology is used: a system with an additional degree of freedom means that this system offers one additional degree of parallelism and therefore has a greater flexibility. The parallelism of a system is divided into spatial and temporal parallelism. Spatial parallelism is used to distinguish between systems with single or multiple streams. Here temporal parallelism is used to differentiate static and reconfigurable systems. Reconfigurable systems have one or two degrees of temporal parallelism, depending, if only one or both, instruction and data streams are reconfigurable. Static systems, therefore, do not have a temporal parallelism following to this terminology. Of course, from the view of a computer architect, processor system with pipelined architectures, like vector processors, also have a temporal parallelism. In this terminology this kind of temporal parallelism is low level compared to the temporal parallelism offered by reconfigurability. Therefore, it will not be considered.

Using this terminology a hierarchy between the classes exists depending on their degree of parallelism. This way, some classes are a subclass of others. This hierarchy can be seen in Figure 3. Classes on the same level offer the same degree of freedom. The difference between each of the 5 levels is one degree of freedom.

This means that SISD is the lowest class by offering the lowest flexibility. In other words, each of the other classes can be reduced down to SISD by removing its spatial and/or temporal parallelism. By adding one degree of parallelism to the SISD architecture, either in respect to supporting multiple data or instruction streams, or by adding the reconfigurability of either the instructions or the data, the next level is reached. Therefore, this level consists of the following classes: SIMD, SISD_RI, SISD_RD, and MISD. By

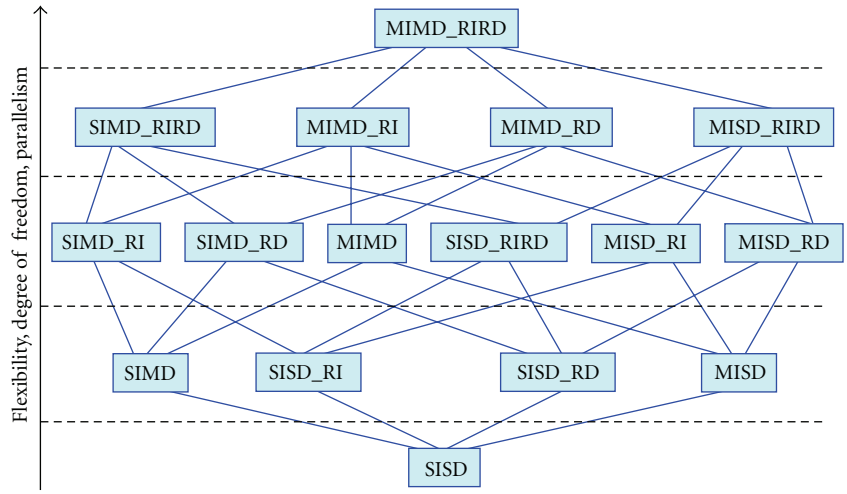


Figure 3: Hierarchy of the classes depending on their degree of freedom, their flexibility, and parallelism. The neighborhood relations are shown by the connections between the classes.

adding more flexibility the next level is reached and so on until finally the highest level is reached. In this level only the MIMD_RIRD class is present. This class supports the highest degree of freedom and therefore is the superclass of the whole taxonomy. MIMD_RIRD resembles a reconfigurable multiple instruction and reconfigurable multiple data stream architecture. To the best of our knowledge, so far only one system combines this flexibility. This system is a runtime adaptive MPSoC called RAMPSoC, which will be presented in Section 5.

4. Examples for the Different Classes

In the following subsections each class will be described. Several representative systems will be referenced for each class. Of course, for most classes more systems exist than can be mentioned here.

4.1. SISD. SISD processors are usually based on von Neumann or Harvard architectures. Examples are processors based on older x86 architectures, like the Intel 80486 processors. Most microcontrollers used in embedded systems fall into this category. Examples are the PowerPC405 cores [14] used on Xilinx Virtex-II Pro and Virtex-4 FPGAs and soft-core processors like Xilinx MicroBlaze [15] or Leon [16].

4.2. MISD. Systems with this architecture are very rare. In [7] a pipelined structure is presented as a MISD architecture. There, a number of separate execution units with their own instruction code act on the data processed by the preceding execution unit. Examples for such a scheme are streaming architectures for image processing used on FPGAs. It is also possible to include redundant computer systems like the one used in the space shuttles [17] in this category. Additionally, MISD architectures are used for pattern matching [18], because they support to simultaneously search for multiple patterns within a single data stream.

4.3. SIMD. This class can be further divided in array and vector processor-based classes. Generally speaking, array processors use parallelization in space while vector processors use low level parallelization in time to gain processing power.

Array processors consist of an array of regularly connected identical execution units. At each cycle all units execute the same instruction on different data. This architecture was used in older systems, like the Connection machine CM-2. Systolic arrays can be included in this category. Array processors are also used in embedded systems to increase the performance compared with SISD architectures. Especially, in the image processing domain (e.g., [19–21]) they are used to increase the overall performance, for example, for matrix multiplications.

Vector processors, for example the NEC SX-6 [22] processor, use vector pipelines on vectors of floating point numbers. The execution unit is deeply pipelined. Vector pipelines with the same or different functionality can be chained to form vector units. Vector processors also include scalar units.

4.4. MIMD. The MIMD class is usually divided in different subclasses, as too many different architectures fall into this category.

Memory coupled processor systems, like multicore CPUs, can be grouped depending on the type of memory access in UMA (uniform memory access), NUMA (non-uniform memory access), and COMA (cache only memory access) systems.

Processor systems based on message passing can be grouped in MPP (massive parallel processing) systems, like the Intel ASCI-Red [23] and COW (cluster of workstations).

Superscalar processors, like the recent x86 architectures, VLIW (Very Large Instruction Word) architectures mostly used in DSP processors and the similar EPIC (Explicitly parallel instruction computing) architectures used in the Intel Itanium [24] processors can be included in the MIMD class.

An additional example for the MIMD category is the Cell processor [25], which consists of one power processor element (PPE) and eight synergetic processor elements (SPEs). While each single PPE or SPE has an SIMD architecture, the overall system belongs to the MIMD architecture.

4.5. SISD_RI. SISD_RI is similar to SISD with a parallelism in the time domain. This means, that the instruction stream can be reconfigured at runtime. The best known systems that fall into this class are Reconfigurable Instruction Set Processors (RISPs) [26]. These processors can adapt their instruction set on-demand. They consist of a processor with an attached reconfigurable block. Today, lots of research are done in this field, and there exist several different types of RISPs (such as [2, 3, 27]). They differ from each other with respect to their processor architecture, the reconfigurable block (coarse, fine grained), how the reconfigurable block is connected to the processor (coprocessor, tightly-coupled or loosely coupled on-processor), the reconfiguration method (partial reconfiguration, context switch ...), and so forth.

4.6. SISD_RD. These systems have a parallelism in the time domain. The difference to SISD_RI systems is that here the instruction stream remains static, but the data stream is reconfigurable. Such systems are SISD processors which can exchange their data memory or the data path to connect to a different data source or both. The data memory can be exchanged by using dynamic and partial reconfiguration as it is done in [28, 29]. There also exist reconfigurable communication infrastructures [30], which can be used to adapt the data path at runtime.

4.7. SISD_RIRD. This is a combination of the SISD_RI and the SISD_RD systems described before. Here, two degrees of flexibility are added, as now the instruction stream as well as the data stream can be reconfigured at runtime. A representative system for this class is one single processor in RAMPSoC [6], because the exchange of both instruction and data memory is supported, and a reconfigurable communication structure is used between the processing elements as well as with the environment.

4.8. SIMD_RI. These systems are basically SIMD systems with an extended parallelism in the time domain. These systems can reconfigure their instruction stream at runtime, by reconfiguring, for example, their processing elements/ hardware accelerators or their instruction memory. A representative system is the Montium Tile Processor from Recore Systems [31], which has a reconfigurable instruction set. Also, systems consisting of a processor with a loosely coupled reconfigurable hardware accelerating the instructions of the processor belong to the SIMD_RI architecture, if both can access independently the data memory and if the data path outside the reconfigurable hardware remains static. In this case the reconfigurable hardware is an accelerator and cannot work without the processor. Therefore, from the abstract system view, only a processor with an attached accelerator is seen. Reconfigurable data paths or processing elements within the accelerator itself are not taken into account when classifying these systems. Such a system is, for example, MorphoSys [32]. Also, the MORPHEUS platform [33] consisting of one ARM9 processor connected to three heterogeneous reconfigurable accelerators belongs to this class. In [34] three additional systems falling into this category are described: DReAM (a coarse-grain Dynamically Reconfigurable Architecture for Mobile communication systems), Triscend A7, and XPP (eXtreme Processing Platform).

4.9. SIMD_RD. These systems are SIMD which can exchange their data memory or reconfigure their communication infrastructure [35].

4.10. SIMD_RIRD. SIMD_RIRD systems combine the two different kinds of parallelisms in the time domain of the SIMD_RI and SIMD_RD systems. Therefore, these systems can configure both their data stream and their instruction stream. An example of such a system would be a single processor in RAMPSoC with several reconfigurable accelerators that compute the same instruction on different data streams. Additional to the accelerators also the data and instruction memory as well as the communication infrastructure support runtime reconfiguration. Furthermore, the Maia architecture [34], consisting of an ARM processor connected over a reconfigurable network to an FPGA accelerator as well as several embedded memories, MACs and ALUs, belongs to this class. In this system the network (data stream) and the FPGA accelerator (instruction stream) are reconfigurable. A similar system is the Pleiades architecture [36] consisting of a microprocessor, which is connected over a reconfigurable interconnect to several reconfigurable accelerators called satellites.

4.11. MISD_RI. These systems extend the static MISD architecture by having multiple reconfigurable instruction streams. These could be redundant systems consisting of several processors that can reconfigure their instruction memories or their processing elements/hardware accelerators.

An additional example would be a group of SISD_RI processors, which compute multitarget image processing algorithms, where each processor is searching for a different object in the same data. An additional example would be a streaming architecture for image processing with reconfigurable instruction streams.

4.12. MISD_RD. These systems are MISD systems with an additional parallelism in the time domain. As opposed to the MISD_RI systems here the instruction stream is static, but the data stream is reconfigurable. These systems support the runtime exchange of the data memory or the communication infrastructure. An example for a MISD_RD system is a group of SISD_RD processors, which are connected to the same data source.

4.13. MISD_RIRD. These are the most flexible of the MISD systems because they have reconfigurable instruction and data streams.

4.14. MIMD_RI. MIMD_RI systems are MPSoCs with reconfigurable instruction streams. Several research labs investigate this kind of systems. For example, Paulsson et al. [37] presented a system, which supports the reconfiguration of the instruction memories. Furthermore, Claus et al. [5] and Bobda et al. [4] developed MPSoCs with reconfigurable accelerators. Also, the XiRisc reconfigurable processor [38] is a representative system for this class. It consists of a VLIW RISC core with a runtime reconfigurable data path, called PiCoGA (Pipelined Configurable Gate-Array). By reconfiguring the PiCoGA, which is within the data path of the processor; the instruction set of the processor and therefore the instruction stream are reconfigurable. An additional example for such a VLIW processor with a runtime reconfigurable data path is the ADRES architecture [39].

4.15. MIMD_RD. These systems are MPSoCs with reconfigurable data streams. This means, they have either reconfigurable data memories or a reconfigurable communication infrastructure or both by using similar functionalities as described in the SISD_RD class.

4.16. MIMD_RIRD. RAMPSoC [6] includes all the above classes. To the best of our knowledge, this is the only architecture supporting such flexibility. Section 5 describes the advantages over static MPSoCs and the new designflow methodology used for RAMPSoC. In addition, the hardware system architecture is described in detail. Finally, different hierarchy layers are defined for the RAMPSoC hardware architecture, the software toolchain, and also the runtime reconfigurability.

5. RAMPSoC—the Superclass of the New Taxonomy

RAMPSoC represents the MIMD_RIRD class and is therefore the superclass of all classes within the new taxonomy. This means, it supports the highest degree of flexibility. So, by reducing this flexibility it can be reduced down to the other classes. For example, by removing the reconfigurability, RAMPSoC can be seen as an MIMD architecture. By further reducing the instruction stream to a single stream, it resembles a SIMD structure. Finally, by reducing the data stream from multiple to single, we get a SISD architecture, which resembles one single processor in RAMPSoC without the reconfiguration capabilities.

RAMPSoC was developed to overcome the deficiencies of static MIMD approaches. The drawbacks of static MIMD architectures are that they are not optimized for the tasks they have to execute. Therefore, they are not efficient with respect to performance and power consumption, as for most applications the workload between the processors is not well balanced. Furthermore, due to their high-power consumption, they are often not well suited for embedded systems.

Certainly, there also exist static MIMD architectures, which are optimized for a limited field of applications within an application domain. For these applications they achieve a very good performance in combination with low-power consumption. Their disadvantage is that they are only efficient for the applications, for which they are optimized. They cannot efficiently be reused for a different application domain, nor is their workload efficiently balanced for all applications of their application domain.

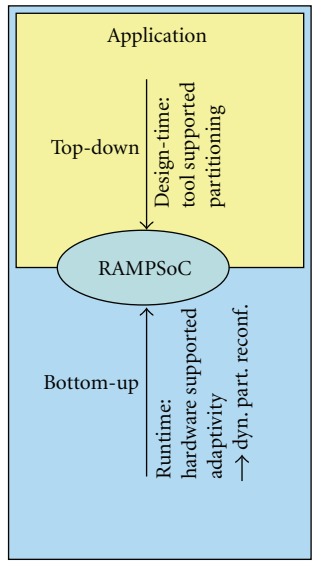


Figure 4: "Meet in the middle" Designflow approach for RAMPSoC.

This leads to the main drawback of all MIMD architectures, which is their top-down design flow. This means, the application integration has to follow the given hardware architecture of the MPSoC. Therefore, a given application can only be optimized to a certain extend to a static MPSoC architecture. This drawback exists not only at design time but also at runtime as the hardware cannot be adapted on-demand in respect to the application requirements.

Therefore, runtime reconfigurable hardware is used to overcome these deficiencies of static MIMD systems. The exploitation of runtime reconfiguration offers a new degree of freedom, because the hardware architecture can now be adapted to the application requirements. This adaptation is possible at design-time as well as at runtime allowing an optimized distribution of computing tasks. Therefore, constraints regarding the performance, the power consumption, or the used area can be achieved more efficiently. By combining the standard top-down designflow from static MIMD systems with the new bottom-up designflow the novel "meet in the middle" designflow for RAMPSoC is created. This designflow is illustrated in Figure 4. From the application perspective it offers the standard top-down designflow in terms of application partitioning and mapping. At the same time it supports from the hardware perspective the new bottom-up approach, enabling hardware adaptation at design-time as well as at runtime using dynamic and partial reconfiguration. At design-time the bottom-up approach is used to generate an initial RAMPSoC hardware architecture optimized for the actual application requirements. Additionally, the bottom-up approach supports the on-demand adaptation of the RAMPSoC system to the time variant application requirements.

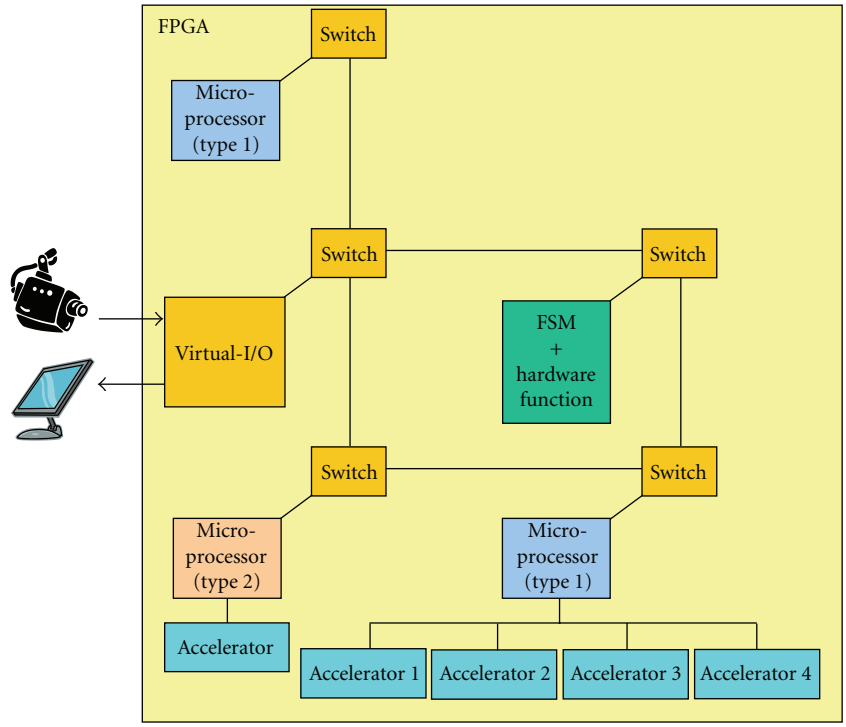


Figure 5: RAMPSoC system at one point in time.

Exactly this feature improves furthermore the functional density as discussed in [40, 41]. In contrast to the traditional and well-established methodologies of the Hardware/Software-Codesign (see [42]), the "meet in the middle" approach includes the option of a static or dynamic realization of hardware system parts. It therefore contains the "static/dynamic"-Codesign as a subpartition option of the hardware distribution. This option supports the development process at design phase, where a flexible hardware realization, based on elements from a hardware library, helps to improve the efficiency of the technology mapping process. At runtime the "meet in the middle" approach is used to provide the hardware elements on-demand. This is physically realized through dynamic and partial reconfiguration. In general, the "meet in the middle" approach targets to improve furthermore the benefits of the Hardware/Software-Codesign by the introduction of new design-time and runtime flexibility through the exploitation of reconfigurable hardware.

High performance within RAMPSoC can be achieved by exploiting the following three kinds of concurrency, which are inherently included in most applications. First, data parallelism is supported by distributing the data onto different processors, which execute the same (SIMD) or different (MIMD) instructions on their subset of data. Especially image processing algorithms lend themselves very well to data parallelism as an input image can be divided into independent tiles. Second, task parallelism is supported, as different processors or processor groups can execute different algorithms on the same (MISD) or different (MIMD) set of data. For example, this can be used in Multitarget Tracking algorithms, where each processor searches for a different target. And finally, instruction level parallelism is exploited by using processors with an instruction pipeline and by outsourcing complex instructions into one or several hardware accelerators connected to the processor. This provides an additional speedup. Here, also the benefit of the runtime reconfiguration is exploited since not all complex instructions have to be present at the same time. Instead, the hardware accelerator can be reconfigured with the required instruction by its corresponding processor on-demand at runtime.

5.1. Hardware System Architecture. Figure 5 shows the hardware system architecture of RAMPSoC at one point in time. As can be seen, RAMPSoC represents a modular, heterogeneous runtime adaptive MPSoC.

The processing elements in RAMPSoC can be different types of processors with an arbitrary number of accelerators. The processor types differ from each other for example, with respect to their bitwidth or their performance. Instead of a processor, it is possible to use a Finite State Machine (FSM) closely coupled with a hardware function.

Additionally, different types of communication infrastructures such as Network-on-Chip (NoC), Bus, or Point-to-Point connections, or a mixture of those are supported for interprocessor communication. In the example in Figure 5 a switch-based NoC with a mesh topology is used.

For the communication between RAMPSoC and its environment, such as a camera or a monitor, the Virtual-I/O component is used. The function of this component is to split the input data and forward it to one or several different processors. Additionally, it is responsible for collecting the results coming from the processors and sending them out

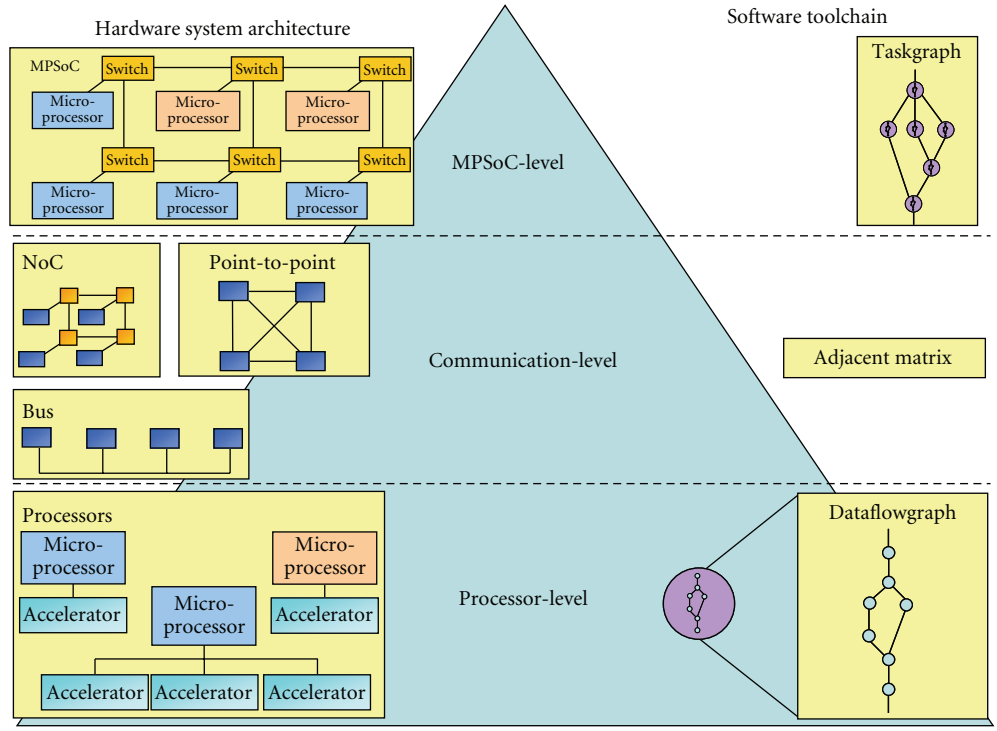


Figure 6: Three levels of hierarchy of RAMPSoC.

to the environment. The values to control the splitting of the input data as well as for the merging of the results are stored in several registers and can be modified at design-time as well as at runtime. The modification of these values can be done by overriding them either directly or by using dynamic and partial reconfiguration. Additional parts of the Virtual-I/O are two FSMs and several input and output buffers. If the number of processors in the system changes, the whole Virtual-I/O component can be modified, using dynamic partial reconfiguration. In this case, the loss of data is prevented by the data buffers.

Implementation results for three different RAMPSoC systems can be found in [43]. The systems differ from each other in respect to the number of used processors (1, 2, and 4). All use the Xilinx MicroBlaze processor, which can run at a maximum frequency of around 150 MHz. These systems were implemented to test the achievable speedup for the calculation of an image processing algorithm. Reconfiguration was not required by this algorithm, but can be easily extrapolated. As described in [43], these RAMPSoC systems were implemented on a Xilinx Virtex-4 FPGA. This FPGA family uses a 32-bit Hardware ICAP (Internal Configuration Access Port), which can run at 100 MHz. This results in a theoretical throughput of 400 Mbytes/s. Therefore, the reconfiguration of a Xilinx MicroBlaze Processor would take approximately 216 microseconds. The reconfiguration of the Virtual-IO component would require approximately 189 μs.

5.2. The Three Hierarchy-Levels of RAMPSoC. Within RAMPSoC, three virtualization layers exist forming the hierarchy-levels of RAMPSoC. These virtualization layers apply to the hardware system architecture as well as to the software toolchain as can be seen in Figure 6. Also, the runtime adaptation capabilities of the hardware using dynamic and partial reconfiguration are applied to the different layers.

The highest abstraction layer is the MPSoC-Level. Here, the exchange of whole processors is supported to fulfil quality of service parameters, such as achieving a higher performance or reducing the power consumption. Also, in this level a coarse partitioning of the application into several tasks is done. At the same time, a preliminary decision about the number and types of processors is made. Furthermore, as a result of the application partitioning the required communication infrastructure is defined.

The second abstraction layer is the Communication-Level. Here, the communication infrastructure, defined in the MPSoC-Level, is physically established. Using runtime reconfiguration the communication infrastructure can be modified by adapting the connections between the processors, for example, to change the bitwidth or the topology. Additionally, the routing algorithms can be adapted. To make this adaptation feasible and flexible the communication infrastructure is transferred to a processor-based abstraction level as shown in Figure 7.

The idea here is to see the communication infrastructure as a processor, which is distributed over the reconfigurable area. The picture shows exemplarily one possible scenario with a mixture of a NoC and a bus-based topology connecting several IP cores. The heterogeneous communication system makes sense, if, for example, a bus-based interconnect between cores is required due to a streaming data application,

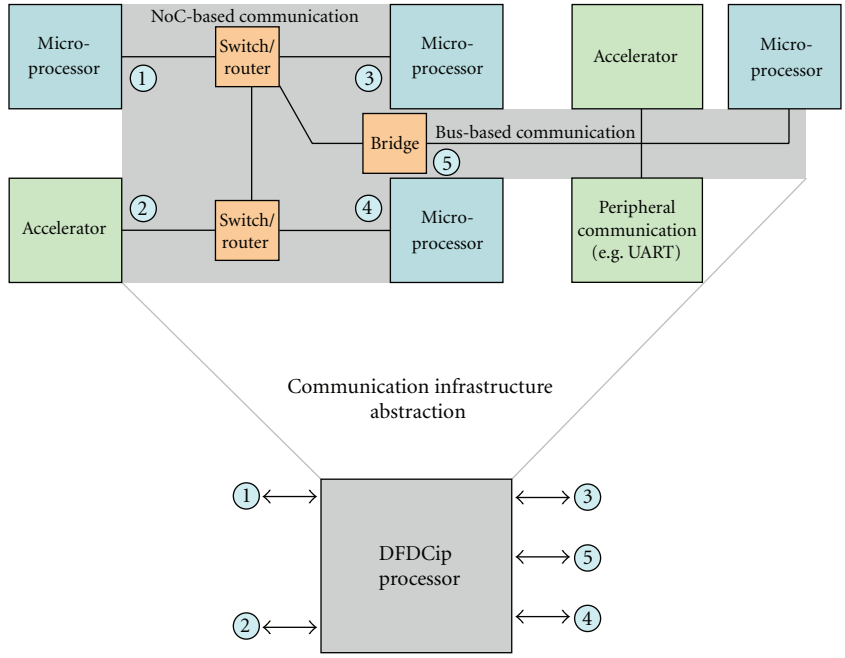


Figure 7: Communication infrastructure abstraction into a data flow dependent communication infrastructure processor (DFDCip).

which would block a NoC. Furthermore, a star or mesh topology enabled by a NoC has advantages for some applications in terms of performance and power consumption as described in [44, 45]. The idea, which is depicted in Figure 7, transforms the communication infrastructure into a module, which only has the required input and output channels connected to the respective IP cores. The bridge between the NoC and the bus system is also integrated in the module, which is called Data Flow Dependent Communication Infrastructure Processor (DFDCip). The DFDCip is a model for the communication infrastructure of the RAMPSoC approach, which can be handled as a processor providing communication interfaces on-demand. It is obvious, that this module has to be adaptive to a changing requirement of the dataflow of an application. Transferring the communication infrastructure of the MPSoC approach to a processor-based model has the advantage that a well-established programming paradigm can be used to adapt the communication hardware in relation to the dataflow of tasks running on the several IP cores. The example shows that also the DFDCip enables a new degree of freedom in terms for adaptivity by parameterization. Additionally, in case of a request for restructuring of the communication infrastructure, the components within the DFDCip block can be adapted using dynamic reconfiguration.

The third and lowest abstraction layer is the Processor-Level. Using dynamic and partial reconfiguration the instruction set of the processor can be modified by reconfiguring the accelerator. Additionally, the instruction as well as the data memory can be modified at runtime. Therefore, each processor within RAMPSoC represents a RISP and belongs to the SISD_RIRD class. Normally, RISPs only fulfil the requirements for the SISD_RI class, but the processors within RAMPSoC support the exchange of the data memory and therefore have a reconfigurable data stream, which classifies them for SISD_RIRD.

6. Conclusions and Outlook

This paper presents a new taxonomy for static and reconfigurable Single- and Multiprocessor Systems-on-Chip. This taxonomy extends the well-known taxonomy for very high-speed computers, which was proposed by Flynn in 1966. The new classification scheme is validated by referencing several representative systems for each class and extendable for novel architectures of the future. Finally, RAMPSoC is described as the representative systems for the superclass of the new taxonomy. This superclass supports the highest degree of freedom by using multiple reconfigurable instructions and data streams. The novel "meet in the middle" approach, which exploits the hardware adaptivity at design and runtime, enables to overcome the restrictions of traditional MPSoC architectures.

References

[1] A. Gary, "The Power of Parallelism," IEEE Computer World, November 2001, http://www.computerworld.com.

[2] T. Vogt and N. Wehn, "A reconfigurable application specific instruction set processor for viterbi and log-MAP decoding," in *Proceedings of IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS '06)*, pp. 142–147, Banff, Canada, October 2006.

[3] L. Bauer, M. Shaflque, and J. Henkel, "A computation- and communication-infrastructure for modular special instructions in a dynamically reconfigurable processor," in *Proceedings of the International Conference on Field Programmable*

Logic and Applications (FPL '08), pp. 203–208, Heidelberg, Germany, September 2008.

[4] C. Bobda, T. Haller, F. Mühlbauer, D. Rech, and S. Jung, "Design of adaptive multiprocessor on chip systems," in *Proceedings of the 20th Symposium on Integrated Circuits and System Design (SBCCI '07)*, pp. 177–183, Rio de Janeiro, Brazil, September 2007.

[5] C. Claus, W. Stechele, and A. Herkersdorf, "Autovision—a run-time reconfigurable MPSoC architecture for future driver assistance systems," *Information Technology Journal*, vol. 49, no. 3, pp. 181–187, 2007.

[6] D. Göhringer, M. Hübner, V. Schatz, and J. Becker, "Runtime adaptive multi-processor system-on-chip: RAMPSoC," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, Miami, Fla, USA, April 2008.

[7] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[8] "Cray XD1 Datasheet," http://www.cray.com.

[9] E. Sanchez, M. Sipper, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, and A. Perez-Uribe, "Static and dynamic configurable systems," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 556–564, 1999.

[10] I. Page, "Reconfigurable processor architectures," *Microprocessors & Microsystems*, vol. 20, pp. 185–196, 1996.

[11] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of Design, Automation and Test in Europe (DATE '01)*, pp. 642–649, Munich, Germany, March 2001.

[12] M. Sima, S. Vassiliadis, S. D. Cotofana, J. T. J. van Eijndhoven, and K. A. Vissers, "Field-programmable custom computing machines—a taxonomy," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '02)*, pp. 79–88, Montpellier, France, September 2002.

[13] B. Radunovic and V. Milutinovic, "A survey of reconfigurable computing architectures," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '98)*, pp. 376–385, Tallin, Estonia, 1998.

[14] "Xilinx PowerPC Processor Reference Guide," http://www.xilinx.com.

[15] "Xilinx MicroBlaze Reference Guide," http://www.xilinx.com.

[16] J. Gaisler, "The LEON Processor User's Manual," http://www.gaisler.com.

[17] A. Spector and D. Gifford, "The space shuttle primary computer system," *Communications of the ACM*, vol. 27, no. 9, pp. 872–900, 1984.

[18] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snove Jr., and O. R. Birkeland, "A recursive MISD architecture for pattern matching," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 7, pp. 727–734, 2004.

[19] R. L. Rosas, A. de Luca, and F. B. Santillan, "SIMD architecture for image segmentation using sobel operators implemented in FPGA technology," in *Proceedings of the 2nd International Conference on Electrical and Electronics Engineering ((ICEEE '05)*, pp. 77–80, September 2005.

[20] P. Bonnot, F. Lemonnier, G. Edelin, G. Gaillat, O. Ruch, and P. Gauget, "Definition and SIMD implementation of a multi-processing architecture approach on FPGA," in *Proceedings of Design, Automation and Test in Europe (DATE '08)*, pp. 610–615, Munich, Germany, March 2008.

[21] NEC Electronics Europe, "IMAPCAR—the solution to automotive embedded image processing," http://www.eu.necel.com.

[22] "NEC SX-6 Series," http://www.nec.co.jp/press/en/0110/0301.html.

[23] "ASCI-Red," http://www.sandia.gov/ASCI/Red.

[24] "Intel Itanium Processor 9100 Series Product Brief," http://www.intel.com.

[25] D. Pham, S. Asano, M. Bolliger, et al., "The design and implementation of a first-generation CELL processor," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC '05)*, vol. 48, pp. 184–592, San Francisco, Calif, USA, February 2005.

[26] F. Barat, R. Lauwereins, and G. Deconinck, "Reconfigurable instruction set processors from a hardware/software perspective," *IEEE Transactions on Software Engineering*, vol. 28, no. 9, pp. 847–862, 2002.

[27] A. Lodi, L. Ciccarelli, C. Mucci, R. Giansante, and A. Cappelli, "An embedded reconfigurable datapath for SoC," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 303–304, Napa Valley, Calif, USA, April 2005.

[28] M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin, and R. Fong, "Metawire: using FPGA configuration circuitry to emulate a network-on-chip," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 257–262, Heidelberg, Germany, September 2008.

[29] O. Sander, L. Braun, and J. Becker, "An exploitation of data reallocation by performing internal FPGA self-reconfiguration mechanisms," in *Proceedings of the Reconfigurable Computing: Architectures, Tools and Applications (ARC '08)*, pp. 312–317, London, UK, March 2008.

[30] L. Braun, D. Göhringer, T. Perschke, V. Schatz, M. Hübner, and J. Becker, "Adaptive real time image processing exploiting two dimensional reconfigurable architecture," *Journal of Real-Time Image Processing, Springer*, vol. 4, no. 2, pp. 109–125, 2009.

[31] L. T. Smit, G. K. Rauwerda, A. Molderink, P. T. Wolkotte, and G. J. M. Smit, "Implementation of a 2-D 8×8 IDCT on the reconfigurable Montium core," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 562–566, Amsterdam, The Netherlands, August 2007.

[32] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[33] A. Deledda, C. Mucci, A. Vitkovski, et al., "Design of a HW/SW communication infrastructure for a heterogeneous reconfigurable processor," in *Proceedings of Design, Automation and Test in Europe (DATE '08)*, pp. 1352–1357, Munich, Germany, March 2008.

[34] J. Becker, "Configurable systems-on-chip: commercial and academic approaches," in *Proceedings of the IEEE 9th International Conference on Electronics, Circuits and Systems (ICECS '02)*, vol. 2, pp. 809–812, September 2002, Dubrovnik, Croatia.

[35] H. Fatemi, B. Mesman, H. Corporaal, T. Basten, and R. Kleihorst, "RC-SIMD: reconfigurable communication SIMD architecture for image processing applications," *Journal of Embedded Computing*, vol. 2, no. 2, pp. 167–179, 2006.

[36] M. Wan, H. Zhang, V. George, et al., "Design methodology of a low-energy reconfigurable single-chip DSP system," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 28, no. 1-2, pp. 47–61, 2001.

[37] K. Paulsson, M. Hübner, H. Zou, and J. Becker, "Realization of real-time control flow oriented automotive applications

on a soft-core multiprocessor system based on Xilinx Virtex-II FPGAs," in *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC '05)*, pp. 103–110, Algarve, Portugal, February 2005.

[38] A. Cappelli, A. Lodi, C. Mucci, M. Toma, and F. Campi, "A dataflow control unit for C-to-configurable pipelines compilation flow," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)*, pp. 332–333, Napa Valley, Calif, USA, April 2004.

[39] M. Berekovic, A. Kanstein, and B. Mei, "Mapping MPEG video decoders on the ADRES reconfigurable array processor for next generation multi-mode mobile terminals," in *Proceedings of the Global Signal Processing Conferences & Expos for the Industry: TV to Mobile (GSPX '06)*, Amsterdam, The Netherlands, March 2006.

[40] J. M. Wirthlin and B. L. Hutchings, "Improving functional density using runtime circuit reconfiguration," *IEEE Transactions on VLSI*, vol. 6, no. 2, pp. 247–256, 1998.

[41] P. Benoit, G. Sassatelli, L. Torres, D. Demigny, M. Robert, and G. Cambon, "Metrics for digital signal processing architectures characterization: remanence and scalability," vol. 3133 of *Lecture Notes in Computer Science*, pp. 128–137, 2004.

[42] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.

[43] D. Göhringer, M. Hübner, T. Perschke, and J. Becker, "New dimensions for multiprocessor architectures: on demand heterogeneity, infrastructure and performance through reconfigurability: the RAMPSoC approach," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 495–498, Heidelberg, Germany, September 2008.

[44] L. Benini and G. de Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[45] P. T. Wolkotte, G. J. M. Smit, N. Kavaldjiev, J. E. Becker, and J. Becker, "Energy model of networks-on-chip and a bus," in *Proceedings of the International Symposium on System-on-Chip (SoC '05)*, pp. 82–85, Tampere, Finland, November 2005.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 162416, 12 pages
doi:10.1155/2009/162416

Research Article

Multilevel Simulation of Heterogeneous Reconfigurable Platforms

Damien Picard[1,2] and Loic Lagadec[1,2]

[1] *Université Européenne de Bretagne, 35000 Rennes, France*
[2] *CNRS, UMR 3192 Lab-STICC, ISSTB, Université de Brest, 20 avenue Le Gorgeu, 29285 Brest, France*

Correspondence should be addressed to Damien Picard, damien.picard@univ-brest.fr

Received 17 December 2008; Accepted 15 April 2009

Recommended by Gilles Sassatelli

This paper presents a general system-level simulation and testing methodology for reconfigurable System-on-Chips, starting from behavioral specifications of system activities to multilevel simulations of accelerated tasks running on the reconfigurable circuit. The system is based on a common objectoriented environment that offers valuable debugging and probing facilities as well as integrated testing features. Our system brings these benefits to the hardware simulation, while enforcing validation through characterization tests and interoperability through on-demand mainstream tools connections. This framework has been partially developed in the scope of the EU Morpheus project and is used to validate our contribution to the spatial design task.

1. Introduction

Continuous advances in the VLSI processing technology enable to produce ever more complex systems on a single chip. These System-on-Chips contain dedicated circuits, processors, memories, and also reconfigurable circuits for increasing flexibility. As a result, an application executed on a RSoC is partitioned between the heterogeneous components of the system involving concurrent activities such as memory transfers, accelerator execution, and processor activities. The heterogeneity and the concurrent nature of the platform make it hard to program and to perform functional verifications of a running application.

In order to cope with the increasing complexity of SoC design, the abstraction level of the specification has been raised [1]. The VHDL and Verilog languages, the main hardware description languages employed today, support abstraction levels up to the functional level [2, 3]. However the lack of high-level programming features makes them unsuited for developing high-level models and systems.

The verification of an application running on an RSoC has to take into account all the system activities and their impact in term of behavior and performances. For example, an application mapped on a reconfigurable accelerator is strongly dependent of the communications, managed by a DMA controller, between the local buffers and the main memory. Therefore, the execution of an application can be considered as concurrent and involves specific issues for testing and debugging such as collecting local states in order to build up back a snapshot of the whole execution state.

A higher abstraction level entry for SoC design is achieved by extending mainstream programming languages such as C/C++ or Java [4–6]. The well-known SystemC enables to describe and simulate hardware/software at system-level by using a C++ class library [7]. The particularity of the SystemC approach is that it brings to SoC design the advantages of object-oriented software development. Some other works using object-oriented concepts in various languages have been reported such as [8–11].

However, the mainstream environments used (e.g., SystemC) have restricted debugging exploration capabilities and lack of dynamic features such as hot-code replacement or object state alteration at runtime. Furthermore, despite they are oriented toward software development, engineering methodologies such as eXtreme programming remain underexploited keeping mitigated the productivity gain.

This paper focuses on the simulation, testing, and debugging of an application running on an RSoC. The

presented methodology aims at bringing to RSoC application design the agility and efficiency of software development techniques as found in object-oriented languages and eXtreme Programming (XP) methodologies [12]. The proposed framework enables to model the platform at a system level by assembling components that inherit from an object framework. The application mapped on a reconfigurable accelerator is specified at different abstraction levels and integrated as a component in the system model. The specification starts at the behavioral level as object-oriented code which is refined automatically through an intermediate representation (IR). This IR is a Control/Data Flow Graph (CDFG) encoding algorithms and is taken as input of simulators and synthesis tools [13].

The framework is developed in a Smalltalk-80 object-oriented environment which offers symbolic debugging facilities and testing features as well as tools for exploring the application at runtime giving access to instantaneous snapshots of the system state [14, 15]. Our framework brings these benefits to the hardware simulation, while enforcing validation through characterization tests. Our approach relies on a nonmainstream environment, therefore, interoperability with mainstream tools is a critical issue and is provided through automated code generation.

The article is organized as follow. Section 2 discusses software engineering methods and tools support for validation and how their concepts are applied to our framework. Section 3 details the intermediate representation of the applications supporting multilevel simulation and used for synthesis. The system-modeling framework based on components is described as well. Section 4 presents the simulation models used by the framework and the automated generation of HDL wrapper enabling interoperability with mainstream tools. Section 5 shows simulation results at multilevel for the system and the mapped applications.

2. Software Tools Support and Methods for Validation

Software engineering methodologies are well known for enhancing productivity. This section presents tools and methods used in software design for validating applications and how it is applied to our methodology.

2.1. Debugging Tools and Use. Debugging is the task of tracking down causes of program malfunction. Some subtle errors, such as the mishandling of unusual assignments to a variable, can take a lot of exploration to trace and resolve. To trace these the programmer needs a mechanism for tracing the flow of a program and variable assignments at various points.

VisualWorks Smalltalk [15] provides several facilities to help the programmer debug his programs. Software probes insert triggers into the compiled code, without changing the source code, which interrupt either processing (breakpoints) or log status information (watchpoints). A walkback window is opened when an unhandled exception is detected, showing the last several executed methods. The debugger tool allows for extensive exploration of the execution history, for modifying variable values, and modifying code on the fly, and for controlling program execution.

```
test1
    self shouldnt: [CDFGSynthesisAPI example1]
        raise: TestResult error
```

Algorithm 1: Validating an example. The method test1 is defined in a class inheriting from the SUnit framework. It provides a set of methods that detect all conditions that return false while handling unexpected errors.

To diagnose a problem, sometimes it is sufficient to see the last few entries in the context stack. The Debugger's top view lists as much of the stack as the programmer wants to see. Then, one would like to continue execution in the debugger for the next iteration for some iterator construct. The software debugger provides these capabilities.

Software probes provide a way to check the state of the system at a specific point. A software probe does not change the source code design but will affect the timing of the program execution. Similarly, using an electronic probe does not change the design of an electronic circuit, but, when used, it may change the circuit's characteristics slightly.

A breakpoint, which is the simplest kind of probe, immediately opens the system debugger, skipping the notifier stage, when it is triggered. The top method in the stack is the method containing the breakpoint.

A conditional expression may be used with a breakpoint, allowing the programmer to test for specific conditions and selectively trigger the breakpoint. The expression, that must return a Boolean upon completion, can include any arbitrary operation such as data collection.

Debugging makes an intensive use of such mechanism as gaining an in-depth understanding of a program behavior often goes through a cyclic speculate-test scheme. Stressing a hypothesis then relies on probes usage.

2.2. Methods: Iterative and Test-Driven Development. Developing reusable software typically involves many design iterations. Each iteration may introduce new requirements that change or extend the original design. This iterative process of rearchitecting a design may be described as code refactoring. Whereas reworking or rewriting code may involve dramatic changes in functionality; refactoring is an intermediate step that generally does not disturb the behavior of an application.

Extreme Programming [12] advertises the creation of unit tests for test-driven development. The developer writes a unit test that exposes either a software requirement or a defect. This test will fail because either the requirement is not implemented yet or it intentionally exposes a defect in the existing code. Then, the developer writes the simplest code to make the test, along with other tests, passes (Algorithm 1).

XP mandates a *test everything that can possibly break* strategy and provides guidance on how to effectively focus the limited resources we can afford to expend on the problem.

```
test2
    CDFGSynthesisAPI new
                example: 'example.step'
                family: 'F5'
          resF5 := self readResult.
       CDFGSynthesisAPI new
                example: 'example.step'
                family: 'F4'
          resF4 := self readResult.
     self assert: resF5 = resF4
```

Algorithm 2: Characterization-based validation for synthesis cases on two reconfigurable device families (F5 and F4). Both postsimulation memories contents are checked to be equal.

XP's thorough unit testing allows several benefits, such as simpler and more confident code development and refactoring, simplified code integration, accurate documentation, and more modular designs. Besides, these unit tests are also constantly and automatically run as a form of regression test.

The main drawback using such tests in a simulation scope is obviously to write significant tests, whereas there may be no specification available for the process being simulated.

Characterization tests act as a replacement. A characterization test characterizes the actual behavior of an existing piece of software and therefore protects existing behavior of legacy code against unintended changes via automated testing [16].

The goal of characterization tests is to help developers to verify that the modifications made to a reference version of a software system did not modify its behavior in unwanted or undesirable ways. They enable, and provide a safety net for, extending and refactoring code that does not have adequate unit tests.

2.3. Moving from Software to Hardware Design. Despite hardware debugging is a major concern, getting such facilities as speculate-test pairing support and deep state analysis are not obvious to offer in a simulation framework. One solution relies on some design patterns such as *memento* and *observer* [17]. Respectively, they are used for state recovering and propagating changes through dependent objects.

Evaluating conditions before clocking the circuit when running simulation allows to freeze the execution on demand, hence implements conditional probes. Recording a stack of states allows to trace some bugs back from a conditional break point.

In addition, the simulation framework provides feedback when refactoring the CDFG that models the system and offers by then an iterative development support. For example, the designer may change the I/O data types or substitute a parallel node by a sequential one. Further, as some portions of the initial CDFG are replaced by RTL ones during the synthesis process, refactoring also covers migrating from high-level to low-level CDFG while preserving the agile behavior of software development. As a low-level CDFG is an extension and so a special case of high-level CDFG, synthesis appears as a deep refactoring similar to source-to-source transformation at software engineering level.

As our simulation framework operates at several abstraction levels ranging from untyped functional code (Smalltalk) to RTL netlists, a set of characterization tests can be automatically generated to validate the synthesis scheme. This is done by exercising the one step above specification level (code, CDFG) with a wide range of relevant and/or random input values, recording the output values (or state changes) and generating a set of characterization tests. When the generated tests are executed against a new version of the code (e.g., synthesized CDFG), they will produce one or more failures/warnings if that version of the code has been modified in a way that changes a previously established behavior. Characterization tests remain change detectors; it is up to the person analyzing the results to determine if the detected change was expected and/or desirable, or unexpected and/or undesirable.

Algorithm 2 illustrates a characterization test for validating synthesis cases on two reconfigurable device families (F5 and F4). Both produced CDFGs are simulated on a common set of inputs. The output values are stored into memories. Identity between these two memory contents is assumed to point out that both CDFGs share a common behavior.

3. Application and System Modeling

Our multilevel simulator takes as input a specification of an accelerated application and a model of the host system. The global flow of the methodology is depicted by Figure 1.

At the highest level the application is specified by behavioral code. Entry point is not restricted to a particular language since simulation can be performed on an intermediate representation (IR), which is a Control/Data Flow Graph (CDFG), generated from the behavioral code. However, to simulate the application at a behavioral level the language has to be the same as the framework since application is executed as software.

The high-level CDFG is synthesized to a lower level, by our tool named Madeo+, and can be simulated or exported to EDIF as well as Verilog netlists. Target dependent tools perform final place and route.

Simulation of the application is integrated in a system-level simulation for a global simulation. The host system is modeled from an object framework defining components and communication links. The multilevel simulator produces Gantt and interaction diagrams giving information on the system behavior as well as signals waveforms generated by the application at RTL level.

Debugging and testing iterations are performed on simulation and synthesis results.

3.1. Multilevel Representation Supporting Simulation

3.1.1. Intermediate Representation for Modeling and Simulating Applications. A CDFG is a directed graph whose nodes are operations and edges represent data flows. A classical representation of the CDFG consists in isolating data

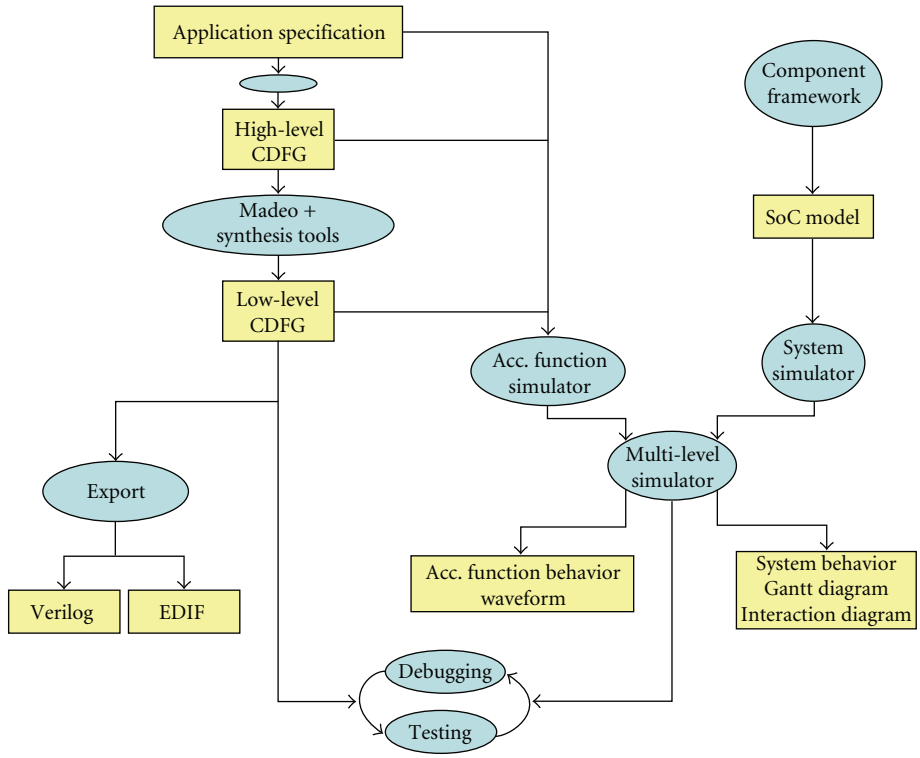


Figure 1: Global flow.

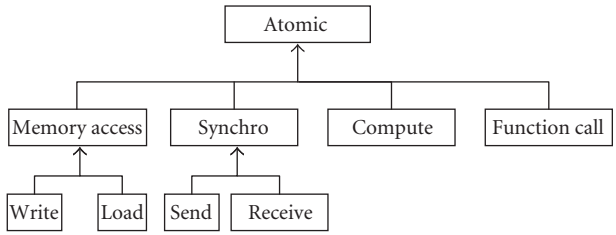


Figure 2: Atomic node inheritance tree in our CDFG model.

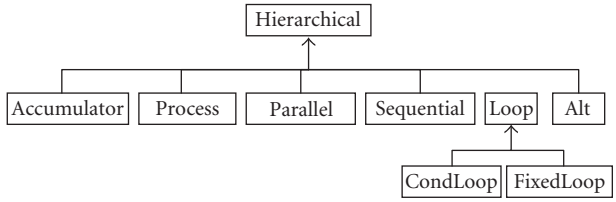


Figure 3: Hierarchical node inheritance tree in our CDFG model.

nodes constituting a DFG apart from control nodes (CFG) which role is to drive the execution flow of operations by establishing a specific instruction ordering (e.g., loops and conditions). Representing a CDFG requires to manipulate nodes and edges. Edges represent data; a datum classically knows its source, its consumers, and its type. An edge cannot cross a hierarchy; to connect to a node located into another hierarchy, the datum must be encapsulated within an *AliasData* which allows (1) a better modularity of the hierarchical CDFG and (2) renaming of signals.

CDFG nodes carry their own semantic. They do not explicitly dissociate the control from the operative nature. For example, a loop is a particular hierarchical node; it is not a control structure in charge of the execution of the loop core.

The CDFG is defined in an object-oriented environment; Figures 2 and 3 give the class hierarchy of the model for both types: atomic and hierarchical.

Atomic Nodes. An atomic node (see Figure 2) is a node without suboperators; it represents an abstract hardware primitive. If hardware primitives are not available in the target or in libraries, nodes are implemented as soft macros.

A computation is modeled by a *ComputeNode*, which result is returned as the output edge of the node. However, to simplify the detection and the processing of specific functions, some particular compute operations are handled as specific operators: memory accesses (*LoadNode*, *WriteNode*), communication, and synchronization on channels in a CSP like formalism [18] (*ReceiveNode*, *SendNode*).

Hierarchical Nodes. A hierarchical node (see Figure 3) is a node containing suboperators. We can distinguish three types of hierarchical nodes.

(i) Concurrency: *ProcessNode* for implementing parallel coarse grain operation (*threads*), *ParallelNode*, for parallel fine grain operations, or *SequentialNode* for sequential ones.

(ii) Control: loops (*FixedLoopNode* and *CondLoopNode*), alternative (*TestNode*, *AltNode*), and accumulation (*AccumulatorNode*).

(iii) Semantic-less organization: *CompositeNode* that can come from loop bodies or functional calls.

3.1.2. RTL Modeling

(a) Elementary Components. The circuit appears as a set of operators and data; the data carry their source and consumers, and the operators keep a link to their data's IOs. These double-linked dependencies enable to simulate the circuit.

(b) Object-Oriented Modeling: Polymorphism and Synthesis. The RTL modeling makes use of another structure, qualified as low-level CDFG, partially inherited from the CDFG one, but that is linked to the target platform. This structure conforms to the abstract node/data schemata (defined in the EXPRESS formalism [19] and generated using Platypus tools [20]). As this RTL model complies to the CDFG's application programming interface, and due to object-oriented facilities, these two levels can be interleaved within a single model to be simulated.

This model includes additional constructs (soft macros), primitive operators (libraries), registers/flip-flops as well as random logic nodes (e.g., BLIF format), and FSM description (e.g., KISS format). This model supports outputing EDIF files; hence it allows coupling with back-end tools providing among other feedbacks the circuit operating frequency.

Pushing forward the extension of the high-level CDFG to build up a low-level CDFG is a natural way to tailor RTL modeling. This ensures that a low-level CDFG can easily fit to any target at the cost of redefining the library of primitive operators. Such an operator is specified by its I/O bitwidth and its behavior to permit mapping from high-level CDFG nodes as well as by its latency which is used for timing generator during synthesis. An alternative solution consists in relying on random logic to design operators.

3.2. Object-Oriented System Modeling

3.2.1. A Component Approach for Structuring the System. SmallSystem is an object-oriented framework providing to the designer a set of abstract entities for modeling concurrent systems, built as a set of components interconnected through communication ports. Concepts implemented in SmallSystem are independent from any implementation language and could be retargeted to another environment/language. However, our approach takes advantage of a Smalltalk-80 environment that provides dynamic features as mentioned above [15].

An object-oriented framework provides to the designer a set of software components that can be directly used or tailored for a given application by subclassing. An application appears as an extension of the framework. The more it is subclassed, the more it is enriched with new reusable functionalities, bringing to the designer a flexible resource. Moreover, all the classes of a framework inherit from common classes' hierarchy. Therefore, they share a set of methods allowing to apply generic algorithms to all the elements of the framework. The tools addressing the framework are developed independently from specific extensions guaranteeing a high-degree of reusability.

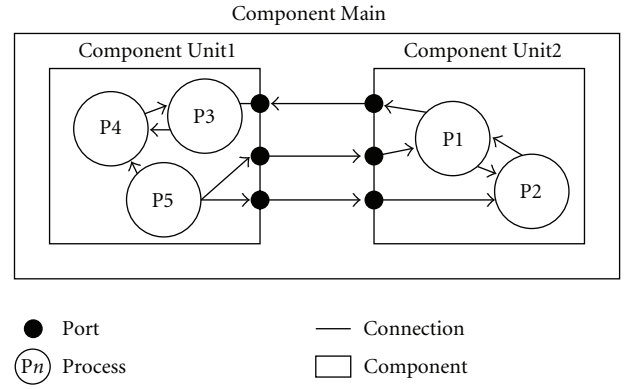


Figure 4: Example of a model with three components. The top hierarchy is the component *Main* that encapsulates two subcomponents *Unit 1* and *Unit 2*. Internal activities of each subcomponent communicate through the ports of the interfaces.

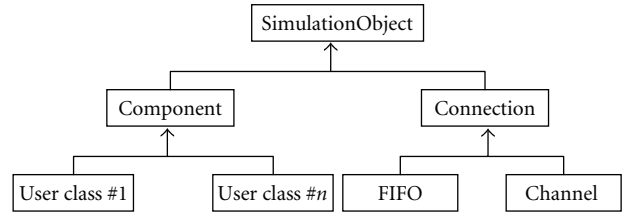


Figure 5: Class hierarchy of the modeling framework SmallSystem. In order to use simulation functionalities the framework SmallSystem inherits from the simulator.

SmallSystem defines abstract entities that are specialized by inheritance for modeling concrete system elements and communication links. An abstract system element is structured as a component implementing a behavior and a communication interface defined as a set of input and output ports. A system is built by interconnecting the component's interface with communication links of varying semantics and capacities (see Section 3.2.2). A component encapsulates its internal states and behaviors meaning that only communications with its interface can modify them. For integrating a component in a system, only its functionality and interface have to be known. This approach enables to define concrete elements that are structurally independent from a specific model. Additionally a component can be hierarchical; it then contains an entire subsystem being connected through the interface of its encapsulating hierarchy to an external component. The internal activities of an encapsulating component can also communicate with a subcomponent.

Figure 4 illustrates an example of a hierarchical model. The top hierarchy is *Main* encapsulating the two components

Unit1 and *Unit2*. *Unit1* defines its internal activities by three processes. Its interface defines three ports, and two processes are in charge of the external communications. Internal communications between processes are performed by local declarations of communication links.

3.2.2. Raising Up the Abstraction Level: Object-Oriented Modeling. Figure 5 illustrates the class hierarchy of the SmallSystem framework. The classes *User Class (1 and n)* correspond to an extension of the framework.

The basic framework elements are as follows.

(i) Component. It represents abstract behavioral system elements. A specialization adds behaviors to the component by the definition of additional methods corresponding to internal activities. These methods can be executed as processes or used as classical functions. An interface is defined by the declaration of input/output ports in an initialization method. The internal states are represented by instance variables and are not directly accessible by external components.

In order to access the simulation features this class inherits from the simulator framework. More details about the simulator are given in Section 4.3.1.

(ii) Connection. This abstract class corresponds to a connection between two component ports and allows to perform communications. At this abstraction level no semantic is associated to the connection. The class is specialized in two entities that are Channel and FIFO. The former is an equivalent of a CSP channel as defined in Occam language [21]. Communications are synchronized by *rendezvous* providing a fine-grain synchronization. The latter is a classical FIFO with a parametric size. Communications are blocking if either the FIFO is empty or the maximal capacity is reached (this parameter is defined by the designer).

Because of the delays created by the synchronizations when the model is simulated, this class also inherits from the simulator framework.

4. Methodology of Simulation

A multilevel simulation needs to manage different models addressing the different abstraction levels. This section details the simulation models used by our framework and how they interact. The simulation engines for system-level and RTL-level simulation engines are described. Automated HDL wrapper generation ensuring interoperability of our framework with mainstream tools is explained.

4.1. Simulation Models. A simulation model defines the evolution of the system's states according to the simulated time. The model we use for system-level simulation is based on discrete states with an event-driven approach while the RTL level circuit simulation is cycle-accurate, that is, time-driven.

An *event* corresponds to a change of state in the simulated system caused by incoming data or by internal processes. A system state simulated in a discrete event model is defined by discrete state variables. Events are produced at discrete instants called *dates*. Continuous variable values, including time, are taken into account only when they are used, that is, at event dates.

Discrete event simulation can be classified according to how the simulation time goes on: event-driven and time-driven.

Event-driven model does not use global clock; the time is updated with the next earliest event date. Thus the time evolution only corresponds to an ascending order of the dates. The event scheduler consumes the event queue in this order, triggers the events, and updates the simulated time. Because of the sparse event repartition in a system simulation, the event-driven model is well suited for behavioral system simulation compared to RTL level. It avoids simulating empty cycles.

The simulated time of a time-driven model is incremented by a constant time step that is set according to the problem. For each time step, the event scheduler searches for dates in the list of events corresponding to the current step, then the selected ones are executed. Obviously, it is possible to obtain a time step with no events scheduled wasting simulation time. Time-driven simulation is interesting for an RTL level circuit simulation because of the activities produced at each clock cycle such as updating values in flip-flops. In this case the time step is set to one clock cycle.

4.2. Embedding Cycle-Accurate Simulation within an Event-Driven Scope. The platform described in this paper combines two complementary simulation models for simulating an application running on an RSoC. The system level is simulated by a discrete-event and event-driven simulator whereas the RTL level, corresponding to an application mapped on an accelerator, is simulated by a cycle-accurate simulator. This configuration raises the integration problem of both simulation models with separate notions of time. In order to synchronize both models, the cycle-accurate simulation is embedded within the discrete-event one as an atomic task. At the discrete-event level, when a start signal is received, for example, from a DMA controller, the mapped application is executed by starting the cycle-accurate simulator (for more details on the API used to interface both simulators see Section 4.3.2(b)). When computation ends, it returns the amount of cycles spent. This latency is converted in the time unit used by the discrete-event simulator in function of the accelerator frequency. The result corresponds to the delay applied in the event-driven simulation for simulating the application execution. Although the mapped application is time-driven, at the system-level the simulation is homogeneously event-driven.

4.3. Simulation Engines

4.3.1. System-Level Simulator. The high-level discrete-event simulator is event-driven and mainly inspired by [14]. The simulation kernel is based on a scheduler providing all the functionalities related to the creation, the scheduling, and the execution of events. A simulation model is composed of simulation objects that create delays for simulating execution

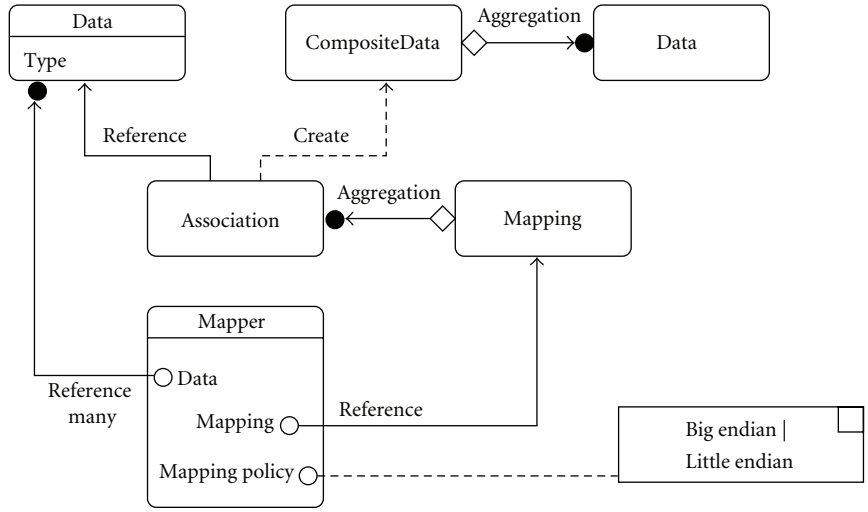


Figure 6: The mapper links high-level data to an aggregation of Boolean data, with respect to data type.

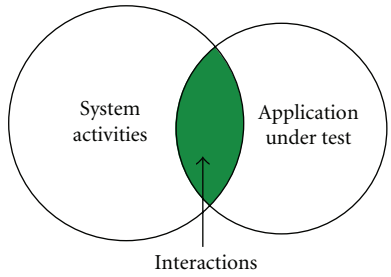


Figure 7: Interactions between the design under test and the host system correspond to the activities emulated by the mock object.

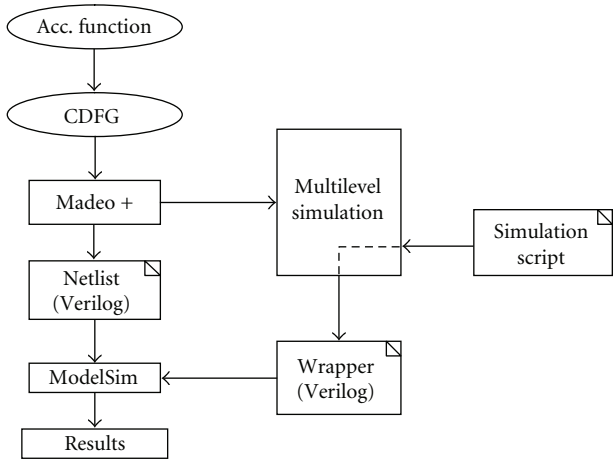


Figure 8: Simulation with generated wrapper and third party tool: ModelSim.

latencies and schedule activities. The initial scheduling of Smalltalk methods belonging to the components performs the simulation of a system model. They represent the component internal activities and are executed as processes.

Methods are executed by the Smalltalk virtual machine and by default spend no time in the simulation. Latencies of tasks are simulated by the insertion of specific calls to the simulation kernel features for creating delays and tracing the activities. The delay specified can be constant or generated in function of a probability law for simulating nondetertiministic behaviors such as bus contentions.

This technique allows monitoring fine grain system activities and allows multiple levels of detail. For example a task can be seen as atomic or traced at different points for more accuracy. The simulation trace obtained is used by the visualization tools presented in Section 5.1.

In order to have access to the simulation kernel functionalities, the modeling framework inherits from the simulator class *Simulation Object* as illustrated by Figure 5.

The simulation kernel defines two abstract classes.

(i) Simulation Object. This class corresponds to an abstract simulation object and gives a restricted access to the simulation kernel. It only provides to its subclass a set of methods for scheduling and suspending activities. For example, a simulation object has no direct access to the event queue under the responsibility of the scheduler. The components and communication links of the modeling framework inherit from this class.

(ii) Simulation. This class is the entry point of a simulation model. It permits to initialize the simulator and to start a simulation by scheduling the initial activities.

4.3.2. Cycle-Accurate Simulator for Mapped Applications. The simulation consists in computing the current value for the data. Simulation is performed at RTL level and is cycle-accurate.

Simulation relies on the following circuit structure.

(i) *Data* are computed by operators.

(ii) *Constant Data* force evaluating their consumers.

(iii) *Operators* compute their outputs following a data driven scheme.

(iv) *Flip flops* own a special behavior to delay their change until the next clock edge.

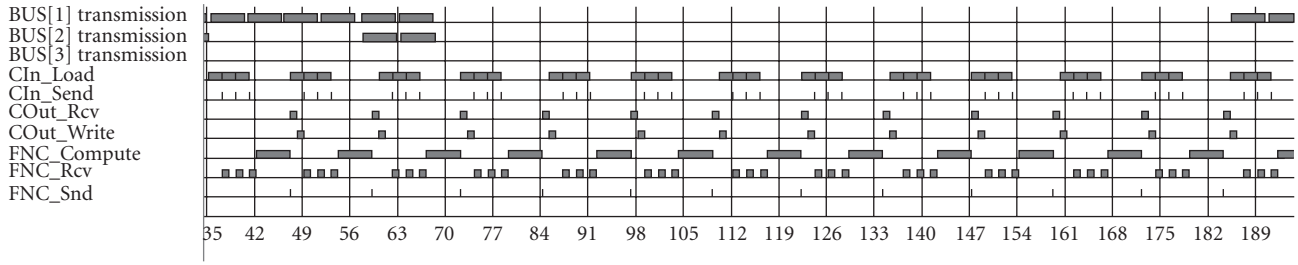


Figure 9: Gantt diagram generated by the system simulator.

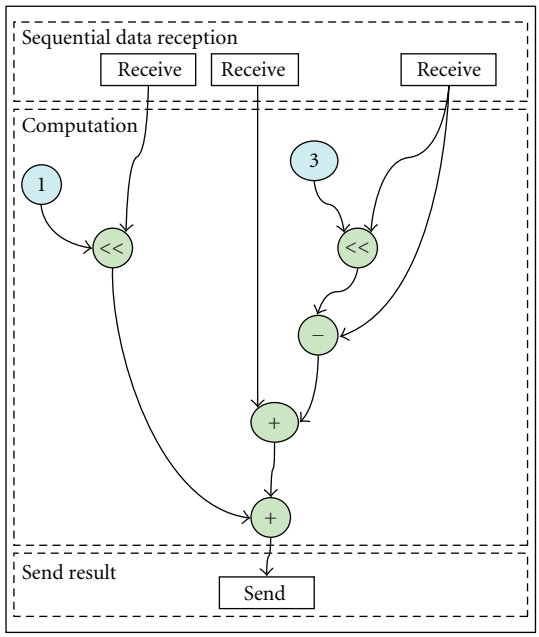


Figure 10: Application is an FIR filter function connected to input and output address generators (not represented on this figure).

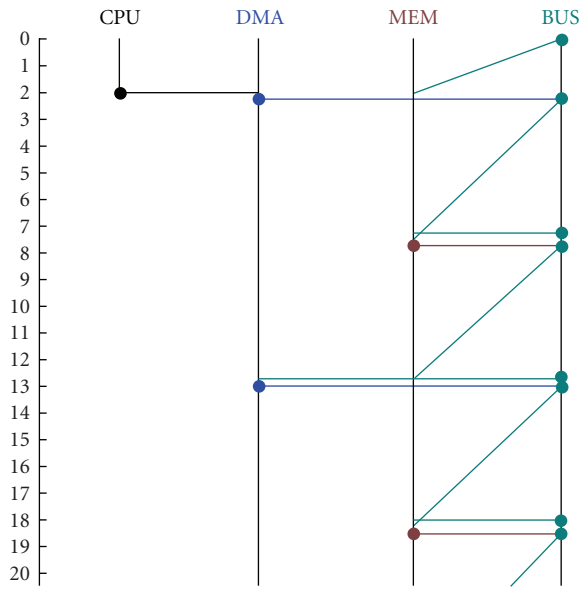


Figure 11: Interaction diagram generated by the system simulator.

The algorithm iterates over data and tries to fire their consumers evaluation. *Evaluation* is correct when all data changes have been considered.

This algorithm cools down until data values are stabilized. Clocking the circuit validates the flipflops' values and evaluates again.

(a) High-Level CDFG. As the mapped structure keeps alive the link between high-level variables and low-level signal (see Figure 6), both low-level and high-level values can be simultaneously probed during simulation (the *mapping Policy* is used to rebuild the high-level values based on the signals Boolean values).

(b) Low-level CDFG Simulator API. The low-level simulator provides an API enabling to set conditional breakpoints on the low-level CDFG top interface signals. SmallSystem accelerator components interact with the low-level CDFG under simulation through the API by defining execution scenarios in a method.

(c) Probing CDFGs. Two kinds of conditional breakpoint are used which either stops the simulation or performs an action when triggered. Actions and breakpoints are defined in a metamodel which is instantiated and used by the simulator. This model can also be used for generating an HDL wrapper for simulation with third party tools (see Section 4.4).

Algorithm 3 gives the syntax for a conditional breakpoint set on the application termination signal stopping the simulation when signal *done* is equal to 1. It corresponds to the instantiation of a probe object with an implicit action. The signal is an output and is retrieved by its name in the low-level CDFG through the simulator API.

In order to simulate response on signals, it is also possible to define breakpoints that perform user-defined action. Algorithm 4 defines a conditional breakpoint associated to an action setting the signal *dma_Req_Read* to value 0 after 10 cycles when signal dma_Ack_Read is equal to 1. The message *forceSignal:* sent to the class *SimulatorForceSignal* creates an action executed by the low-level simulator.

4.4. Automated Wrapper Generation for Third-Party Simulation. Stimuli are performed by a wrapper mimic behavior of the external environment such as systems activities. In software engineering and object-oriented development this technique is referred as *mock object*. A mock object simulates a real object by defining the same interface and

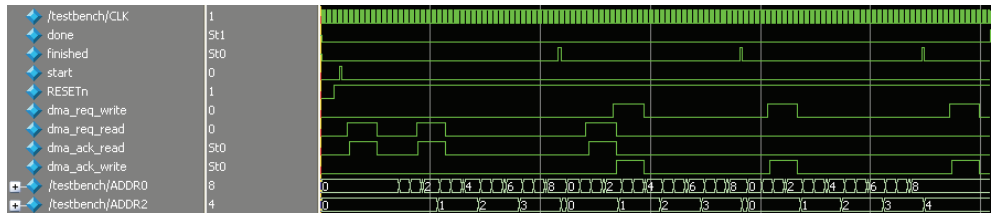


Figure 12: ModelSim results.

```
Simulator
    probe: (Simulator synthesizedCDFG
                                outputNamed: 'done')
    relation: '='
    value: 1
    probeName: 'Task Done.'
```

Algorithm 3: Conditional breakpoint set on the application termination signal. When the breakpoint is triggered, the simulation is stopped.

```
Simulator
    probe: dma_Ack_Read
    relation: '='
    value: 1
    probeName: 'dma_ack_read'
    action: (SimulatorForceSignal
                        forceSignal: (dma_Req_Read)
                        to: 0
                        in: 10).
```

Algorithm 4: Conditional breakpoint associated to a user-defined action. The signal dma_Req_Read is set to 0 when the breakpoint is triggered. The action can also be conditional, and multiple actions can be defined in an array.

providing equivalent services but without implementing the real functionalities. It enables to quickly validate the behavior of another object in a controlled way.

In a standard hardware design flow, applications are validated through HDL simulations performed by mainstream tools such as ModelSim [22]. Validation goes through using a wrapper written in HDL which defines a set of stimuli interacting with the application's top interface. Writing HDL wrappers can be burdensome and time-consuming when applications have complex interfaces. Furthermore each wrapper is specific to a given application, and it is necessary to rewrite it for addressing different cases. In order to ensure the interoperability of our methodology with mainstream tool flows and to increase productivity, HDL wrappers are generated from a higher-level specification.

In our framework, system activities, such as data transfers, are modeled by SmallSystem (see Section 3.2) and interfaced with the application through the low-level simulator API. However, it is also possible to test the application apart from a system model by defining stimuli in a standalone script. Scripting produces stimuli by instantiating the metamodel described in Section 4.3.2(c); stimuli correspond to breakpoints with actions simulating system activities (normally handled by the system model, e.g., memory requests) that interact with the design's top interface. The mock object corresponds to the probing model made up of the set of stimuli.

Activities emulated by the mock object correspond to the intersection between system activities and the design (see Figure 7), for example, DMA handshakes and memory requests.

This software approach takes advantage of testing and debugging features as described in Section 2. The abstraction level of scripting enables to be more productive; moreover, low-level aspects are taken into account by the simulator (scripting and generating HDL wrapper enables to save 50% of the designer's coding effort compared to hand-written HDL wrapper). For example, signal declarations or design instantiation is not necessary since the simulator gives access to the top interface through an API.

In order to ensure the interoperability of our methodology and final validation by mainstream tools, a Verilog wrapper is generated from the mock object. This corresponds to a model-to-model transformation with refinement on data. A probe on a variable is translated to a composite probe on a signal vector conforming to Figure 6. Two rewriting schemes are supported depending of the kind of breakpoint (Algorithms 4 and 3) as shown in Algorithm 5. The generated wrapper is used for testing the design netlist produced by Madeo+ as depicted by Figure 8.

Algorithm 5 shows the generated Verilog code for breakpoints of Algorithms 4 and 3. Declaration of signals and design instantiation are generated automatically as well.

5. Simulation Results

In order to perform verifications on the model, the designer needs tools to visualize the concurrent activities of the system under test. Our visualization tools show the system behavior as well as the interactions between components at different levels of detail. For example, at the system level, the activities are seen as tasks evolving on a time scale while at the mapped application level each clock cycle is reported.

To illustrate this, a good case study must be an application simple enough to ease explanation, that makes sense in a DSP scope and that designers do know pretty well, with no compromise on the ability to highlight the benefits our approach carries. These considerations lead us to select an

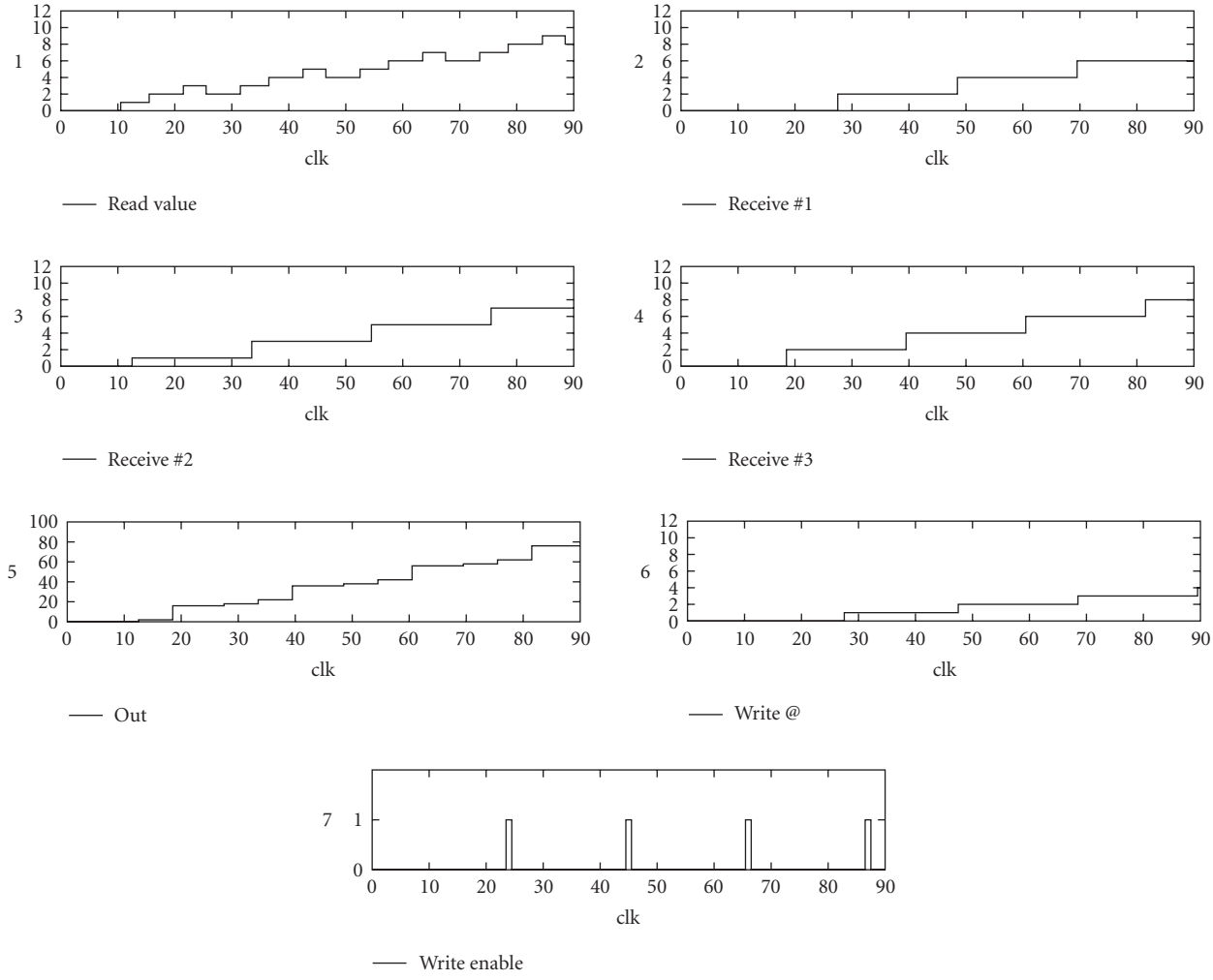


Figure 13: Internal view of a low-level simulation.

FIR filter. The global application is on purpose simplified: it is composed of two processes performing reading/writing operations on accelerator's local memories (input and output processes) for feeding an FIR filter function process and writing back results. CDFG representation of the function is given by Figure 10. The function receives three data from the input processes and sends back one result to the output process.

5.1. System-Level Simulation. The system simulator reports the behavior of each traced activities on a Gantt diagram. Figure 9 illustrates the system-level simulation of the FIR filter application running on a modeled SoC including a main memory, a bus, a DMA controller feeding accelerator's local memories, and a reconfigurable accelerator. The task names are listed on the Y-axis and the time scale on the X-axis. All the system components are modeled at a behavioral level in the SmallSystem framework.

Only the needed functionalities of components have been modeled at a high abstraction level. The processor task is to initialize the DMA and sends a start signal to the accelerator. The DMA controller performs communications in a pipelined way. The bus transfers the packets and simulates contention penalties by a probability law chosen by the designer.

The diagram of Figure 9 shows tasks related to a local execution on the reconfigurable accelerator. Traces prefixed by *BUS* show the data transfers between the main memory and the accelerator. Activities of input and output processes computing addresses for local memory accesses are reported by traces *CIn* and *COut*. Function's activities (receive, compute, send) are given by traces prefixed by *FNC*. Figure 9 focuses on the application behavior, but it is also possible to visualize system activities such as memory accesses and DMA transfers.

At this level the application is specified in Smalltalk, and the simulation is performed at software level. The refinement of the abstraction level is obtained by a translation of Smalltalk into a CDFG taken as input of the cycle-accurate simulator integrated as a component in the model (see Section 5.2).

A Gantt diagram gives a global vision of the system behavior with a fine grain tracing of the components' internal activities. Another aspect concerns the interactions

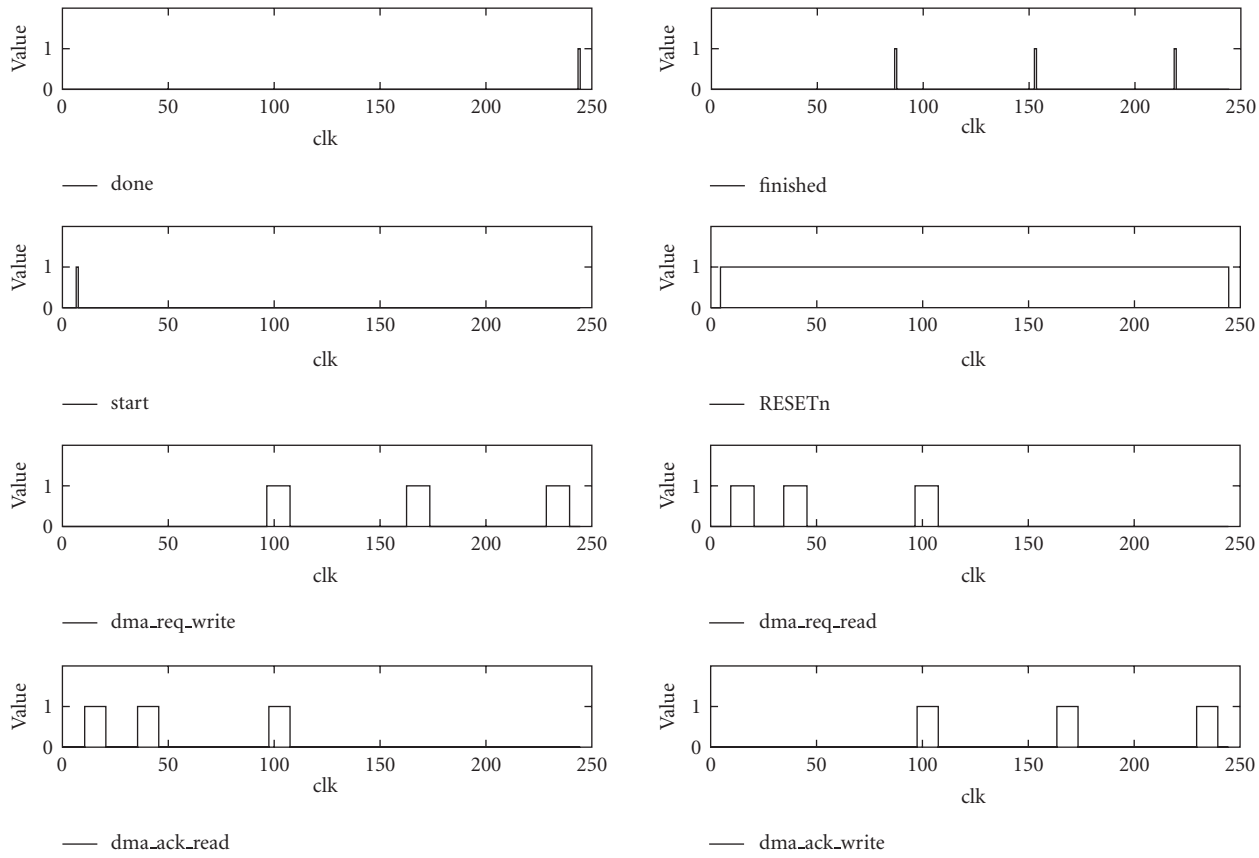


Figure 14: Low-level simulation for a computation divided in three phases.

```
initial begin
    @(posedge done); − − HALT
    $stop
end
always @(posedge dma_ack_read) begin
    #(PERIOD ∗ 10) dma_ack_read = 0;
End
```

Algorithm 5: Generated Verilog code of two conditional breakpoints.

between the components performed by communications. Communications between SoC's components are represented by an interaction diagram shown in Figure 11. A circle corresponds to a communication starting point. For example, at the time 2 the CPU sends a start signal to the DMA for initiating data transfers to the reconfigurable accelerator. Then a request is sent to the main memory (*MEM*) through the bus (*BUS*).

5.2. Cycle-Accurate Simulation of an Application. Figure 13 presents a list of traces for the low-level CDFG simulation of the previous example.

The curves represent, respectively, the value read from memory (1), the buffered values for the FIR filter computation that come from the read values at different timestamps (2) (3) (4) and correspond to the three receive operations in Figure 10, FIR filter result (5), the write address (6), and the write activation (7). Signals (1) to (6) carry numerical values while (7) is an RTL Boolean value.

Comparing Figures 9 and 13 clearly illustrates both the multilevel simulation, with as an example system activities (top three lines of Figure 9) on one side and low level and accurate values manipulation on the other side (Figure 13), and both the correlating values (line 7, Figure 9 with (7), line 9 with (2) to (4) sampling values).

5.3. Simulation with Third-Party Tools. The generated Verilog wrapper and the Madeo+ netlist are taken as input of ModelSim for final validation. Figure 12 shows application activities at the netlist level. Activities such as DMA (signals *dma*) result from interactions between the netlist and the testbench (Algorithm 4). They also appear in Figures 14 and 13. At a higher abstraction level, read/write requests are traced on Figure 9. Addresses generated by IOs processes are *ADDR0* and *ADDR2*; they also appear in Figure 13.

6. Conclusion

This paper introduces a methodology for multi level simulation of applications running on an RSoC. This work focuses on fast and early verification while preserving the ability to deeply probe the RTL model. Our debugging scheme

exploits several object-oriented facilities: agile development, test-driven design, and abstraction. Multilevel ranges from behavioral code execution to mainstream simulation engines (e.g., ModelSim) and addresses both system activities (e.g., data moves) and accelerated tasks. The successful design of significant test cases has confirmed this approach to be very valuable.

Current on-going work, related to this, addresses implementing the breakpoints as hardware primitives, with a detailed study on performances and the characterization of probe-effect in case of synthesized probes.

References

[1] A. Bernstein, M. Burton, and F. Ghenassia, "How to bridge the abstraction gap in system level modeling and design," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '04)*, pp. 910–914, IEEE Computer Society, San Jose, Calif, USA, November 2004.

[2] IEEE, "Std 1076-2000: IEEE Standard VHDL Language Reference Manual," IEEE, 2000.

[3] D. E. Thomas and P. R. Moorby, *The VERILOG Hardware Description Language*, Kluwer Academic Publishers, Norwell, Mass, USA, 1996.

[4] "Open SystemC initiative," http://www.systemc.org.

[5] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, "The specC methodology".

[6] R. Helaihel and K. Olukotun, "Java as a specification language for hardware-software systems," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '97)*, pp. 690–697, IEEE Computer Society, San Jose, Calif, USA, November 1997.

[7] A. Habibi and S. Tahar, "Design for verification of SystemC transaction level models," in *Proceedings of Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 560–565, Munich, Germany, March 2005.

[8] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai, "A framework for object oriented hardware specification, verification, and synthesis," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 413–418, Las Vegas, Nev, USA, June 2001.

[9] M. C. W. Geilen, J. P. M. Voeten, P. H. A. van der Putten, L. J. van Bokhoven, and M. P. J. Stevens, "Object-oriented modelling and specification using SHE," *Computer Languages*, vol. 27, no. 1–3, pp. 19–38, 2001.

[10] S. Vernalde, P. Schaumont, and I. Bolsens, "An object oriented programming approach for hardware design," in *Proceedings of IEEE Computer Society Workshop on VLSI (WVLSI '99)*, Orlando, Fla, USA, April 1999.

[11] S. Swamy, A. Molin, and B. Covnot, "OO-VHDL: object-oriented extensions to VHDL," *Computer*, vol. 28, no. 10, pp. 18–26, 1995.

[12] "Extrem programming methodology," http://www.extreme-programming.org.

[13] L. Lagadec, J. Boukhobza, and A. Plantec, "Chaîne de programmation pour architecture hétérogène reconfigurable," in *Le Prochain Symposium en Architecture de Machines (SympA '08)*, 2008.

[14] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Boston, Mass, USA, 1983.

[15] "Visualworks smalltalk," http://www.cincom.com.

[16] M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[17] S. R. Alpert, K. Brown, and B. Woolf, *The Design Patterns SmallTalk Companion*, Addison-Wesley, Boston, Mass, USA, 1998.

[18] C. A. R. Hoare, "Communicating Sequential Processes," 1985.

[19] Part 11: edition 2, "EXPRESS Language Reference Manual," ISO 10303-11, 2004.

[20] "Platypus Technical Summary and download," 2007, http://cassoulet.univ-brest.fr/mme.

[21] S.-T. M. Limited, OCCAM 2.1 reference manual, 1995.

[22] "Modelsim," http://www.model.com.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 851613, 15 pages
doi:10.1155/2009/851613

Research Article

Providing Memory Management Abstraction for Self-Reconfigurable Video Processing Platforms

Kurt Franz Ackermann,[1,2] Burghard Hoffmann,[2] Leandro Soares Indrusiak,[1] and Manfred Glesner[1]

[1] *Institute for Microelectronic Systems, Darmstadt University of Technology, Karlstrasse 15, 64283 Darmstadt, Germany*
[2] *VITRONIC Dr.-Ing. Stein Bildverarbeitungssysteme GmbH, Hasengartenstr. 14, 65189 Wiesbaden, Germany*

Correspondence should be addressed to Kurt Franz Ackermann, kurt.ackermann@vitronic.com

Received 30 December 2008; Revised 24 April 2009; Accepted 15 June 2009

Recommended by Peter Zipf

This paper presents a concept for an SDRAM controller targeting video processing platforms with dynamically reconfigurable processing units (RPUs). A priority-arbitration algorithm provides the required QoS and supports high bit-rate data streaming of multiple clients. Conforming to common video data structures the controller organizes the memory in partitions, frames, lines, and pixels. The raised level of abstraction drastically reduces the complexity of clients' addressing logic. Its uniform interface structure facilitates instantiations in systems with various clients. In addition to SDRAM controllers for regular applications, special demands of reconfigurable platforms have to be satisfied. The aim of this work is to minimize the number of required bus macros leading to relaxed place and route constraints and reducing the number of critical design paths. A suitable interface protocol is presented, and fundamental implementation issues are outlined.

1. Introduction

Industrial video processing applications typically require high data-rates to obtain detailed information on target scenes. Since real-time demands of complex algorithms often cannot be satisfied by software solutions, suitable hardware implementations are required [1]. Adequate processing rates and a high degree of flexibility make today's FPGAs a preferable technology for video-processing implementations. Nevertheless, limited resources of FPGAs are bottlenecks for many complex algorithms. Modern FPGAs support the feature of partial dynamic reconfiguration enabling the change of FPGA sections during run-time while the remainder of the device continues to operate [2, 3]. Within the research work described in this article, this technology is applied to a video processing platform equipped with a single FPGA [4, 5]. If a complex algorithm can be decomposed into a set of smaller sequentially executed processing units, time-multiplexing might significantly reduce demands on FPGA resources. A reconfiguration order is determined according to an initially transferred scheduling plan by the FPGA itself. The ability of mapping tasks to reconfigurable processing units (RPUs) during run-time makes the platform suitable for a wide range of algorithms.

Depending on application demands, single lines or even multiple frames have to be buffered by the hardware in order to facilitate the desired functionality. In this regard, systems deploying high-resolution cameras, with frame dimensions of three mega pixels and more, further increase memory requirements. Today's largest FPGA devices contain approximately 3 MB on-chip memory [6, 7], allocated to some extent by a design's configuration data already. Thus, insufficient memory is available for buffering video data and intermediate processing results. Owing to this lack of memory, a suitable platform is equipped with additional fast external DDR SDRAM, connected to the FPGA. This article presents the concept of an application domain specific SDRAM controller providing an appropriate QoS.

The design of RAM controllers in multimedia applications has been widely discussed already [8–11]. Instead of targeting the physical access to the RAM, the presented work moreover focuses on providing services on higher

abstraction layers. A motivation for a generalized addressing scheme is not at least due to increasing the compatibility between various cores, producing differently arranged output data. Video processing applications typically aim to address data in units of frames, lines, and pixels, defining the desired level of abstraction. Furthermore, multiple clients aim to store results concurrently in a shared RAM and request data from different locations as well. Thus, a suitable memory management is indispensible and too complex to be handled by the processing cores themselves. This work encompasses a centric solution inside the memory controller, with support for multiple partitions and scalable frame buffers.

Memory controller designs with dynamically reconfigurable clients have further demands also not considered in research projects so far. Although dynamic reconfiguration does not inherently increase the resulting traffic within applications, problems arise due to a more complex place and route stage as part of the implementation process [5, 12]. Especially signals crossing reconfigurable boundaries potentially induce critical paths. Due to a majority of affected port signals in designs deploying a memory controller, the article discusses important related aspects and provides an adequate concept of a communication interface and protocol.

In the remainder of this article, Section 2 introduces the underlying self-reconfigurable frame grabber platform. Its functional components and essential data paths are outlined. Section 2 further defines the required QoS and provides scaling methodologies. The concept of the proposed memory controller is presented in Section 3. Demands and difficulties in dynamically reconfigurable designs are explained, and a suitable interface protocol is introduced. Section 3 details the implementation of the controller's modular structure. Focal points are priority arbitration, instruction decoding, and memory organization. An abstract case study in Section 4 provides results of real time analyses. Furthermore, the operating sequence of a sample client is demonstrated. Finally, Section 5 summarizes this work.

2. Reconfigurable Video Processing Platform

The aim of this section is to gain an elementary understanding of a dynamically reconfigurable framework required to deploy the proposed memory controller. A corresponding video processing hardware platform is realizable either as a smart camera or as a frame grabber, basically differing in the data source only. The latter will be exemplified in the following due to its wider range of applicability. Essential components of frame grabbers are interfaces to cameras and PCs, data processing cores, and fast on-board memories. Aiming to obtain a multifunctional and at the same time flexible platform, processing cores are typically implemented in FPGAs [1, 13].

Additional to conventional frame grabbers the proposed concept relies on partial dynamic reconfiguration. In spite of the implementation overhead, this technology offers significant advantages regarding resource utilization. Video processing systems typically contain a number of reusable components arranged in a pipeline. Each element has to be implemented and synthesized regarding the underlying hardware platform. Thus, a desired functionality is achievable by sequentially configuring the generated partial bitstreams into reconfigurable processing units (RPUs) during run-time. Time-multiplexing of IP-cores generally enables the designer to extend the depths of applications realizable on a certain device. Nevertheless, such a reconfiguration principle also incurs some considerable drawbacks not at least motivating for the improvements introduced in Section 3. The first to mention is the increase of the system's overall latency, which is directly correlated to occurring reconfiguration times. In order to compensate the latency overhead, a frame grabber based on a minimum of two RPU slots, operating mutually exclusive, inherently incorporates an adequate solution [5, 14]. An according reconfiguration scheduling for a sequence of n jobs is presented in Figure 1. While one RPU is processing (R), the other one is configured (C) and parameterized (P), and vice versa. Therefore, the scheduling order must avoid reconfiguration times exceeding concurrent processing periods, making it imperative that the reconfiguration process allocates a minimum time slot. Thus, the writing of partial bitstreams to the FPGA's configuration memory needs to be controlled by the FPGA itself, referred to as self-reconfiguration.

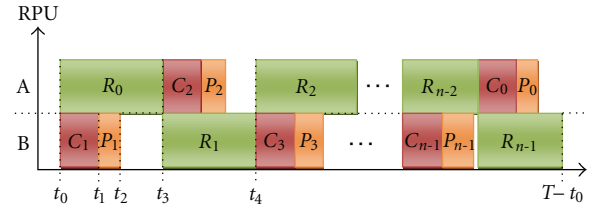


Figure 1: RPU job-scheduling.

2.1. Modular Structure and Data Flow. Essential components of the proposed framework are combined in Figure 2. Except for an external DDR-RAM and SRAM, all components are embedded inside a single FPGA. The paths of image data, parameters, and partial bitstreams are outlined subsequently aiming to explain the principle of operation and moreover to elaborate required features of the aspired memory controller.

A Camera-Link (C-Link) interface establishes a connection between the frame grabber and a camera. In order to support various configurations according to the C-Link standard definition [15], the interface is dynamically reconfigurable, too. All data IO passes through the proposed memory controller implementing the design's data communication center. Received video data is buffered in an assigned RAM section and made available to all clients. The platform contains n RPUs. Each of them may access the memory controller with requests to read buffered data and to write back intermediate results. A Gigabit-Ethernet module realizes the communication interface to a connected PC. After the final pipelined application stage has been configured in an RPU and finished processing, the

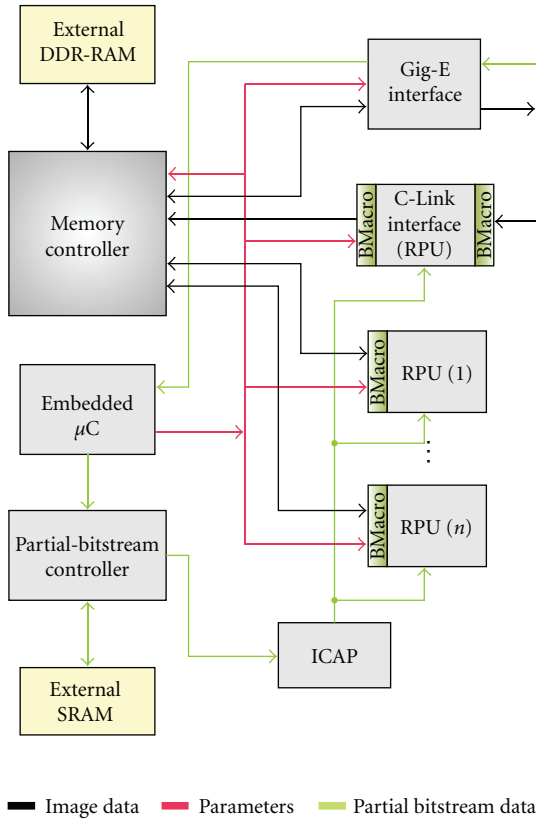


Figure 2: Frame grabber structure.

memory controller transfers the results to the PC. Finally, the processing cycle starts anew as the next image is received from the camera.

Not least because most algorithms are sensitive to changes in the environment—parameterization is indispensable for video processing applications. The capability to transfer information regarding the configuration, the application, and the environment to dedicated design-modules during run-time significantly increases flexibility. In this manner, an initial set of parameters is determined at system start-up time and transferred together with a core scheduling plan to a universal embedded microcontroller (μC), realizing the sequence in Figure 1. A suitable resource efficient parameterization interface enables a proper distribution of parameters from the μC to all frame grabber modules [5]. Furthermore, partial bitstreams, resulting from RPU-specific algorithm implementations, are initially transferred from the PC to the platform's SRAM. Eligible FPGAs support self-reconfiguration by providing an Internal Configuration Access Port (ICAP), implementing an interface between the user-logic and the configuration memory. During operation, the μC periodically triggers reconfigurations by initiating a partial bitstream transfer from the SRAM to the ICAP interface.

2.2. Quality of Service (QoS) Requirements. In industrial applications, all system components have to ensure a defined quality of their services to satisfy global real-time specifications. In case of the presented framework, the desired QoS can be expressed as the maximum frame-rate and resolution that can be processed without missing any input-data after a certain latency period.

Video data is continuous media, producing periodic processing loads. As a matter of principle, periodic demands are easier to comply with than sporadic ones [16]. Since the presented design implies a deterministic system with predictable bandwidth demands, a certain minimum QoS can be guaranteed to the user at system-startup time. The QoS is mainly negotiated between the PC, running the design automation flow [4] and the FPGA-embedded microcontroller on the hardware platform. The PC must be knowledgeable of the peak data-rate, transmitted by the camera. Major determining factors are frame size, line size, line gap, frame gap, and pixel frequency. The design automation software obtains detailed information about the underlying hardware structure by a hardware capabilities report of the microcontroller. Consequently, a reconfiguration scheduling plan is created regarding the data I/O demands of particular stages within the processing pipeline, and priorities are assigned to all RAM clients. Finally, communication requirements of the platform's Gigabit-Ethernet interface, such as packet-size, packet-rate, bandwidth, and latency, are limiting the overall QoS.

The heterogeneity of application demands requires the services of system components to be parameterizable. Thus, if a desired QoS is not achievable, the platform informs the PC about the location of the bottleneck. The QoS could be scaled down, if bottlenecks cannot be resolved by adaptations of the scheduling cycle or a redistribution of priorities. Therefore, entire input frames could be periodically skipped (temporal scaling) or the frame-size reduced (spatial scaling). The application specific definition of an adequate scaling methodology always depends on the location of the bottleneck. Within the presented framework, scaling is initiated by the PC and can be activated in the platform by a simple parameterization of its components. A general and more detailed discussion on achieving an appropriate QoS in scalable video applications is given in [17].

3. SDRAM Controller Concept

After the operational environment of the proposed memory controller has been outlined in previous sections, subsequent the key idea underlying this article is brought into focus. Not all video processing cores read their input data linearly, as received from a camera. Especially complex algorithms may require random access on data of multiple frames. The proposed SDRAM controller introduces an additional layer of abstraction, providing dedicated services to video processing applications. Thus, an abstracted addressing scheme increases the compatibility between diverse cores in the processing pipeline. Furthermore, the complexity of RAM clients can be drastically reduced, as they need not to consider the underlying memory organization. The fundamental architecture of the proposed memory controller

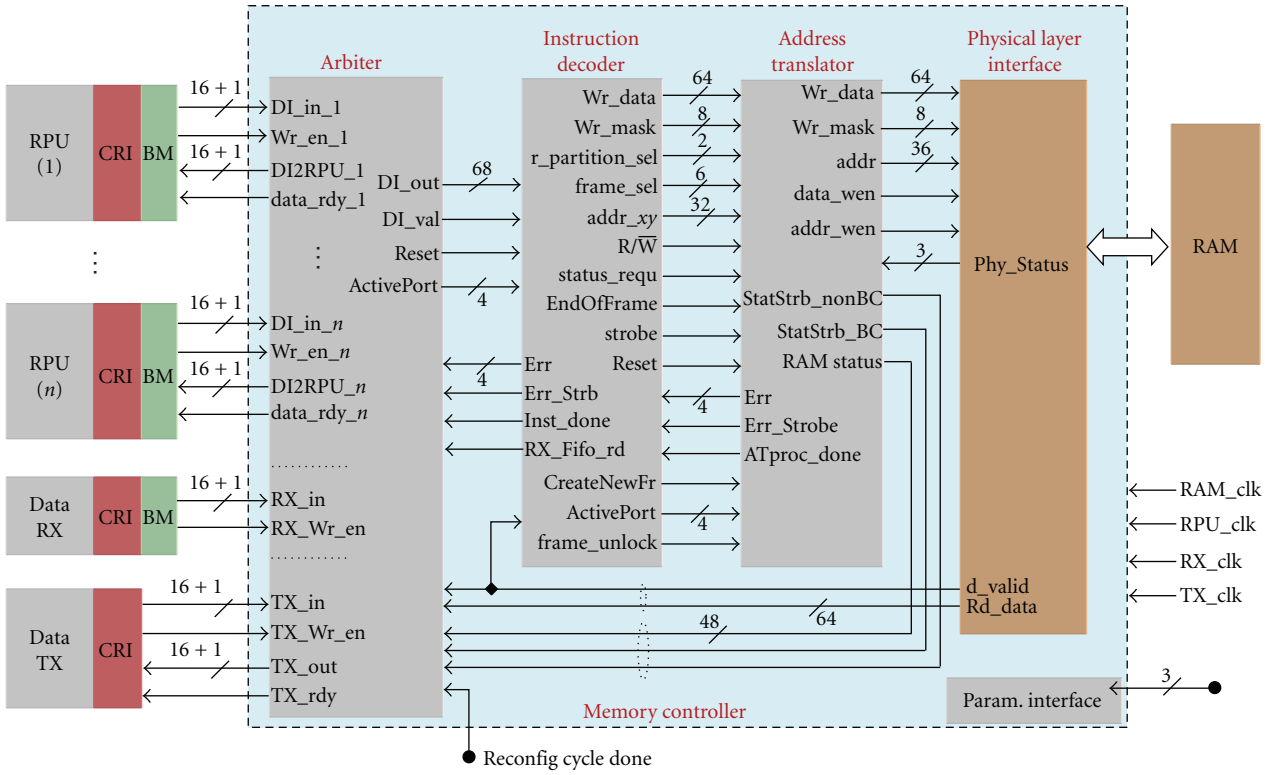


Figure 3: Architecture of the memory controller.

is presented in Figure 3. Four hierarchical modules and a parameterization interface provide the aspired functionality:

(i) support for multiple clients,

(ii) priority arbitration,

(iii) support for R/W data bursts,

(iv) memory partitioning,

(v) frame-based ring-buffers,

(vi) support for variable frame dimensions,

(vii) support for high-level addressing (units: partitions, frames, lines, pixels),

(viii) providing high-level status information.

Unlike other modules, there are no special demands on the physical layer interface, which is responsible for generating RAM clock signals, initializing the memory, transmitting data, and applying required refresh signals. Common memory interfaces, such as the Xilinx MIG [18], provide appropriate low-level services and are not subject of this article.

Although the projected memory abstraction is an important and far from trivial task, the major driven force for this work is to reduce the number of critical design paths emerging from constraints in the partial reconfiguration design flow [19]. The following subsection specifies these constraints and discusses in particular the impact on dynamically reconfigurable designs connecting an external RAM. In this regard, the problem is approached by a novel interface concept determining the further organization of the proposed memory controller.

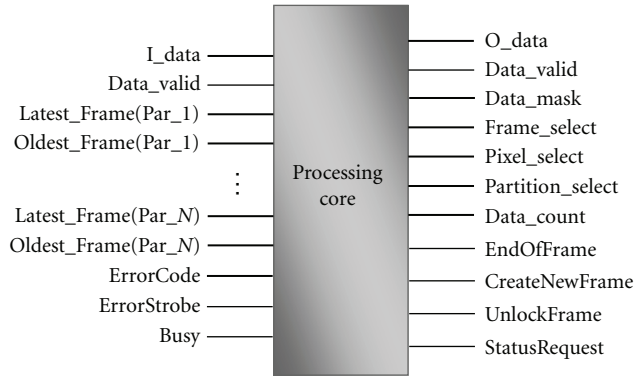


Figure 4: Client ports to memory controller (ver. 1).

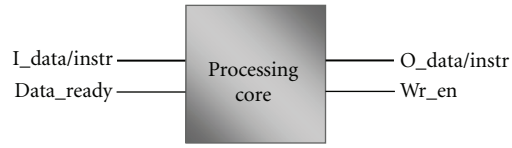


Figure 5: Client ports to memory controller (ver. 2).

3.1. Memory Controller Interface. Figure 4 exemplifies a client's I/O interface to a memory controller implementing the desired functionality. The client input signals (left) contain besides a data-vector moreover partition information

and reported error codes. Respective output signals (right) contain a write data-vector, corresponding mask, and high-level addressing signals. Depending on the memory size, the data widths and consequently the number of required port I/Os may vary.

The presented concept particularly targets designs with dynamically reconfigurable RAM clients. In order to facilitate proper run-time reconfigurations, port-signals crossing reconfigurable boundaries need to be routed through bus-macros (BM) [2, 5, 12, 19]. At this, the designer has to ensure that all BMs are placed directly on dedicated boundaries of corresponding regions. BMs are either implementable based on slice macros or on 3-state busses [12]. Note that, modern FPGAs, like the Xilinx Virtex-4, do not support internal 3-state busses anymore. Furthermore, slice-based macros are more efficient and easier to place. However, bus macros inherently impart routing restrictions affecting the signal timing. The client-RAM interface (CRI), as presented in Figure 4, occupies approximately 252 I/Os requiring 32 bus-macros for every reconfigurable client in a Virtex-4 implementation. The potential generation of critical paths as well as the time-consuming manual placement of the bus-macros motivates for a more efficient solution.

Figure 5 presents an optimized version of the memory controller's client interface with a drastically reduced set of IOs. Instead of using multiple channels to transmit addresses, data, and status information, only a single 17-bit wide bus and a corresponding control signal are required for each communication direction.

This significant improvement is achievable by sharing a bus for data and so called instruction telegrams. Instruction telegrams are part of a simple communication protocol and function as a container for remaining signals. They are decoded by the respective receivers and encompass the equivalent functionality as the complex unit in Figure 4. The most significant bit on the bus serves as a tag, distinguishing the two telegram types. Inside the memory controller, all data-/ instruction telegrams are buffered in asynchronous FIFOs, enabling concurrent access of multiple clients. Clients deploy the *Wr_en* signal to store data in respective FIFOs associated to their connection ports. Vice versa, the memory controller asserts the *Data_ready* signal as soon as data is available for readout. A thusly triggered client transfers an instruction telegram, signalizing its ready-state. Consequently, the memory controller initiates the readout of the corresponding FIFO and transfers available data.

Contents and sizes of instruction telegrams are different for either direction. Table 1 lists elements of instructions sent from clients to the memory controller. The "Size" column represents the required bits per instruction field.

Asynchronous FIFOs serving as data buffers between the memory controller and its clients do not necessarily have the same bus-widths and clock rates on both of their ports. Therefore, it is feasible to reduce the bus-widths of the interface (Figure 5) beyond the physical data width of the RAM. Instantiating the underlying architecture of Figure 3, clients read and write two pixels per clock cycle, resulting in a word width of 16 bits plus one tag bit. The controller obtains data in 8-byte words from the physical layer interface, defining its internal data bus width to 64 bits. Therefore, the memory controller inserts $k = 64/16$ tags into each data word written to the clients' read-FIFOs. The resulting size of instruction telegrams—for both directions—is 68 bits. Thus, clients require four clock cycles per instruction word while the memory controller needs one cycle only.

BlockRAMs of Xilinx FPGAs offer one parity bit per memory-byte, enabling proposed FIFO implementations without allocation of additional resources for telegram type tags. Contents of instruction words transmitted by the memory controller are primarily status information and listed in Table 2.

Entries of both tables will be self-explanatory as their processing logic is described in detail within the following subsections.

The proposed concept provides the desired functionality and minimizes the number of required bus macros. Nevertheless, the incurred protocol overhead increases the effective memory access time and represents a considerable drawback. Hence, particularly clients frequently addressing noncontinuous small data units are affected. However, due to the majority of processing cores preferring burst access to the memory, the protocol related latency, which does not exceed a few clock cycles, is negligible in most cases.

3.2. Client RAM Interface. In order to increase the concept's applicability, the RAM controller provides the interface on Figure 5 to all of its clients in the same manner. However, the reduced interface type leads to significant protocol-handling overhead inside clients. To counter negative effects, corresponding design parts are encapsulated and separated from clients' processing logic, according to Figure 3. Such client RAM interfaces simply connect to clients providing the original IO-set (Figure 4) and hide all protocol related operations.

The interface design is based on an RX-and TX-module, processing in parallel. The latter is presented in Figure 6 and is responsible for forwarding data to the memory controller as well as the encoding of instruction telegrams according to Table 1. As mentioned previously, in the presented design clients are writing to a FIFO inside the memory controller in words of 17 bits while the controller itself reads words of 68 bits. In this regard, the TX-module avoids telegram fragmentations by ensuring that always blocks matching the controller width are written into the memory controller. In the following, the architecture of the interface is explained in detail.

The TX-module contains two finite state machines (FSMs). FSM-1 handles clients' write requests and is furthermore triggered by unlock or create-new-frame (CNF) commands. The functions of commands are not relevant at this point and will be handled later in Section 3.5. Instruction telegrams are composed in states 1–4 and stored inside a FIFO. The FIFO is an essential component to buffer video data and write instructions during instruction word generations of FSM-2. A write request causes the video gate to open, enabling the client to transmit its data. FSM-1 remains in state 6 until the requested amount of data

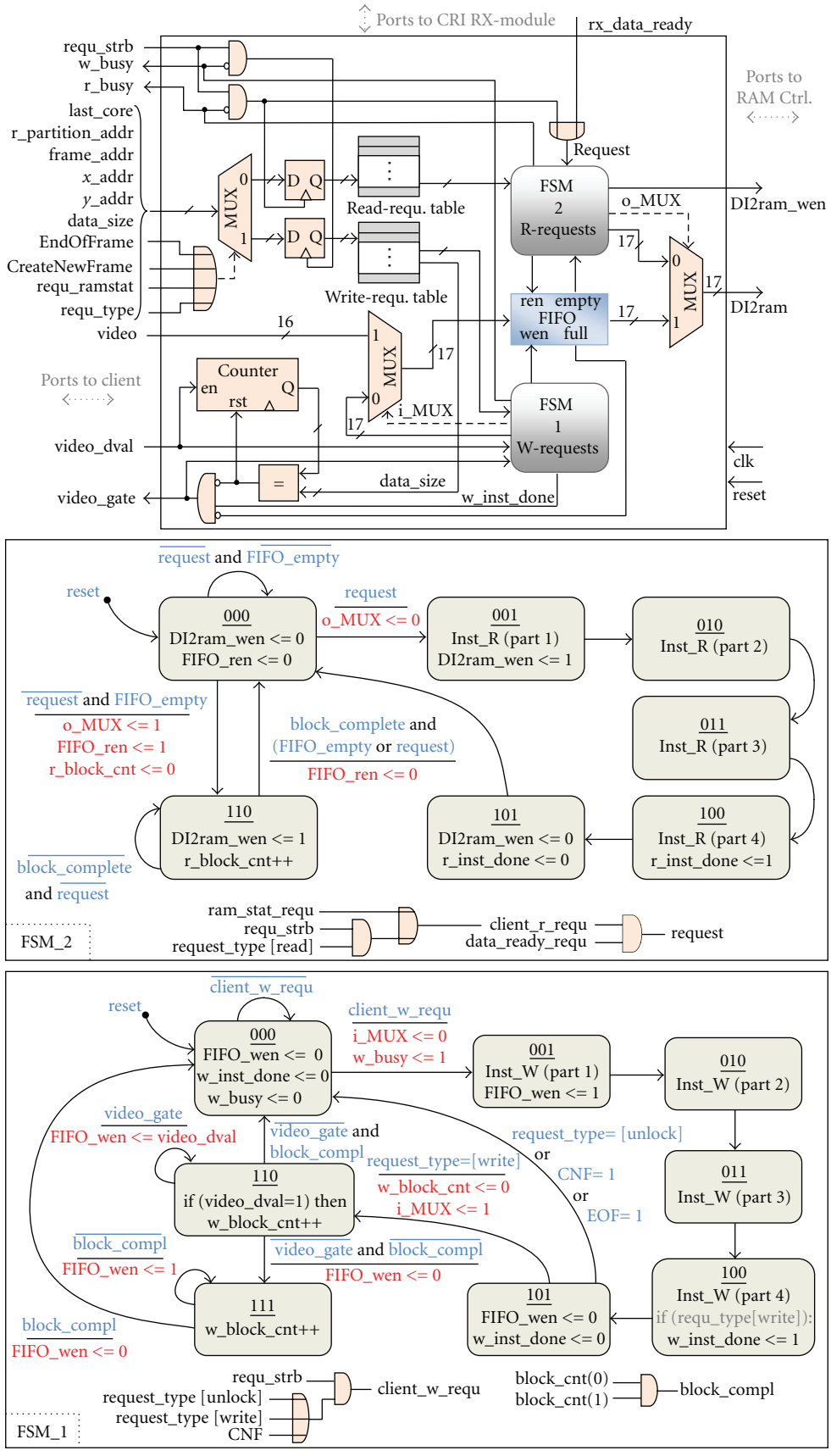


Figure 6: Client interface (TX-module).

has been buffered inside the FIFO, and the video gate is closed again. If the buffered data-size is not a multiple of the controller's input-buffer width, the data-block is complemented to avoid telegram fragmentations inside the memory controller. A busy signal informs the client when the interface is available for further write requests.

Although FSM-1 controls the execution of write requests, any kind of data is exclusively transferred to the memory controller by FSM-2. FSM-2 is triggered by a client's read or RAM-status request, and additionally when data is available for readout from the memory controller. In either case, FSM-2 composes an instruction and transfers it directly to the memory controller. Moreover, it is responsible to control the readout of the interface FIFO containing write instructions and corresponding data. In conformance with write requests, telegram fragmentations are avoided during FIFO readouts, respectively. The client RAM interface writes data to the memory controller in a deterministic order. Thus, video data is always preceded by a write instruction specifying the exact amount of data. Instruction telegrams with read or status requests are prioritized and may occur irregularly in the buffer, if supported by the memory controller.

The architecture of the RX-module of the client RAM interface is depicted in Figure 7. Complementary to the TX-module, it processes all data received from the memory controller. Receiving instructions and data is not obvious due to the lack of data valid signals. Hence, transmitted data blocks are encapsulated in two instruction words facilitating synchronization, according to Figure 8(a). Instruction telegrams are decoded by the FSM to obtain all information listed in Table 2. Received data blocks are forwarded to the client, and a corresponding valid signal is generated.

The memory controller transmits instruction telegrams replies either as on client requests or on frame-buffer content changes. In order to obtain solely valid status information, the RX-module exclusively extracts it from instruction telegrams with the "data valid" flag enabled. Thus, a pure status update requires an additional instruction telegram, respectively, disabling the update again. The corresponding protocol is illustrated in Figure 8(b).

3.3. Priority Arbitration. Concurrent requests of RAM clients make a suitable arbitration inside a memory controller indispensable. The achievable QoS, defined in Section 2.2, is strongly related to the arbitration algorithm. In order to satisfy various bandwidth requirements of clients the arbiter has to comply with priorities assigned by design automation software. The software, executed on the connected PC, is knowledgeable about the underlying hardware platform and defines a video processing scheduling plan according to user specifications [4]. Hence, bandwidth requirements and priority constellations can be determined in advance and are initially transferred to the frame grabber as parameters. Note that, due to the support of dynamically reconfigurable clients, priorities are not necessarily constant during a scheduling cycle. Thus, in order to avoid QoS limitations it is imperative that the arbiter provides priority updates during run-time, controlled by the embedded microcontroller.

Table 1: Instruction word: client → controller.

Number	Size	Contents
1	1	Telegram type tag (data/instruction)
2	1	Request RAM status
3	1	Last core in processing pipeline
4	1	Start readout of Rd_FIFO
5	2	Memory access request type (read/write/unlock/NOP)
6	2	Partition select
7	6	Frame select
8	1	End-Of-Frame (EOF)
9	1	Create-New-Frame (CNF)
10	16	Column address (x)
11	16	Line address (y)
12	16	Number of bytes to read/write

Table 2: Instruction word: controller → clients.

Number	Size	Contents
1	$k*1$	Telegram type tags (data/instruction)
		Partition 1:
2	6	index of oldest frame
3	6	index of newest frame
		Partition 2:
4	6	index of oldest frame
5	6	index of newest frame
		Partition 3:
6	6	index of oldest frame
7	6	index of newest frame
8	4	Error code
		Partition 4:
9	6	index of oldest frame
10	6	index of newest frame
11	1	Data valid

Round Robin is a frame-based scheduling algorithm, serving flows in cycles [20]. Each client gets at least one opportunity per cycle to read or write data. The weighted round-robin (WRR) algorithm facilitates the distribution of available throughput corresponding to clients' priorities [21]. Thus, clients may transmit a fixed amount of data in provided time-slots, making WRR perform well in terms of fairness. To increase the arbitration efficiency the visits of clients have to be spread evenly in time, considered in advance by the design automation software.

A lightweight design of an arbiter implementing the desired functionality is illustrated in Figure 9. In the following, implementation details are outlined making the principle of operation clear. Client RAM interfaces connect directly to the arbiter implementing FIFOs to buffer data streams in both directions. Besides RPUs, two additional types of clients with special demands exist within this project. First, the RX client—a write-only client, continuously delivering incoming video data—does not utilize an RX-FIFO

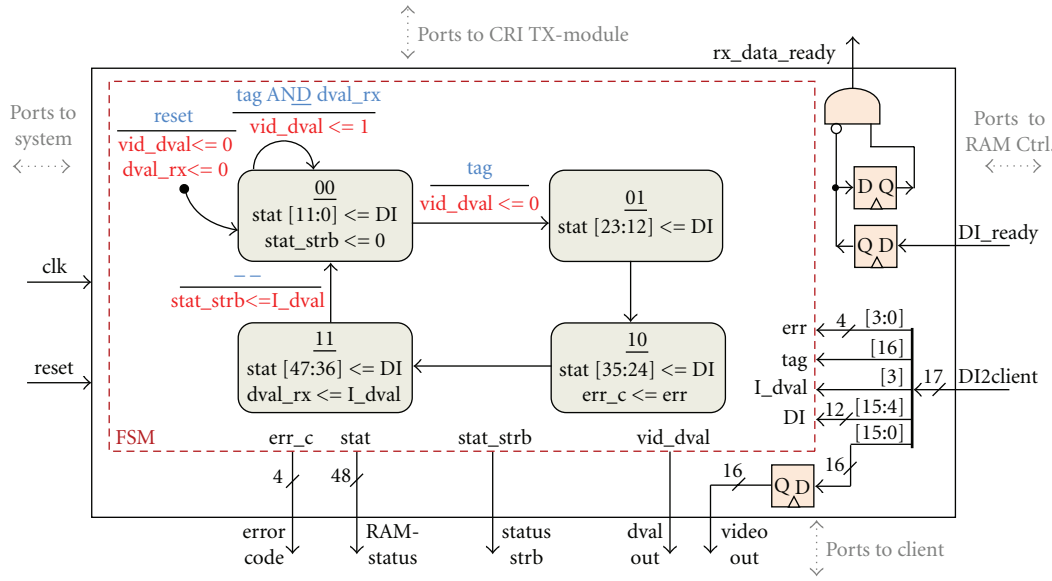


Figure 7: Client interface (RX-module).

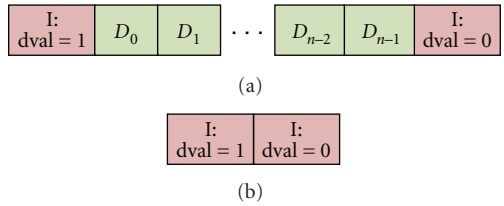


Figure 8: Transfer protocol.

on its port. In order to provide reliable frame acquisitions, this client always obtains the highest arbitration priority. Second is the TX client, a read-only client responsible to transmit the final processing results to a PC. Unlike the previous client, the TX-FIFO is not redundant here, as it is required to buffer read-out instruction telegrams. FIFO sizes determine the maximum allowed burst lengths of clients and are relevant for terms of real-time. Due to the line-oriented characteristic of a variety of image and video processing applications, appropriate buffering solutions are encompassed by choosing FIFO depths larger than the respective image line dimension.

The sequential characteristic of the arbitration process makes it natural to control it by a finite state machine. FSM-1 obtains the port-numbers from parameter registers according to the predetermined scheduling order. A timer is cyclically reloaded with the visit-time as the arbitration changes. If a client's TX-FIFO is empty, the arbiter immediately proceeds with the next scheduling element. Otherwise, an instruction plus optional subsequent data blocks (but no further instructions) is read out of the FIFO and passed to the instruction decoder. As soon as the instruction decoder finished processing, the sequence repeats until the visit-time elapsed. At this, a watchdog timer prevents the system from deadlocks if acknowledge signals exceed a certain time limit.

The second finite state machine receives status and error reports. It is responsible to encode instruction telegrams and to control all writing to clients' RX-FIFOs. Data received from the RAM is always internally buffered. FSM-2 controls the read-out and encapsulates containing data-blocks in two instruction telegrams, conforming to the protocol definitions. Furthermore, the address translator autonomously triggers broadcast status reports as contents in its frame-buffer table change. Corresponding instruction telegrams are created inside the arbiter and distributed to all clients.

The intended arbiter concept is characterized by a clear structure, devoid of redundancy. Though the achievable burst sizes depend on available FIFO capacities, the arbiter has no limitations regarding supported data dimensions. Thus, it facilitates a resource-efficient implementation and complies with the basic idea to support various video formats.

3.4. Instruction Decoder. The instruction decoder realizes the link between the arbiter and the address translator. Figure 10 presents an example implementation including a finite state machine depicted in Figure 11. It is responsible to decode client instructions and, furthermore, distinguishes supported request types for according instruction executions. Commands dedicated to the address translator—such as create-new-frame, unlock, end-of-frame, and status requests—are simply forwarded while read and write operations require adequate preprocessing. Thus, burst requests are unrolled, utilizing the implemented up-counter, in order to provide continuing high-level addresses to subsequent modules. Moreover, mask vectors are generated and output synchronously during write instructions.

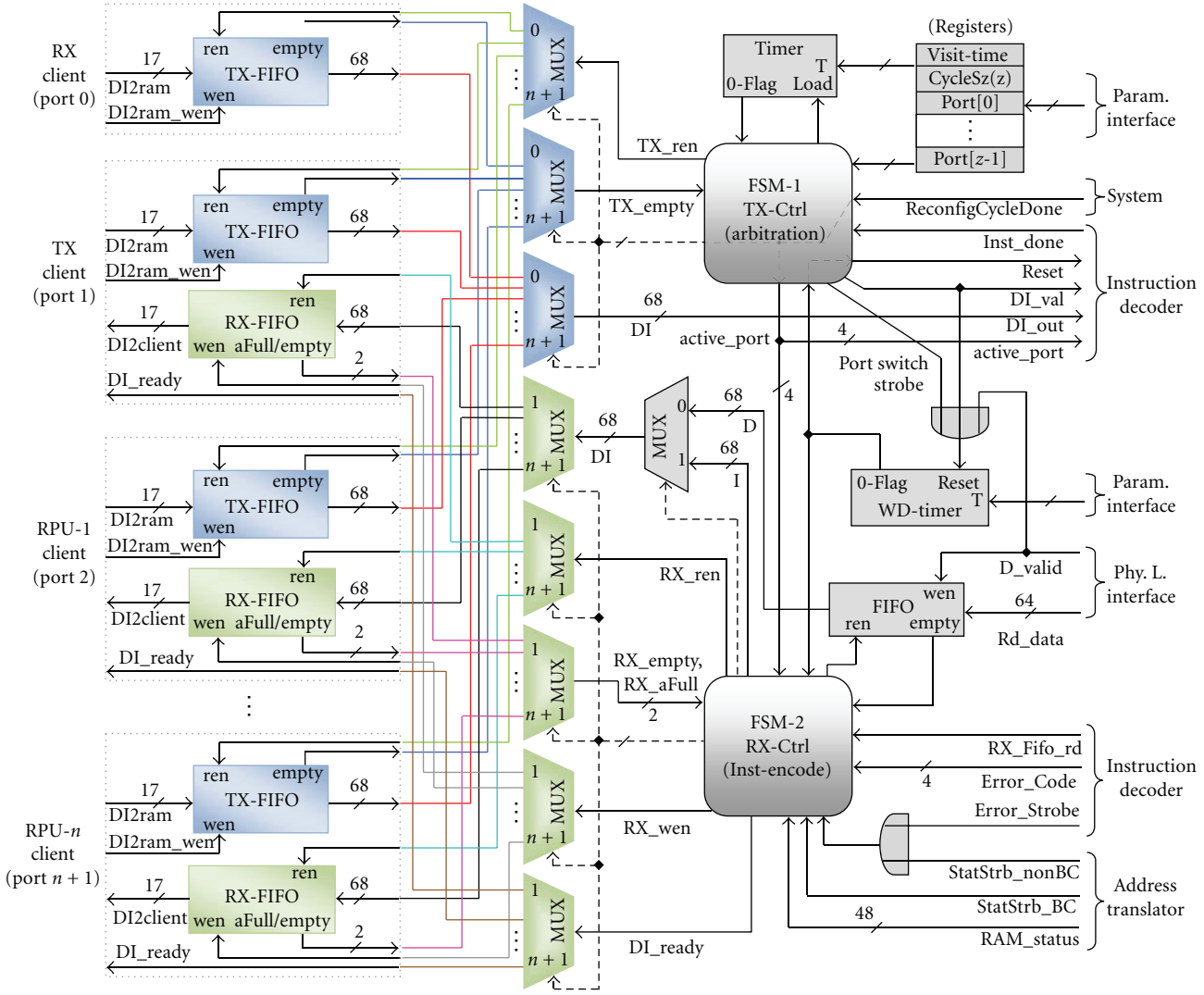


Figure 9: Arbiter architecture.

The instruction decoder features appropriate validation methodologies and provides corresponding error codes to the arbiter. In regard to write instructions a continuous stream of data, matching the requested size, is expected to be contained in the FIFO. The validation process of read instructions is more complex due to the fact that it relies on data consequently transmitted from the physical layer interface to the arbiter. Serving this purpose, an implemented down-counter is triggered by the received data valid signal and facilitates tracking of incoming data concurrent to address generation. The watchdog timer inside the arbiter implies to eliminate the eventuality of deadlocks within this state.

The proposed concept of the instruction decoder relies on a single finite state machine at the expense of parallelism in execution of RW-operations and forwarding of address translator commands. Respective improvements would impart further performance benefits for clients as demonstrated in Section 4.1.

3.5. Address Translator. The address translator undertakes the task to manage the physical memory in structures suitable for video processing applications. It maps high-level address descriptions in real-time to physical RAM addresses required by the physical layer interface, and it is responsible to provide services on higher abstraction layers to clients.

Video processing clients typically address data in units of frames, lines, and pixels. Due to multiple clients sharing a single physical RAM, support for partitions facilitates data integrity during write operations. Organizing partitions as frame ring-buffers provides access to the latest set of frames produced by particular clients and resolves the direct correlation between the totally required and physically available memory size. Owing to potentially nonuniform arranged output-data, especially in systems with dynamically reconfigurable clients, the address translator furthermore requires support for variable frame dimensions within these ring-buffers. In order to replicate a corresponding

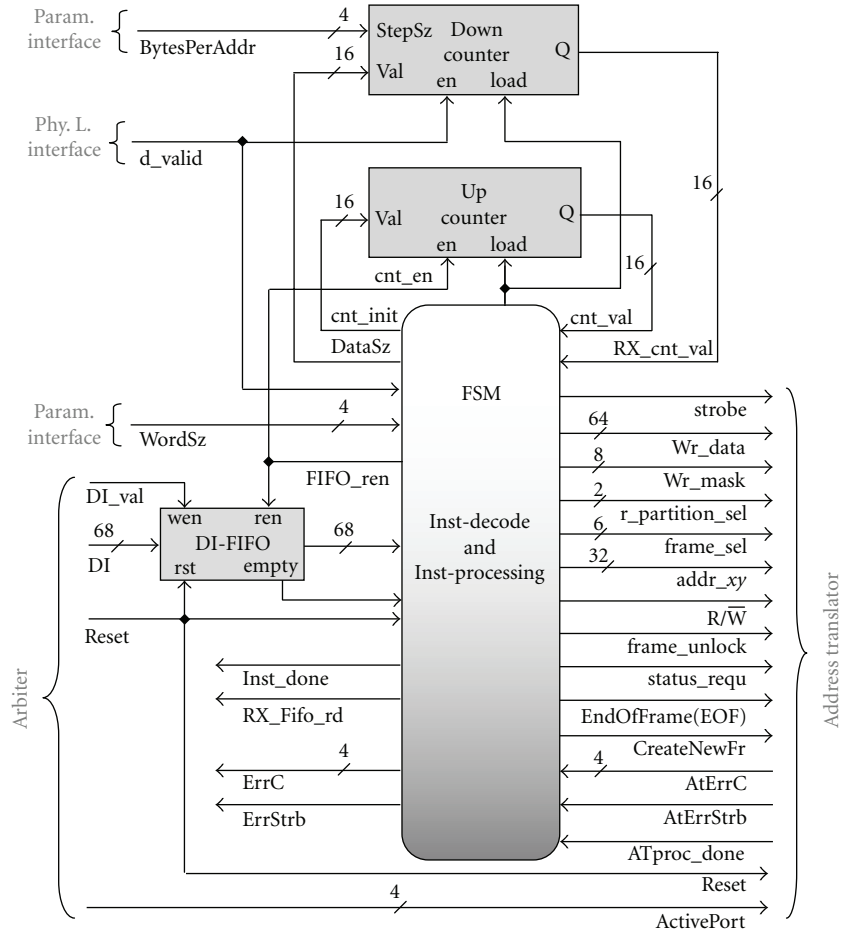


Figure 10: Instruction Decoder.

architecture in hardware, adequate status information has to be provided and managed for partition ring-buffers and containing frames. Status reports according to Table 2 make the availability of data transparent to clients. The transmission to the arbiter is triggered by either a request or autonomously when changes in partition tables occur.

The concept of the address translator is presented in Figure 12. Implementation details are outlined subsequently in order to establish better understanding of important aspects. The presented design supports up to four partitions with a maximum of 64 frames, respectively. BlockRAM memory inside the FPGA holds relevant status information for all frames:

(i) number of columns (16 bit),

(ii) number of lines (16 bit),

(iii) start address (16 bit),

(iv) end address (16 bit),

(v) empty flag (1 bit),

(vi) valid flag (1 bit),

(vii) closed flag (1 bit).

Start and end address entries define the partition internal location of frames, and various flags represent their current state. Create-new-frame (CNF) commands of clients allocate and determine new virtual frames for write access. To avoid conflicts caused by write requests of different clients to overlapping locations, partitions are not shared for write operations. Therefore, the embedded microcontroller initially assigns partitions exclusively to writing clients. This configuration is realized by the parameterization interface setting up the write-partition-allocation LUT inside the address translator. Read requests, in turn, are generally permitted to all partitions. The closed flag is assigned by an end-of-frame (EOF) command, indicating that a client finished writing to a frame. Consequently, broadcast status reports are triggered making the corresponding frame available to other clients. Nevertheless, the address translator supports unlock commands enabling further changes in previously closed frames.

A finite state machine serves the processing of time-noncritical tasks, such as client command executions and error code generation. Furthermore, it continuously updates frame and partition status entries. Latter are underlying status reports encoded by the arbiter. Partition information is implemented in registers, containing the following:

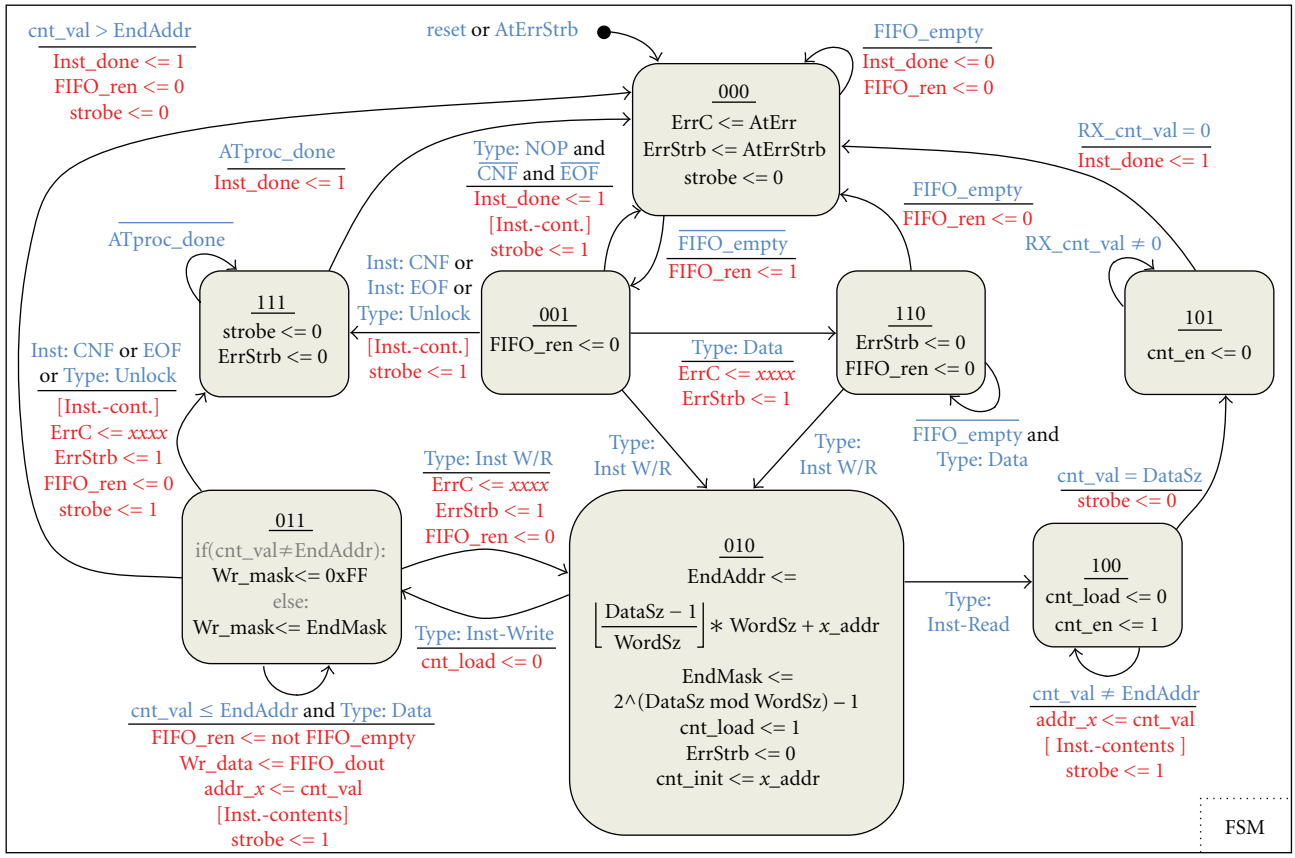


Figure 11: Instruction Decoder-FSM.

(i) size (bytes) (33 bit),

(ii) physical address (36 bit),

(iii) first (oldest) frame number (6 bit),

(iv) last (newest) frame number (6 bit),

(v) full flag (1 bit).

The full flag indicates that no free memory is available in a partition. However, since partitions are organized as ring-buffers, this does not imply that further write requests are prohibited but rather are old frames overwritten. Finally, the size and physical address entries are static parameters, dependent on the memory size and the number of connected clients.

The green highlighted function blocks in Figure 12 represent the calculation of the physical RAM addresses. Note that a pipelined architecture has to be implemented facilitating on-the-fly processing of incoming data from the instruction decoder. Finally yet importantly, the address translator provides versatile correlated error detections, too.

(i) Read attempts on empty or invalid frames (1).

(ii) Address exceeds frame boundary (2, 3).

(iii) Addr_x exceeds number of columns (4).

(iv) Addr_y exceeds number of lines (5).

(v) Write attempts to closed frames (6).

Error codes are generally forwarded to affected clients, facilitating coarse run-time debugging of both the memory controller and its clients. As processing cores configured in an application-specific scheduling pipeline are interdependent, the proposed error detection moreover establishes a fundamental validation of the system setup. For instance, owing to an incompatibility within the configuration queue, a client may expect intermediate results of its predecessor differently arranged. A consequential address violation emerging inside the address translator results in a corresponding error report, exposing the problem to the design automation software.

4. Case Studies

4.1. Flip-Image Sample Client. The relevance and implementation of the aspired memory abstraction within the proposed concept has been elaborated in previous sections. To point out the accomplished benefits from a clients point of view, two different sequences of operation are exemplified in Figure 13. At this, the complexity of the regarded task is not of relevance, as solely data-IO events are focused. Hence, the presented client simply intends to vertically flip buffered images. The statechart on the left depicts a process with sequential read and write operations. The memory controller periodically transmits broadcast status-reports

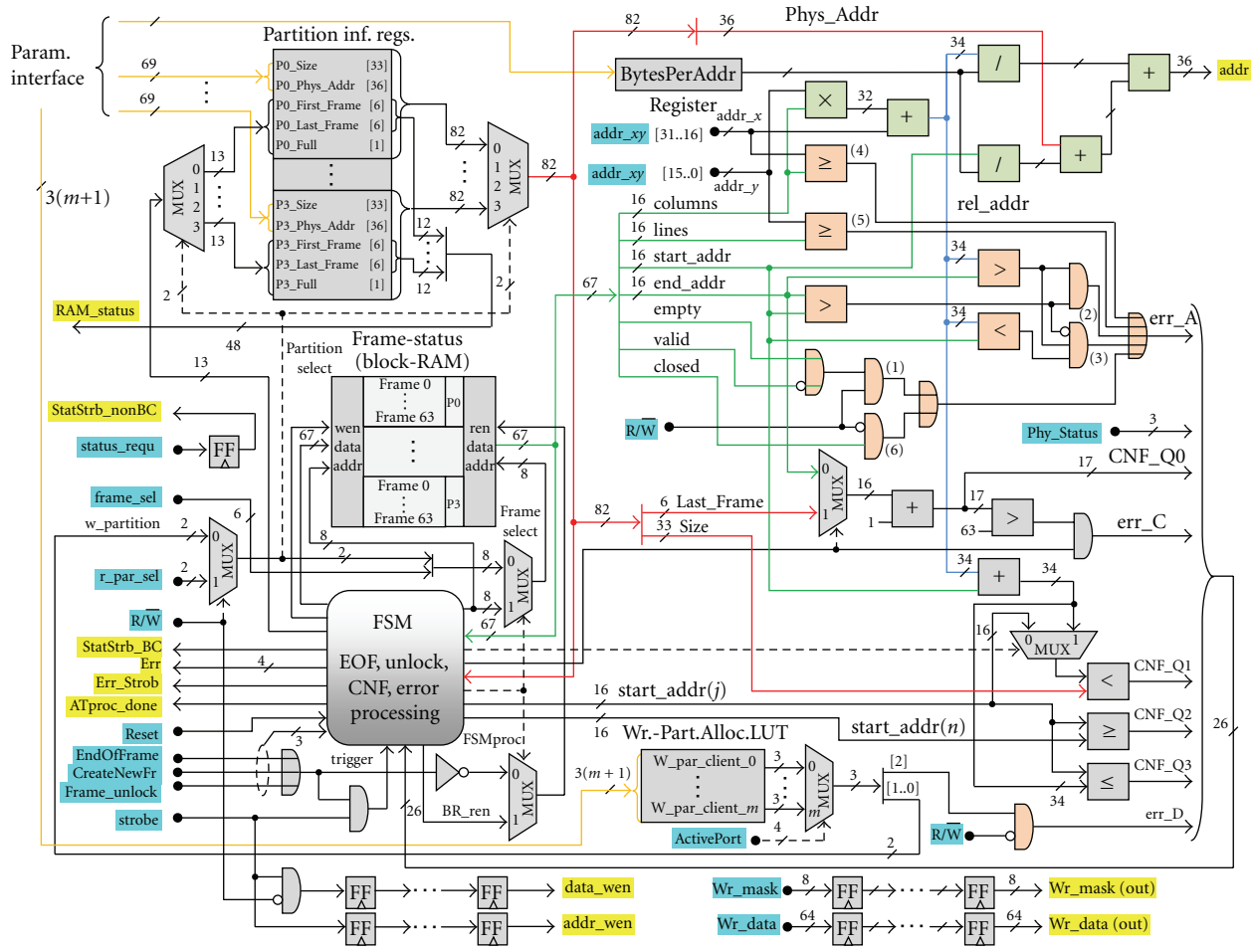


Figure 12: Address translator.

containing high-level addresses of buffered frames. Triggered by a new frame, the client initiates a CNF command to allocate RAM according the dimension of its output frame. Subsequently, the last line of the destination frame is requested and internally buffered as it is received. A write request, addressing the first line of the target frame, causes the client RAM interface to open the video gate, giving access to the corresponding arbiter FIFO. After all buffered data have been forwarded to the controller the procedure starts anew for the next line. Finally, a telegram containing an EOF command makes the created frame available to other clients and ends the process.

Yet, the example incurs the requirement for clients to buffer received data, causing increased processing latencies. In this regard, an optimized sequence, implementing parallel executions of read and write operations, is presented in the second statechart. This enhanced version intends to open the video gate before a read request is applied. Consequently, received data can be directly processed and forwarded without the need for additional buffers. Obviously the encompassed memory abstraction facilitates the desired uncomplex high-level access to a connected RAM and hides underlying memory organization tasks.

4.2. Traffic Analysis. Due to changing requirements of RPUs and varying input data-rates, general bandwidth demands regarding the proposed memory controller are unpredictable. Nevertheless, this section aims at establishing better understanding of relevant traffic aspects, finally instancing a certain case study. To begin with, general calculations of periodic transfer durations are provided for important intersections. As stated before, video data is typically subdivided into units of frames, lines, and pixels, including short gaps between frames and lines. Therefore, the given calculations refer to respective line transfers, representing decisive continuous data bursts. In order to provide numeric results, assumptions about the environment are made in the following.

Thus, assuming a camera producing 15 frames per second in a resolution of 2048 × 2048 pixels with an 8-bit color depth, the resulting incoming datarate is 60 MB/s. Within this case study, arbiter FIFOs are capable to store two image lines, and the scheduling is parameterized to serve one line per visit-cycle. Thus, incoming lines have cycle durations of

$$T_i = \frac{1}{f_{i\_frame} * \text{Lines}} = 32.55\,\mu s. \quad (1)$$

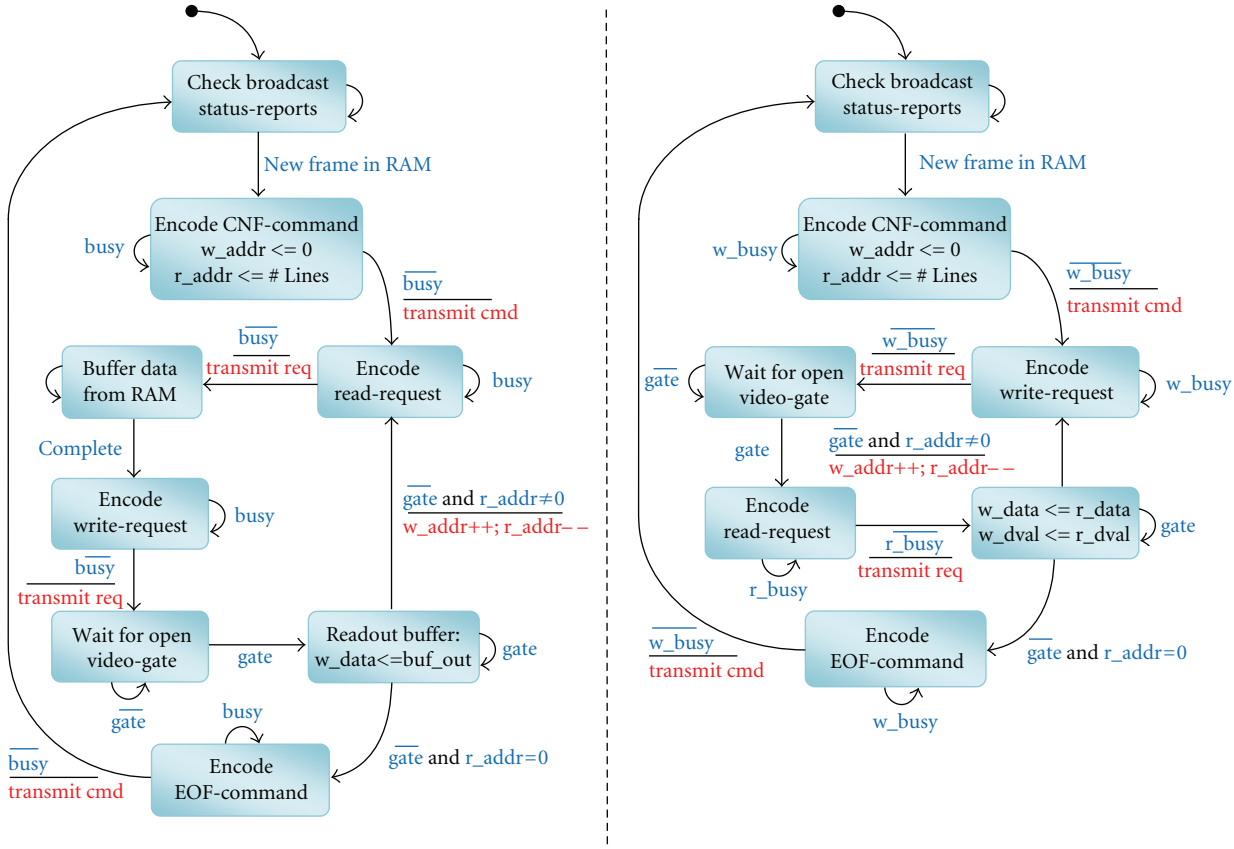


Figure 13: Client processing sequence: sequential (left), parallel (right).

The arbiter accesses included FIFOs in 8-byte wide words with a clock frequency of 100 MHz. Thus, the period for according line R/W operations results in

$$T_1 = \frac{1}{f_{\text{RAMclk}}} * \left(\frac{\text{Pixel/Line} * \text{Bits/Pixel}}{\text{BusWidth}} + 2 \right) < 2.6\,\mu\text{s}. \quad (2)$$

The bracketed expression in (2) represents the clock cycles required to transmit a single line plus the two data-block enclosing instruction words. The time after arbitration, required to perform transfers to or from the DDR RAM, calculates, respectively, as

$$T_2 = T_{\text{CtrlLat}} + \frac{1}{f_{\text{RAMclk}}} * \left(\frac{\text{Pixel/Line} * \text{Bits/Pixel}}{2 * \text{BusWidth}} + \text{RAM}_{\text{Lat}} \right). \quad (3)$$

In (3) it is assumed for simplicity that the desired data-blocks can be obtained in burst-mode without additional interrupting latencies. Furthermore, RAM_{Lat} is less than 27 clock cycles, reflecting the memory access latency preceding bursts. Any overhead-time required by the memory controller to decode and execute instructions is considered in T_{CtrlLat}. Thus, with a low-level DDR RAM bus-width of 4 bytes T_2 results in less than 2.9 microseconds Clients, in turn, are connected via a 2-byte wide bus operating at 100 MHz. Thus, data transfer in the magnitude of a single line requires

$$T_3 = \frac{1}{f_{\text{RPUclk}}} * \frac{\text{Pixel/Line} * \text{Bits/Pixel}}{\text{BusWidth}} = 10.24\,\mu\text{s}. \quad (4)$$

After essential transfer times have been determined, Figure 14 illustrates a traffic analysis for an according case study deploying two clients. A dynamically reconfigurable system with multiple mutually exclusive RPUs leads to decreased bandwidth demands. This is due to impossible operational pipelining among clients, preventing concurrent requests to the memory. Therefore, the presented case study does not imply dynamic reconfiguration, addressing more challenging traffic aspects. The time-slots visualized in Figure 14 conform to (1) to (4). Corresponding labels contain information about accessed frame and line-numbers, with n, m, and z representing frames of different partitions, respectively. Thus the first client processes lines of the three latest received frames, generating single output lines. These intermediate results are requested by the second client, storing single lines in its associated writing partition. The second client furthermore implements the final core in

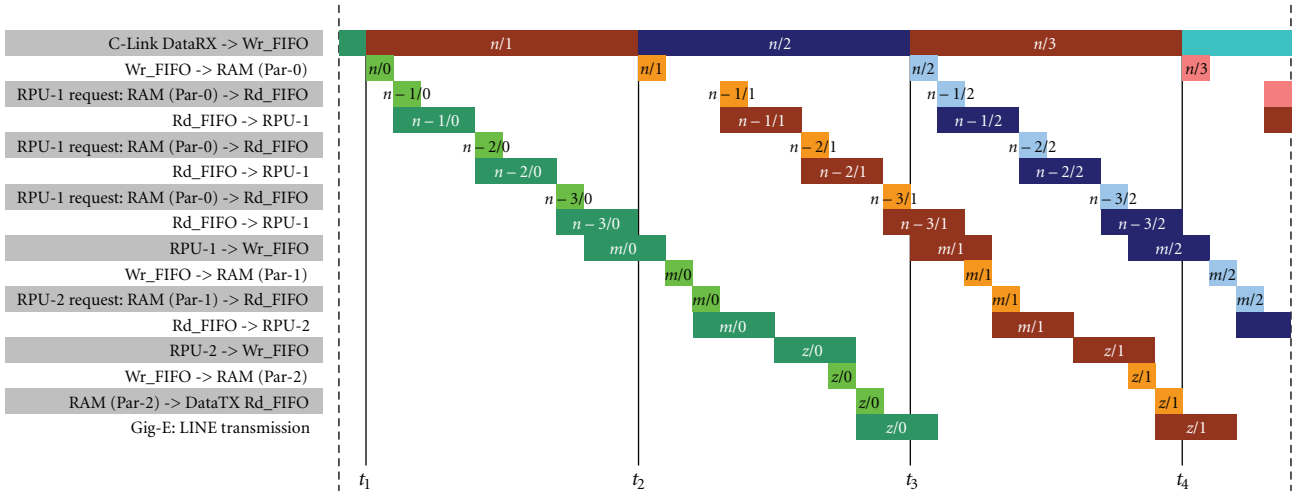


Figure 14: Traffic analysis.

the video processing pipeline. Therefore, its outputs are subsequently transferred to the connected PC. As a matter of principle, the arbitration grants only one client at a time access to the RAM, demonstrated in Figure 14. Finally, the overall latency in the regarded case study is approximately one frame and two lines, referring to the input data-rate.

5. Conclusion

This article presented a novel architecture of a memory controller satisfying special requirements of dynamically reconfigurable video processing platforms. At first, essential components and corresponding data paths of such a platform were specified. An adequate principle of reconfiguration, based on mutually exclusive RPUs, facilitates hiding of reconfiguration-times. In this context, application domain specific terms of real-time were regarded, and an appropriate QoS has been discussed.

Aiming at minimizing the amount of utilized bus-macros, it was a major objective of this work to define a uniform memory controller interface based on a reduced set of IOs. Thus, a corresponding interface-protocol, sharing a single bus for data and instructions telegrams, has been defined. As a result, the number of critical paths in dynamically reconfigurable designs decreases.

The article introduced a comprehensive concept of a memory controller providing adequate memory management abstraction. The proposed concept consists of four hierarchical modules and augments conventional SDRAM controllers focusing solely on low-level tasks. Architecture and implementation details were presented for all modules respectively to facilitate a deeper understanding of the nested structure.

A case study exemplified two client's sequences of events and pointed out according benefits due to the abstracted addressing scheme. Finally, a second case study implied general bandwidth calculations and provided results on abstract real time analyses.

References

[1] K. F. Ackermann, F. Mayer, L. S. Indrusiak, and M. Glesner, "Adaptable image processing system based on FPGA modular multi kernel instantiations," in *Reconfigurable Communication-Centric SoCs (ReCoSoC '06)*, 2006.

[2] Xilinx, Inc., "Virtex-4 Configuration Guide," 2007.

[3] S. Craven and P. Athanas, "Dynamic hardware development," *International Journal of Reconfigurable Computing*, vol. 2008, Article ID 901328, 10 pages, 2008.

[4] K. F. Ackermann, L. S. Indrusiak, and M. Glesner, "System level design of a dynamically self-reconfigurable image processing system," in *Reconfigurable Communication-Centric SoCs (ReCoSoC '07)*, 2007.

[5] K. F. Ackermann, B. Hoffmann, L. S. Indrusiak, and M. Glesner, "Enabling self-reconfiguration on a video processing platform," in *Proceedings of the 3rd International Symposium on Industrial Embedded Systems (SIES '08)*, pp. 19–26, Montpellier, France, June 2008.

[6] Xilinx, Inc., "Virtex-5 Family Overview," 2008.

[7] Altera Corp., *Stratix IV Device Handbook*, vol. 1, 2008.

[8] B. Donchev, G. Kuzmanov, and G. N. Gaydadjiev, "External memory controller for Virtex II Pro," in *Proceedings of the International Symposium on System-on-Chip (SOC '06)*, Tampere, Finland, November 2006.

[9] S. Heithecker, A. do Carmo Lucas, and R. Ernst, "A mixed QoS SDRAM controller for FPGA-based high-end image processing," in *Proceedings of the Signal Processing Systems (SIPS '03)*, August 2003.

[10] S. Heithecker and R. Ernst, "Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements," in *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pp. 575–578, Anaheim, Calif, USA, June 2005.

[11] Z. Zhou, S. Cheng, and Q. Liu, "Application of DDR controller for high-speed data acquisition board," in *Proceedings of the Innovative Computing, Information and Control (ICICIC '06)*, 2006.

[12] M. Hübner, T. Becker, and J. Becker, "Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration," in *Proceedings of the 17th Symposium on Integrated Cicuits and Systems Design (SBCCI '04)*, pp. 28–32, Pernambuco, Brazil, September 2004.

[13] D. R. Lee, S. W. Lee, and J. W. Jeon, "Frame grabber circuit for IEEE1394 image transfer," in *Proceedings of the International Conference on Control, Automation and Systems (ICCAS '07)*, Seoul, Korea, October 2007.

[14] A. Warkentin and F. Dittmann, "Data transfer protocols for a two slot based reconfigurable platform," in *Reconfigurable Communication-Centric SoCs (ReCoSoC '06)*, 2006.

[15] PULNiX America, Inc., "Specifications of the camera link interface standard for digital cameras and frame grabbers," 2000.

[16] R. Steinmetz, *Multimedia Technology*, Springer, New York, NY, USA, 2nd edition, 1999.

[17] H. Eeckhaut, H. Devos, P. Lambert, et al., "Scalable, wavelet-based video: from server to hardware-accelerated client," *IEEE Transactions on Multimedia*, vol. 9, no. 7, pp. 1508–1519, 2007.

[18] Xilinx, Inc., "Xilinx Memory Interface Generator (MIG) User Guide," 2008.

[19] Xilinx, Inc., "Early Access Partial Reconfiguration (EAPR) User Guide," 2008.

[20] Z. Uykan, "A temporal round robin scheduler," in *Proceedings of the 68th Semi-Annual IEEE Vehicular Technology (VTC '08)*, Calgary, Canada, September 2008.

[21] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, pp. 1265–1279, 1991.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 923415, 11 pages
doi:10.1155/2009/923415

Research Article

Experiencing a Problem-Based Learning Approach for Teaching Reconfigurable Architecture Design

Erwan Fabiani[1,2]

[1] *Université Européenne de Bretagne, France*
[2] *Université de Brest and CNRS, UMR 3192 Lab-STICC, ISSTB, 6 Avenue Victor Le Gorgeu, 29200 Brest, France*

Correspondence should be addressed to Erwan Fabiani, fabiani@univ-brest.fr

Received 31 December 2008; Revised 14 June 2009; Accepted 28 August 2009

Recommended by Peter Zipf

This paper presents the "reconfigurable computing" teaching part of a computer science master course (first year) on parallel architectures. The practical work sessions of this course rely on active pedagogy using problem-based learning, focused on designing a reconfigurable architecture for the implementation of an application class of image processing algorithms. We show how the successive steps of this project permit the student to experiment with several fundamental concepts of reconfigurable computing at different levels. Specific experiments include exploitation of architectural parallelism, dataflow and communicating component-based design, and configurability-specificity tradeoffs.

1. Introduction

When teaching "reconfigurable computing" (RC), a coarse distinction should be made between two different aspects:

(1) reconfigurable architecture programming: knowing the fundamental steps for programming, being able to use languages and environments to produce a configuration, usually targeting an FPGA; these skills can be divided into technical and conceptual ones;

(2) reconfigurable architecture design: understanding and designing the "reconfigurable" nature of an architecture.

The difficulty for the first aspect is to manage the distribution and the interactions between the technical and conceptual parts. If the focus is mainly directed at technical skill development, then students would not have the ability to adapt their experience to other languages and environments and to reuse programming strategies.

Conceptual aspects include using programming models to ease the design of an ad hoc component structure for a specific application and to be able to apply parallelization algorithms to improve performance. If the conceptual aspects are favored, then the students eventually would not see their advantages, since they do not experiment the difficulties of programming and achieving performance objectives without it. Moreover, a lack of technical skills in their curriculum may be a disadvantage for employment. Also, technical skills may be necessary to optimize a program issued from a high-level synthesis tool.

The second aspect induces the need for generic tools to design, program, and simulate a reconfigurable architecture. If focus is put on physical design (basic configurable blocks and routing channels), there are academic frameworks like VPR [1] or Madeo [2] that permit definition of a reconfigurable architecture of various granularity, with generic synthesis, place and route tools that act on any design description. However such a design level may be too much advanced and difficult to be comprehended by software CS students. Moreover, it does not answer the question: how to teach design of a parallel architecture (and not how to teach expressing an architecture specification as a synthesizable description)?

The course presented here aims to be at the interface between RC programming and RC design, without being as exhaustive as other courses (e.g., [3, 4]). It gives students some basic skills and knowledge on reconfigurable computing, in a limited time, by designing a reconfigurable

coarse grain datapath architecture for an application class of image processing algorithms and implementing it on an FPGA. The design methodology is similar to intermediate level component design [5].

Our pedagogical approach relies on "problem-based learning" [6]. The goal is that skill and knowledge acquisition should be induced from the realization of a concrete project by the students.

The course context and pedagogical objectives are detailed in Section 2. Section 3 is focused on the "teaching by project" approach applied to our course. Sections 4, 5, 6, 7, and 8 follow the design methodology, respectively, defining the application model, deducing an architectural model, implementing it on FPGA, and reusing the architecture design pattern via object modeling.

2. Course Context and Objectives

2.1. Course Context. This project is included in a CS master course (1st year) on parallel architectures. Scope of this course includes knowledge of basic parallel architecture models, ability to identify in an application the different kinds of potential hardware parallelism, programming for various parallel computing models, and experiencing the design of a specialized, reconfigurable or programmable architecture, using automated methods for regular architecture synthesis.

Usually, practical work is based on lab sessions with a limited focus, followed by an evaluated project with a "closed subject": the teacher gives a very detailed specification which should plan all aspects, and oral explanation is provided to help the students to understand the subject. There are two main disadvantages of this kind of project: students do not participate in the design and the project could lead to "making the students do exactly what the teacher has defined and potentially penalising the students for not precisely adhering to this."

However, considering an "open subject" project instead induces other difficulties. An "open subject" means that a significant part of the design of the solution is a result of the student's work; thus the subject is opened to various solutions. It is difficult to define an adequate level of difficulty of the project, and the risk of "off topic" answers is increased. It is also difficult to manage evaluation criterion.

Consequently, in order to facilitate knowledge and skill acquisition by students in a limited time, most of the course is based on a participative project (problem-based learning) where students learn by participating to design a solution to a problem. Problem-based learning is known to emphasize not only the final solution but also the method used to design this solution [7].

The project begins with a subject which is opened to various design alternatives, instead of having a highly detailed and structured subject that would make students just implement and not design a solution. Even if the teacher knows which design approach may be the best one to study, it is important that the students feel that they reach a solution by themselves and by cooperation. The teacher helps and directs students to express complete specifications. This kind of project also facilitates the activation of previous knowledge (previous courses and self acquired knowledge), helping to place the subject in a more general context.

Since 2006, practical work has been based on design implementation using FPGAs. We have chosen the Handel-C language [8] for several reasons. First of all, in the student's curriculum, a previous course on distributed algorithms, introducing synchronous and asynchronous modeling, uses a CSP- (Communicating Sequential Processes-) based language (*kroc* [9]) for specifying and simulating algorithms. Students are by this way familiar with basic programming primitives of a CSP model: system organization with communicating processes, expression of parallelism (*PAR statement*), reading (*? statement*), and writing (*! statement*) in synchronous communicating channels. Secondly, the high-level language structure and the C syntax make the apprenticeship of Handel-C faster for CS students than standard hardware design languages.

Concerning the reconfigurable computing platform, we have chosen the RC10 board from Celoxica [10], which includes a Xilinx Spartan 3L-1500 (containing 26 624 LUTs and flip flops). VGA output and camera peripherals included in the RC10 board permit easy development of real-time video image processing, which is one of the major application domains of reconfigurable computing [11]. Although the course is not image processing oriented, the requirements for the hardware platform are similar to [12]: those kinds of applications are a motivating context for students.

Concerning the toolset used, it is divided in two parts.

(i) The DK design suite [13] provides the Handel-C language and EDA toolsuite for design capture, compilation and synthesis of a design into an EDIF netlist. It permits the compilation, and synthesis of an Handel-C program into an EDIF netlist. Prebuilt libraries are available to abstract board drivers (camera, VGA display, flash memory, etc.).

(ii) The Xilinx ISE design suite [14] is used to generate a bitstream from the EDIF netlist, through map, place, and route processes.

2.2. Project Objectives. The pedagogical objectives of this course are:

(i) to improve technical skill in programming FPGA and experiment the development cycle of a configuration: high-level description, synthesis, map, place, route, bitstream generation; simulation is not addressed in this course and is studied during the 2nd year of the master;

(ii) to experiment the different concepts of architectural parallelism;

(iii) to experiment the partitioning of an architecture into various components and their design, to feel the tradeoff and choices between flexibility, specificity, and efficiency;

(iv) to understand the advantages of object modeling for productivity;

(v) to look at solutions fixing a problem, to present them, to compare them, and to select the best solutions relating to fixed constraints.

Consequently, the choice of the project subject has been a reconfigurable architecture that exploits data flow parallelism and control parallelism, as shown in the following sections.

3. Course Development

3.1. Global Scheduling. The project has been introduced, defined, and carried out in several sessions.

(i) The preliminary courses are two lectures (2 hours) on parallelism and parallel architectures and one lecture on FPGA, reconfigurable architecture, and Handel-C.

(ii) The first supervised practical work (2 hours) aims to highlight the main problems or questions. It is based on the initial subject and is composed of the following steps:

(1) individual reading and analysis of unknown notions and most difficult points;
(2) group working (5 or 6 students): merge of individual reflections, mutual help based on personal experience and knowledge, and selection of most important points to be solved in the project; each point is categorized into lack of knowledge, technical skill, or design reflection;
(3) the whole group questions are orally discussed and classified in similar themes, in order to arrive at a consensus on the main tasks: points to be solved or knowledge to be acquired (six tasks);
(4) the course ends up at assigning one task to each student of a group, who must prepare a presentation about it for the next session.

(iii) The second directed work course (2 hours) is a synthesis of the student's work:

(1) presentation of the student's assigned work: synthesis on a theme or reflection on a problem;
(2) critical analysis of each presentation, questions and answers;
(3) the session ends up at a consensus for solving each point;
(4) following the session, the teacher edits a synthesis document of all discussed points, that becomes the specification reference for developing the project.

(iv) Project labs and tutorials permit the student to take the environment in hand and to acquire the basic technical skills needed to begin the project:

(1) Lab 1 (2 hours): first contact with language and environment, exercises for video acquisition and display, and pipeline for simple pixel processing;
(2) Lab 2 (2 hours): exercises on flow parallelism and data parallelism;
(3) Lab 3 (2 hours): design of a FIFO unit for moving window (neighborhood).

(v) After students have completed the project, they are required to complete an anonymous questionnaire mainly focused on their evaluation of the "problem based learning" approach used. This questionnaire permits to have a student advice on various aspects of the project. For instance, questions can be related to group management, evaluation of the contribution of a student in his group, and evaluation of the benefits of the pedagogical method for knowledge acquisition.

3.2. Initial Subject. The submitted project addresses the design of a coarse grain datapath reconfigurable architecture well suited for a family of image processing algorithms and that can be implemented on an FPGA. Unlike most image processing FPGA implementations (as an example, implementations in Handel-C [15, 16], or for teaching purpose [12]), the architecture is "reconfigurable" in the way that once the bitstream is loaded, a configuration (list of symbols) is read from the flash memory and defines the behavior (taken in a limited set of function) of the units involved in the computations. It is similar to the "configurable window processor" architecture of Torres-Huitzil and Arias-Estrada [17]. However it remains at a lower level than SIMD instruction customization [18]. This configuration is set by the user prior to FPGA reconfiguration and is defined as a list of symbols.

This project was conceived for teaching purpose, but it is not far from real needs. For instance, designing such architecture permits to have a single bitstream that can be used to compute various image processing applications. It is comparable to IP core; or processor core hence it is not specific neither programmable but reconfigurable. Consequently an end user with no knowledge in FPGA programming can easily specify the behavior of his circuit, given the restricted application model defined. This configuration model permits to avoid synthesis, place, and route process for each specific behavior implemented.

On the other hand, such an architecture would obviously be both more resource consuming and both have lower clock frequency than a specialized design. However those disadvantages should be moderated by the real time video processing context: having the higher clock frequency is not the goal; the circuit should just be fast enough to have a satisfying frame rate. Moreover, camera and display peripheral drivers use functions that limit the maximal reachable frequency. Thus the real tradeoff can be considered to be between flexibility and area.

The application-class target of the architecture is spatial filtering for image processing where the new value of a pixel is computed depending on the values of its neighbors (here we consider a 3×3 neighborhood). Typical applications of this kind are morphological operators, convolution filters, and image segmentation, detailed in Section 4.

Concerning input/output timing constraints, at each system cycle an input pixel is read and an output pixel is written. A system cycle requires one or more clock cycles. The latency for computing a pixel is not constrained: this allows maximal level of pipelining. Moreover the architecture must include the memory resources needed for pixel buffering.

3.3. Student's Work. The first session ended up with the list of following points to study:

(1) architectural model: partitioning the architecture into units (computation units, I/O, configuration controller, memory unit);

(2) application model: doing a research on application model (context of use, typical convolution matrix);

(3) application/architecture appropriateness: enumerating the needed functions on the architecture to implement the different kinds of applications and giving typical examples;

(4) programming model: defining the format used to specify the behavior of the computation unit, and how the computation module will be configured;

(5) memory component of the computation module: I/O, size, structure;

(6) typing: size of types in the computing module and in the control unit.

Consequently, each student has to study one specific point for the second session. During this session, students present their ideas. For each point, ideas are compared, discussed, and a choice is made, with the help of the teacher if needed. These ideas and choices are the basis for the synthesis document edited by the teacher, which the students must follow for the implementation of the project. The following Sections 4 and 5 present a synthesis of the results of those studies, respectively, for the application model (point 2) and the architectural model (points 1,3,4,5).

4. Application Model

The application model includes morphological, filtering, and segmentation operations.

Morphological operations are, for instance, the following:

(i) dilatation and erosion, that, respectively, assign to the center pixel the maximum or the minimum of the significant neighboring pixel values identified by a mask;

(ii) pattern recognition, that recognizes a specific pattern, given a mask that specifies if a pixel must be black, white, or could have any value, in a binary image.

Filtering operations are based on convolution (sum of products with varying coefficients), eventually followed by a division.

(i) Smoothing filters are used for blurring and noise reduction. The most known filter is the mean filter (coefficient 1 for all pixels and division by 9) or the Gaussian filter.

(ii) Sharpening filters accentuate the details of an image contour (in fact a subtract of the blur image from the initial image). As the result can be negative, it is eventually necessary to add a fixed value or to take the absolute value.

Image segmentation operations enhance characteristic areas of the image, for instance, edge detection, Prewitt filter, Sobel filter, and Laplacian filter. The computed values may be negative; then an absolute value operation is needed.

Moreover, several operations can be sequenced, to accentuate the effect by repeating the same processing, or to compose a new processing, for instance an opening (one erosion followed by dilatation, to diminish noises) or a closing (one dilatation followed by an erosion, to smooth the form contours).

5. Architectural Model

The basic element of the architecture is a module. A module has one input and one output that are streams of pixel values: pixel coordinates are not transmitted as we assume that pixel order follows a raster scan order (column by column and line by line). The latency between a pixel input and the corresponding pixel output depends on the image width and the number of parallel components that are crossed by a pixel. In cruising speed, a module receives and sends one pixel at each clock cycle.

A module is composed of various units in a pipeline; each of them implements a different class of functionality, each being "configurable"; that is to say its computation behavior is specified by the user and setup before the computations begin. All units operate in parallel and simultaneously, on different instances of pixels in the stream. Thus, data flow parallelism is fully exploited. However the clock cycle is limited by the function with the longest delay. For a more efficient architecture, function delay estimation and node fusion would be necessary.

Modules are designed to be easily interconnected to each other by abutment in order to implement a more complex processing. Scalability is guaranteed by the fact that linking two modules would not increase the critical path, similarly to regular systolic arrays. The set of configurable units defines the programming model: it should be sufficient for describing the implementation of the chosen application model and can be extended.

Figure 1 presents the integration of 3 modules in the experimental context: pixels are generated by the camera controller and sent to the first module. Several modules are chained, using the same interface (one input pixel and one output pixel). The last module sends the output pixels to a frame buffer controller, which store in an embedded block ram the content of a complete frame and display it via the VGA output.

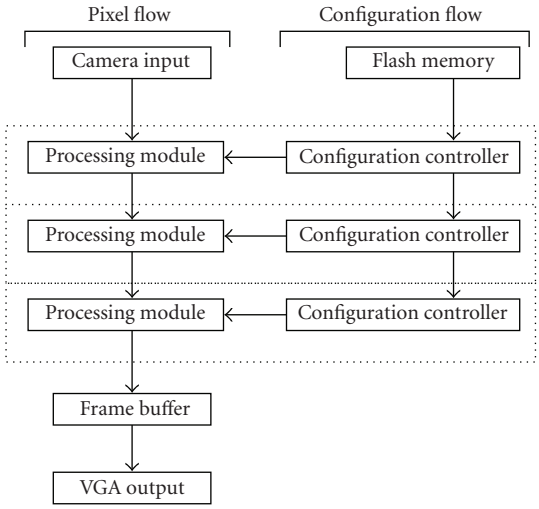


Figure 1: Integration of an array of 3 processing modules in the experimental context.

A configuration controller is used to configure the architecture. In case of several modules, controllers are chained in order to avoid the delay increase and simultaneous access problems that a parallel access of controllers to flash memory would induce. After reading its own configuration, the controller of the first module continues reading configuration file and sending configuration stream to next controller. The architecture is configured only once after the bitstream is loaded, but run-time reconfiguration would also be possible.

An image processing module is composed of various units (see Figure 2): units have a fixed behavior or have configurable parameters or operators. The various units that form the module have been defined by analyzing the application model and extracting common functionality patterns.

(i) Unary identical preprocessing units: they input one pixel and output one pixel (see Figure 3). These units perform an identical processing on all pixels, for instance, thresholding (with a parameter), inverse video, and format conversion.

(ii) Moving window unit: it inputs one pixel and outputs 9 pixels (see Figure 4). This unit memorizes the pixels in order to output a neighborhood at each clock cycle. It is a classical "moving window" approach, where pixels are stored as long as needed (a pixel is received one time and is used for 9 computations). So the moving window corresponds to a $3 * 3$ matrix that regularly goes over the image, left to right and top to bottom. Concretely this unit is made of a shift register of size $2 * W + 3$, W being the number of pixels in an image line. This unit has an initial latency of $W + 3$ clock cycles, that is to say the number of cycles between the input of the first pixel and the output of the first pixel neighbor. Image borders have a configured fixed value.

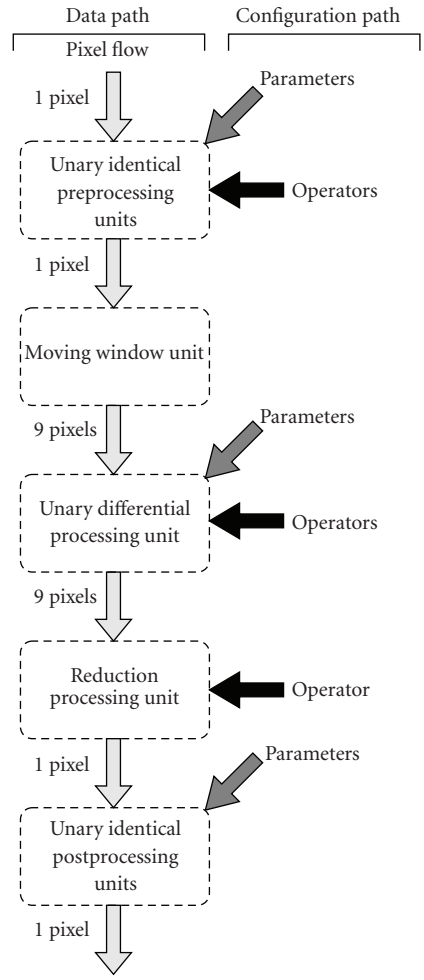


Figure 2: Decomposition of the image processing module into configurable units. On the right-hand side, configuration possibilities are expressed for each unit.

(iii) Unary differential processing unit: it inputs 9 pixels and outputs 9 pixels (see Figure 5). This unit applies a potentially different processing on each pixel of the neighborhood, for instance, multiply by a constant, test of value (black or white), and fixed assignment of value (black or white), *nop*.

(iv) Reduction processing unit: it inputs 9 pixels and outputs one pixel (see Figure 6). This unit is a tree of nodes processing simultaneously the same functionality on different values, to implement a global reduction function on the neighborhood, for instance, addition, minimum, maximum, bitwise *and*, bitwise *or*. Some nodes in the tree are *nop* nodes that are used to synchronize values that belong to the same flow generation.

(v) Unary identical postprocessing units: they input one pixel and output one pixel (see Figure 7). These units

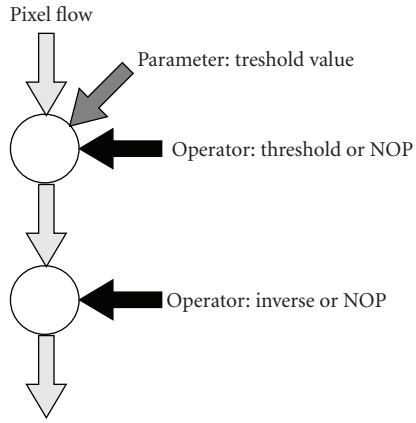


Figure 3: Unary identical preprocessing units.

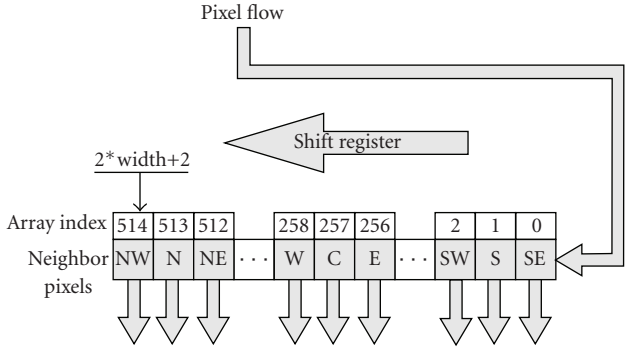


Figure 4: Moving window unit.

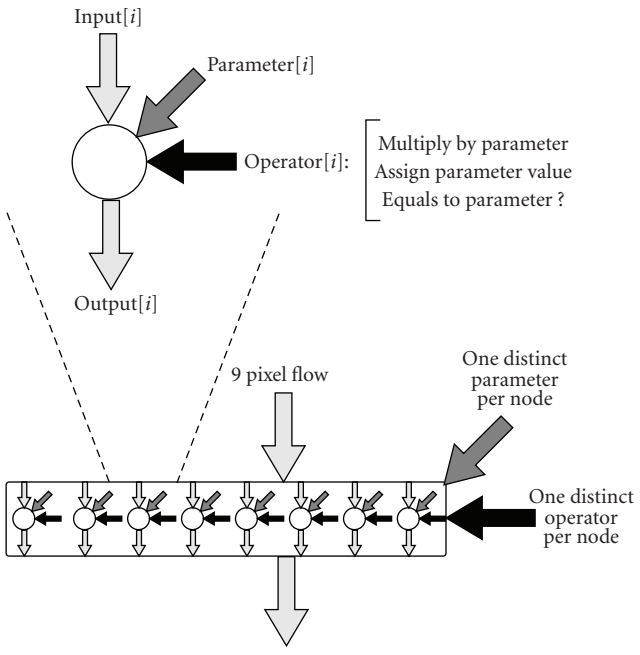


Figure 5: Unary differential processing unit.

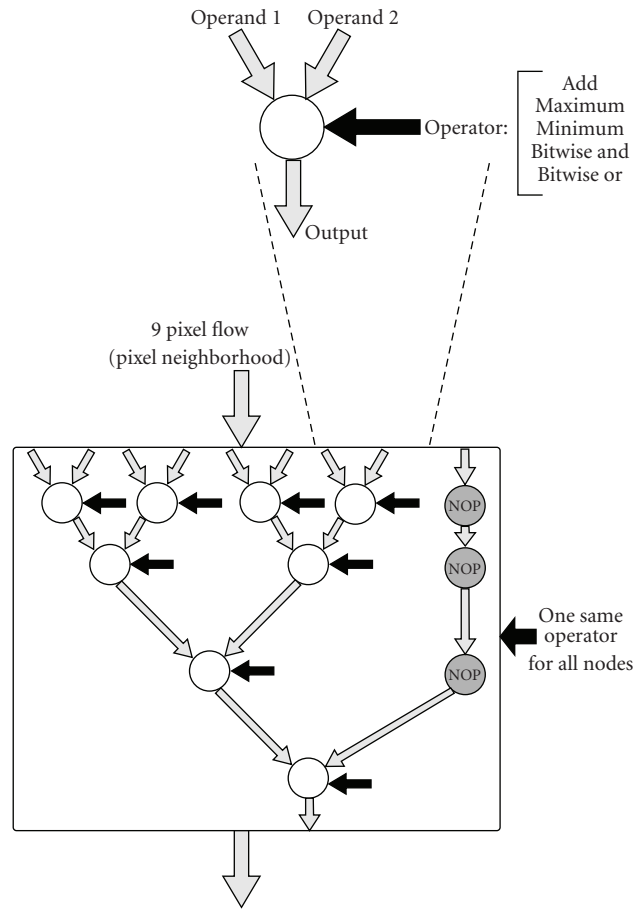


Figure 6: Reduction processing unit.

perform an identical processing on all pixels outputted from the previous unit, for instance, division, absolute value, adding a constant, and saturate. The last unit is not configurable and always computes the normalization of the resulting pixel value.

In addition to the parallel functioning of all units, units have also an internal pipelining: inputs/outputs are processed simultaneously with computation. So at a given time, a unit acts at least on three different pixel generations.

It is clear that the reconfigurability area cost is high compared with specific architecture, for instance, if the convolution matrix has uniform or symmetric values (in this case, similar computations are done several times).

6. Implementation

The architecture design in Handel-C is modeled as an assembly of macro procedures (see Algorithm 1 for the main higher level procedure). Each macro procedure correspond to a functional component of the architectural model and has input and output channels. There are two forms of macro procedures.

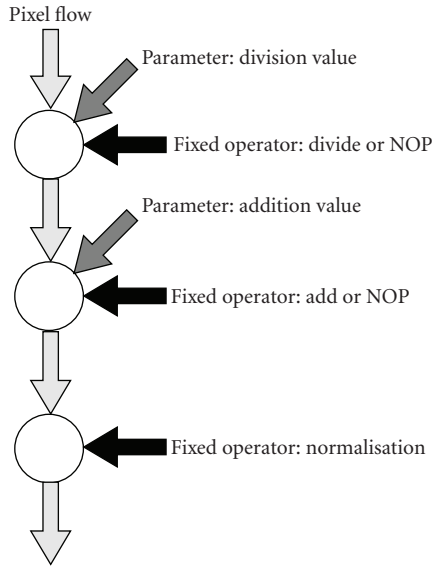


Figure 7: Unary identical postprocessing units.

```
void main (void){

  /*declaring communication channels*/
  chan unsigned 16 pixelCamFlow;
  chan unsigned 8 pixelFlow [4];
  chan unsigned char config [3];

  /*parallel activity*/
  par{
    camera (pixelCamFlow);
    conversion (pixelCamFlow, pixelFlow [0]);

    /* first module */
    module (pixelFlow [0], pixelFlow [1]
                              ,config [0], TRUE);

    /* following modules */
    par(i=1; i < moduleNumber − 1; i++){
      module (pixelFlow [i],pixelFlow[i + 1]
                              ,config [i],FALSE);

    display (pixelFlow[moduleNumber − 1]);
}}
```

Algorithm 1: Main program: all subcomponents (macro procedures) are active in parallel and are connected by synchronous channels.

(i) Terminal macro procedures execute an infinite loop where they read values from input channel, compute a result, and write this result to output channel.

(ii) Intermediate macro procedures interconnect terminal macro procedures to obtain the behavior of a component of the architectural model. The code is mainly made of channel declaration and specification of simultaneous activity of subcomponents.

Algorithm 2 shows a typical code example for a compute node (terminal), illustrated by a node from the reduction

```
macro proc computeUnit
        (functionChan, inChan, outChan){
/*variables*/
signed 13 vIn [2]; /*inputs*/
signed 14 result; /*output*/
unsigned char functionSymbol;

/*receiving the function symbol*/
functionChan ? functionSymbol;

/*initialization of the pipeline*/
/*not shown here (just comments)*/
/*receiving first input value set*/
/*then, in parallel :*/
/*receiving second input value set*/
/*computing first result*/

/*infinite loop*/
While (1){
  par{
  /*doing in parallel in one clock cycle*/

    /*receiving input values*/
    par(i=0; i < 2 i++){
      inChan[i] ? vIn[i];}

    /*computing result*/
    switch(functionSymbol){
     /*ADD*/ case '+' :
       result = vIn[0]+vIn[1];
       break;
     /*MIN*/ case '<' :
       result =
         vIn[0] < vIn[1] ? vIn[0] : vIn[1];
       break;
     /*MAX*/ case '>' :
       result =
         vIn[0] > vIn[1] ? vIn[0]:vIn[1];
       break;
       default : delay; break;}

    /*sending result*/
    outChan ! result;
}}}
```

Algorithm 2: Typical code example for a compute node (terminal macro procedure): in a clock cycle, in parallel, inputs are received, result is computed, and output is sent.

processing unit (initializations and type casting have been removed for comprehension). One can see that, in a infinite loop, three instructions are executed simultaneously during each loop cycle. While first instruction is receiving the pixels of generation $i + 1$, second instruction is computing the result for generation i, and the third instruction is sending the result of generation $i - 1$. The three instructions take one clock cycle to be achieved. Simultaneous reading and writing are not a problem in this case, as all value registers are updated at the same clock tick.

Algorithm 3 shows the code for the reduction computing unit, instantiating and interconnecting compute nodes and nop nodes. It should be noticed that we need different

```
macro proc reductionUnit
            (funcChan, inChan, outChan){
  /*declaring communication channels*/
  chan signed 13 c5,c6,c7;
  chan signed 14 c2[4];
  chan signed 15 c3[2];
  chan signed 16 c4;

  /*parallel activity of compute nodes*/
  par{
    par(j = 0; j < 4; j++){
      computeUnit1(funcChan,
        inChan[2 * j], inChan[2 * j + 1], c2[j]);}
    nopNode (inChan[8], c5);

    par(k = 0; k < 2; k++){
      computeUnit2 (funcChan,
                    c2[2 * k], c2[2 * k + 1], c3[k]);}
    nopNode (c5, c6);

    computeUnit3(funcChan,c3[0],c3[1],c4);
    nopNode (c6, c7);

    computeUnit4(funcChan,c4,c7,outChan);
}}
```

Algorithm 3: Code example for the reduction computing unit (intermediate macro procedure).

```
macro proc movingWindowUnit
            (pixelInChan, pixelsOutChan){

unsigned 8 pixels[2 * Width + 3], pixelIn;

while(1)
  par{/*in parallel*/

    /*receiving a pixel*/
    pixelInChan ? pixelIn;

    /*storing the last received pixel*/
    pixels[0] = pixelIn;

    /*shifting the memory values*/
    par(i = 1; i < 2 * Width + 3; i++){
      pixels[i] = pixels[i - 1]; }

    /*sending the neighborhood*/
    par(j = 0; j < 3 ; j++){
      par{
        pixelsOutChan[j] ! pixels[2 * Width + 2 - j];
        pixelsOutChan[3 + j] ! pixels[Width + 2 - j];
        pixelsOutChan[6 + j] ! pixels[2 - j];}}

}}
```

Algorithm 4: Code example for the moving window unit (simplified version without initialization and border management).

compute nodes corresponding to the different input and output types, even if the behavior is similar.

Algorithm 4 shows the code for the moving window unit. The pixel array is implemented with LUT configured as shift-registers, which limits the number of register and prevents the need of instantiating block ram for it.

As far as pedagogical issues are concerned, the project has also developed student's consciousness about FPGA application development regarding "compile times" (i.e., compile, synthesis, map, place, and route times). It is a fact that, in their curriculum, students have rarely seen "compile times" that exceed dozens of a second. When facing, for the largest circuits, "compile times" that exceed one hour, there is a great disappointment. This is one bad aspect of using a "high-level language" description to program an FPGA: it is easier for CS students, but if students just have in mind the program and the result without having in mind the complexity of the chain tools, comparison with software programming is quite in disfavor.

Practically this fact has the advantages to make students aware of an efficient use of the time spending on development: you cannot manage to efficiently spend your time if you wait for compilation finishing. Solutions include validating the component on a restricted set that is known to be scalable without loss of functionality (e.g., reducing image width) or working on and validating several components independently.

Concerning technical issues, another problem that students faced was managing the data types and the type casting. Even if Handel-C permits some automated type inference in particular cases, we have chosen to define all types explicitly. In fact, managing types in the reduction processing unit induces irregularity since types are different in the various compute nodes; so each node level must be described explicitly. This highlights for the students the importance of being able to precisely know the input minimal type, minimal value interval, or minimal number of value in order to improve the design performances.

Simulation, which is an important aspect of project development process, is not addressed in this course. Using a low-level behavioral simulator, like ModelSim, is out of the scope of the course: the main reason is that it would be a hard task for CS students to make the link between their high-level Handel-C program and the simulated signals as net names are generated by Handel-C compiler. Another solution would be to use the high-level Handel-C simulator of DK design suite, which permits simulation within a graphical environment (input and output images) based on the cycle accurate simulation of integer variable values. However such a software simulation is quite time-consuming and requires additional training for students. Consequently the development process for this project avoids simulation: the implementations are directly tested on the FPGA. The application context (image processing with real-time screen display) allows visual detection of most of errors or problems (e.g., black screen or pixel shift).

One interesting aspect of the implementation is the immediate visual impression about the performances of the system, as a low frame refresh rate implies a too long time for a system cycle. This aspect motivates the students in increasing the parallelism level of their design, in order to obtain a visually acceptable frame rate.

Table 1: Comparing resource use and critical path delay for various multimodule implementations.

Number of modules	LUTs	Flip Flops	Critical path delay
1	3595	2024	31.18 ns
2	7220	3749	45.57 ns
3	10717	5584	55.22 ns
4	14709	7500	49.09 ns
5	18750	9357	49.49 ns

Table 2: Comparing resource use and critical path delay for reconfigurable module and specialized module.

Design	LUTs	Flip Flops	Critical path delay
Configurable module	3595	2024	31.18 ns
Mean filter module	778	553	21.70 ns
Low pass filter module	608	520	19.90 ns
High pass filter module	612	524	20.84 ns

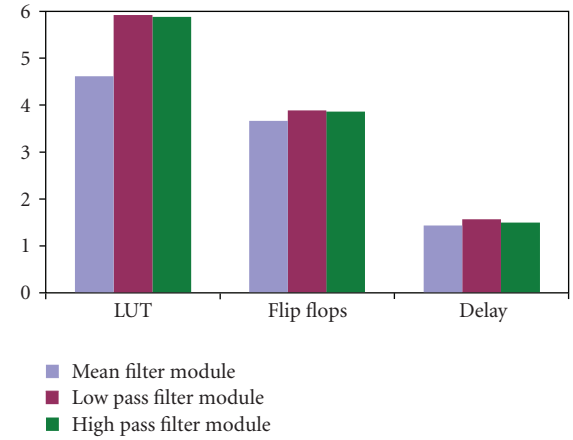


Figure 8: Size multiplication factor of a reconfigurable module compared to a specialized module for three applications.

7. Results

Students have implemented multimodule design for an image of 240 × 256 pixels. Initial pixel values are coded as an 8-bit integer (gray levels). The maximal width for the data path is of 18 bits (signed). Operands for multiplication, division, and last addition have, respectively, 5-, 6- (signed), and 8-bit width.

Table 1 shows the resource use (in LUT and flip flops) and critical path delay for a growing number of modules (maximal reachable number was 5), obtained after synthesis with DK4, map, place, and route with default parameters of ISE 8.2. Presented area is just for the composite of modules: the whole system needs approximately 1450 LUTs and 1150 flip flop more. One should notice that arithmetic units of the spartan 3L were also used (9 Mult 18 × 18 units per module). Results show that implementing several modules increases the critical path delay but it stays low enough to keep a satisfying real time computing and display.

A coarse estimation of the reconfigurability costs has been done by implementing modules with fixed behavior: global organization is the same but configuration controller is removed, and compute node is limited to one fixed function. Table 2 compares the area and delay of reconfigurable module with specialized module for mean filter, low pass filter, and high pass filter. Figure 8 presents the size multiplication factor for the different applications when using a reconfigurable module. Results show a division by 5 of the needed resources for the specialized modules. Obviously reconfigurability is costly in this case; however this should be related to the conceptual gain for user.

8. Object Modeling and Automated Generation

Even if the specification of the architecture is simplified by the language (Handel-C), the communicating process model (CSP), and the restrictive pattern for taking advantage of data flow parallelism, experience has shown that resulting programs still needed various repetitive and unproductive tasks, mainly managing type of variables, type casting, and channel connection. Those aspects prevent a program to be simply adapted to a different operational context, assuming a same application context. Solving these problems require having a generic model instance that is equipped with automated code generation methods, similar to an algorithm description in terms of hardware skeletons [19].

This is the last step of the project: to show concretely what are the gains of abstracting functionalities in terms of productivity. When the project is about to be finished, the principles of an object modeling with automated code generation are presented to students. If more time was allowed to the course, it would have been important to let them develop also this part, but this work has only been proposed as an optional project included in the curriculum.

We do not intend to describe here the exhaustive advantages of abstract modeling of an architecture class, but for this particular case we can list the main ones.

(i) Automated variable type determination and casting: from the initial type of pixel and the operator information, all types in the module are automatically processed.

(ii) Simple assembly and modifications of units and module connections: the logical links between units are specified and communicating channel are automatically allocated. For instance, *nop* nodes are automatically included in a reduction processing unit depending on the total number of inputs and number of inputs of basic nodes.

(iii) Architectural prospection: one can easily modify the functionalities and the granularity of a compute unit, generate the corresponding program, and evaluate its performances to choose the best solution, for instance, merging several successive nodes to a single node with a sequence of functions for saving cost of communication, without loss of performance if the time spent in the sequence is less than the time of the longest function in the architecture. An optimal

combination of pixel buffers and compute units can be determined depending on available resources and targeted computation time [20].

(iv) Code productivity and correctness: compute units (unary, binary, ...) share common structures which are clearly specified in a superclass. Generated code is factorized for all instances of this class.

(v) Agility: functionalities can be easily extended or restricted to focus on a specific application model.

For our problem, the object modeling of the architecture component is made up of the following classes:

(i) computing function, defined by function symbol, behavioral code (compound of inputs, parameters and operators), output type (if fixed), or dynamic type (e.g., for an addition increasing of the output width by one related to the maximal input or parameter widths);

(ii) computing node, defined by input array (pixels and parameters) and function array;

(iii) moving window node, defined by the size of the square neighborhood window and the position of the center pixel in this window, the image size, and the type of input pixel;

(iv) tree computation node defined by the basic computation node to use, and the number and the type of inputs; it defines the reduction processing unit;

(v) computation module: composite of nodes that uses instance of the previous classes to define an architecture similar to the one of the project;

(vi) classes for manipulating values, distinguishing data, parameters, and values that are just read (for a test) or used in a computation.

9. Conclusion and Perspectives

In this paper we have presented a problem-based pedagogical approach for teaching reconfigurable computing to unspecialized CS students in terms of reconfigurable architecture design, programming and technical skills, in a relatively limited time.

It is difficult to precisely evaluate the impact of such an approach on the students. The questionnaires completed by the students at the end of the project help to have various indicators. For instance, in a class of 22 students, 81% of them were more motivated by the course with this approach, 86% of them evaluated their contribution to the group to be significant, 90% of them thought that problem-based learning eased their knowledge acquisition, and 95 % of them wished to have more problem-based courses in their curriculum. Obviously CS students would not be able to design such a system without the gradual steps of the problem-based approach.

Concerning reconfigurable computing fundamentals, some aspects are missing and would merit integration. For instance, data parallelism could be tackled by dividing the image into strips computed simultaneously by parallel modules, with synchronization for border management. On another project, this data parallelism has been illustrated by using concurrent modules operating on same data (e.g., horizontal and vertical edge detection). Moreover we should go further into optimizing functions: as we assumed that a function is computed in one clock cycle, the clock frequency is limited, even if an automated retiming is performed by low-level synthesis tools. Explicit pipelining of functions (e.g., division) would increase clock frequency and data rate. If we go further into this pipelining, another interesting aspect is the tradeoff between flow parallelism and cost of channel communication; that is to say to determine what is the minimal granularity for a function that guarantees that it is not more costly to include it in a communicating process than to sequentially aggregate it with other functions in a coarser process.

We have seen in Section 7 that the end of the project opens up about high-level modeling of architecture and associated tools for optimization, automated typing, and code generation, which is close to research problems. Integrating the project into a more global environment like Madeo [21] would allow to use its automated typing inference by value enumerating, for instance, to optimize the operators in case of a restricted set of available values for parameters, which is the case for most of image processing. Other generalization of the problem would be to define it as a "configurable" CDFG [22] (Control/Data Flow Graph) to take advantage of targeting several systems more efficiently than a specific code generation.

However, including it into a course would require research tools to be accessible to students, which is a great challenge: reconfigurable computing is a fast evolving discipline; but we believe that this access to current research is a key issue to offer companies well-educated, innovation aware, and highly concerned engineers.

Acknowledgment

The author thanks Loïc Lagadec for his fruitful comments.

References

[1] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[2] C. Dezan, L. Lagadec, and B. Pottier, "Object oriented approach for modeling digital circuits," in *Proceedings of the IEEE International Conference on Microelectronic Systems Education (MSE '99)*, IEEE Computer Society, July 1999.

[3] C. Bobda, "Building up a course in reconfigurable computing," in *Proceedings of the IEEE International Conference on Microelectronic Systems Education (MSE '05)*, pp. 7–8, 2005.

[4] E. P. Ferlin and V. Pilla Jr., "The learning of reconfigurable computing in the computer engineering program," in *Proceedings 36th ASEE/IEEE Frontiers in Education Conference (FIE '06)*, San Diego, Calif, USA, October 2006.

[5] E. Fabiani, C. Gouyen, and B. Pottier, "Intermediate level components for reconfigurable platforms," in *Proceedings of the International Workshops on Computer Systems: Architectures,*

Modeling, and Simulation (SAMOS '04), vol. 3133, pp. 59–68, Springer, Samos, Greece, July 2004.

[6] A. Kolmos, "Problem-based and project-based learning," in *University Science and Mathematics Education in Transition*, pp. 261–280, 2009.

[7] A. Stojcevski and D. Fitrio, "Project based learning curriculum in microelectronics Engineering," in *Proceedings of the 4th IEEE International Conference on Parallel and Distributed Systems (ICPADS '08)*, pp. 773–777, IEEE Computer Society, Australia, December 2008.

[8] Celoxica, "Handel-C Language Reference Manual," 2005.

[9] P. H. Welch and D. C. Wood, "The Kent Retargetable occam Compiler," in *Proceedings of the 19th World Occam and Transputer User Group Technical Meeting on Parallel Processing Developments (WoTUG '96)*, B. O'Neill, Ed., pp. 143–166, March 1996.

[10] Celoxica, "Platform Developer Kit: RC10 Manual".

[11] R. B. Porter, "Image processing," in *Reconfigurable Computing*, Springer, New York, NY, USA, 2005.

[12] P. Guermeur, P. Dokladal, E. Dokladalova, and A. Manzanera, "FPGA lab sessions in a general purpose image processing course," in *Proceedings of the 2nd International Workshop on Reconfigurable Computing Education*, May 2007.

[13] Celoxica, "DK Design Suite user guide," 2005.

[14] Xilinx, "ISE 8 Software Manuals," 2006.

[15] V. Muthukumar and D. V. Rao, "Image processing algorithms on reconfigurable architecture using HandelC," *Journal of Engineering and Applied Science*, vol. 1, no. 2, pp. 103–111, 2006.

[16] D. V. Rao, S. Patil, N. A. Babu, and V. Muthukumar, "Implementation and evaluation of image processing algorithms on reconfigurable architecture using C-based hardware descriptive languages," *International Journal of Theoretical and Applied Computer Sciences*, vol. 1, no. 1, pp. 9–34, 2006.

[17] C. Torres-Huitzil and M. Arias-Estrada, "FPGA-based configurable systolic architecture for window-based image processing," *EURASIP Journal on Applied Signal Processing*, vol. 2005, no. 7, pp. 1024–1034, 2005.

[18] D. Etiemble and L. Lacassagne, "Introducing image processing and SIMD computations with FPGA soft-cores and customized instructions," in *Proceedings of the International Workshop on Reconfigurable Computing Education*, 2006.

[19] K. Benkrid, D. Crookes, and A. Benkrid, "Towards a general framework for FPGA based image processing using hardware skeletons," *Parallel Computing*, vol. 28, no. 7-8, pp. 1141–1154, 2002.

[20] X. Liang and J. S.-N. Jean, "Mapping of generalized template matching onto reconfigurable computers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 3, pp. 485–498, 2003.

[21] L. Lagadec, B. Pottier, and O. Villellas, "A LUT based high level synthesis framework for reconfigurable architectures," in *Domain-Specific Processors: Systems, Architectures, Modeling and Simulations*, pp. 19–39, Marcel Dekker, 2003.

[22] M. Rashid, T. Goubier, and B. Pottier, "A high level generic application analysis methodology for early design space exploration," in *Proceedings of the Workshop on Design and Architectures for Signal and Image Processing (DASIP '07)*, November 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 908740, 13 pages
doi:10.1155/2009/908740

Research Article

A Reconfigurable and Biologically Inspired Paradigm for Computation Using Network-On-Chip and Spiking Neural Networks

Jim Harkin,[1] Fearghal Morgan,[2] Liam McDaid,[1] Steve Hall,[3] Brian McGinley,[2] and Seamus Cawley[2]

[1] *School of Computing and Intelligent Systems, University of Ulster, Derry BT48 7JL, Northern Ireland*
[2] *Bio-Inspired Electronics & Reconfigurable Computing Group, NUI Galway, Galway, Ireland*
[3] *Department of Electrical Engineering & Electronics, University of Liverpool, Liverpool L69 3GJ, UK*

Correspondence should be addressed to Jim Harkin, jg.harkin@ulster.ac.uk

Received 1 December 2008; Accepted 13 April 2009

Recommended by Michael Huebner

FPGA devices have emerged as a popular platform for the rapid prototyping of biological Spiking Neural Networks (SNNs) applications, offering the key requirement of reconfigurability. However, FPGAs do not efficiently realise the biologically plausible neuron and synaptic models of SNNs, and current FPGA routing structures cannot accommodate the high levels of interneuron connectivity inherent in complex SNNs. This paper highlights and discusses the current challenges of implementing scalable SNNs on reconfigurable FPGAs. The paper proposes a novel field programmable neural network architecture (EMBRACE), incorporating low-power analogue spiking neurons, interconnected using a Network-on-Chip architecture. Results on the evaluation of the EMBRACE architecture using the XOR benchmark problem are presented, and the performance of the architecture is discussed. The paper also discusses the adaptability of the EMBRACE architecture in supporting fault tolerant computing.

1. Introduction

Biological research has accumulated an enormous amount of detailed knowledge about the structure and function of the brain. The basic processing units in the human brain are neurons that are interconnected in a complex pattern [1]. The current understanding of real biological neurons is that they communicate through pulses and use the timing of the pulses to transmit information and perform computations. Spiking Neural Networks (SNNs), which interact using pulses or spikes, emulate more closely real biological neurons of the brain and have therefore the potential to be computationally more powerful than traditional artificial neural network models [2]. Fault tolerant computing can exploit the brain's behaviour in the repair and restoring of functionality. This is a feature of particular interest to SoCs designers where reliability is becoming difficult to guarantee postdeployment with scaling device geometries, and also in the deployment of computing systems in harsh environments.

Inspired by biology, researchers aim to implement reconfigurable and highly interconnected arrays of Neural Network (NN) elements in hardware to produce robust and powerful signal processing units. However, the standard topologies employed to model biological SNNs are proving difficult to emulate and accelerate in hardware, even for moderately complex networks.

Current software environments available for simulating Spiking Neural Networks (SNNs) provide biophysically realistic models, albeit with large simulation times [3]. Moreover, software simulations of SNN topologies and connection strategies face the problem of scalability in that biological systems are inherently parallel in their architecture whereas commercial PCs are based on sequential processing

architectures [4] making it difficult to assess the efficiency of these models to solve complex problems.

The ability to reconfigure FPGA logic blocks and interconnect has attracted researchers to explore the mapping of SNNs to FPGAs [5–9]. Efficient, low-area/power implementations of synaptic junctions and neuron interconnect are key to scalable SNN hardware implementations. Existing FPGAs limit the synaptic density achievable as they map biological synaptic computations onto arrays of digital logic blocks, which are not optimised in area or power consumption for scalability [3]. Additionally, current FPGA routing structures cannot accommodate the high levels of neuron interconnectivity inherent in complex SNNs [3]. The Manhattan-style mesh routing schemes of FPGAs typically exhibit switching requirements which grow nonlinearly with the mesh sizes [10]. A similar interconnect problem exists in System-on-Chip (SoC) design where interconnect scalability and high degrees of connectivity are paramount [11]. The use of networking concepts has been investigated to address the interconnectivity problem using Network-on-Chip (NoC) approaches which time-multiplex communication channels [12]. The NoC approach employs concepts from traditional computer networking to realise a similar communicating hardware structure. The key benefit provided by NoCs is scalable connectivity; higher levels of connectivity can be provided without incurring a large interconnect-to-device area ratio [11].

This paper presents the EMulating Biologically-inspiRed ArChitectures in hardwarE (EMBRACE) hardware platform for the realisation of SNNs. EMBRACE uses an NoC-based neural tile architecture and programmable neuron cell which address the interconnect and biocomputational resources challenges. The paper illustrates how the EMBRACE architecture supports the routing, biological computation, and configuration of SNN topologies on hardware to offer scalable SNNs with a synaptic density significantly in excess of what is currently achievable in hardware [7–9, 13]. In addition, the paper discusses the potential opportunities EMBRACE offers in providing a new hardware information processing paradigm which has the inherent ability to accommodate faults via its neural-based structures.

Section 2 of the paper provides a review of related work and identifies the challenges of scalable SNN hardware implementations. Section 3 describes the proposed EMBRACE architecture, and Section 4 discusses the neural tile. Section 5 presents results on the evaluation of the architecture using the XOR benchmark problem and on neural tile implementations from the perspective of performance-architectural optimisations. Section 6 provides a discussion on the fault tolerance opportunities of the new paradigm, and Section 7 provides a summary and outline of future work.

2. Background

SNNs differ from conventional artificial NN models because information is transmitted by the means of pulses or spikes. Brain-inspired paradigms such as SNNs offer the potential of an elegant, low-powered, and robust method of performing computing. In particular, SNNs offer the potential to emulate the ability of the brain to tolerate faults and repair itself. The human brain continually replaces neurons by reconnecting to newly generated neighbouring neurons via synaptic junctions. This adaptability allows the brain to overcome faults, so providing the inspiration for establishing robust fault tolerant computers. Other similar approaches are currently under investigation for fault tolerant computing paradigms based on biological organisms [14].

When implemented in hardware, SNNs can take full advantage of their inherent parallelism and offer the potential to meet the demands of real-time fault tolerant applications. However, the first step towards SNN-based fault tolerant systems is the creation of a hardware platform which can support the levels of parallelism and adaptability required.

FPGAs have been widely recognised as a platform which can provide the adaptable requirements of NNs [8] and so enable rapid acceleration of NNs and hardware-in-loop training. The popularity of FPGAs has been fuelled by the reconfigurability offered by such devices and the introduction of platform FPGAs with increased logic and embedded processors [15].

2.1. Biological Spiking Neurons. FPGA implementations of SNNs typically aim to accelerate and prototype biologically inspired computations [5, 7, 8, 13, 16]. However, FPGAs are not appropriately suited for efficient SNN implementation as they attempt to map biological neuron and synaptic computations onto general arrays of configurable digital logic blocks which are not optimised in area and power for dense network realisations [8]. Existing FPGAs can only provide limited synapse density. More importantly, they underutilise silicon area as larger numbers of general programmable units are required to achieve specific, optimal implementations. Providing configurable, dedicated computational SNN blocks within a custom FPGA-type structure will offer a more suitable reconfigurable platform for SNN accelerated prototyping, with optimised utilisation of hardware area and power. Also, issues associated with training must be addressed whereby the programmability and efficient storage of synaptic weights are accommodated.

2.2. FPGAs and Neuron Interconnectivity. SNNs have a significant level of connectivity between neurons and, increasing SNN neuron density results in a nonlinear interconnect growth. For example, a 2-layered feed forward fully interconnected network with m neurons per layer exhibits an interconnect density of m^2, which rapidly increases as the number of neurons per layer increases. Neuron interconnection in current FPGAs is typically achieved using Manhattan style layouts with diagonal [17], segmented, or hierarchical 2-dimensional routing structures [10]. Scaling is one of the key issues associated with all complex electronic systems because interconnect consumes large areas of chip real-estate and subsequently causes longer critical path delays. For example, the switching requirements of Manhattan-style

routing schemes typically grow nonlinearly with the number of logic units on the device [10]. This FPGA interconnect challenge is a significant limiting factor in the suitability of FPGAs for SNN implementation.

Researchers have investigated several routing optimisations and topologies in their attempts to improve the FPGA routing latency and performance. Rose et al. [17] investigated architectural-level techniques with the introduction of bus-based routing between clusters of multibit logic blocks to reduce routing interconnect density. Increased semiconductor multilevel metalisation has increased the number of metal layers that can be realised and has provided new opportunities for scaling FPGA routing using the Mesh-of-Trees (MoTs) topology [10]. More recently, nanotechnology has provided nanowire-based routing topologies for FPGAs [18].

2.3. Network-on-Chip (NoC). SoC design [11] has seen a considerable interconnect challenge with the introduction of multiprocessors. SoCs commonly use Network-on-Chip (NoC) schemes [19], with varied NoC topologies [20], router architectures, and the provision of low-power and high-Quality-of-Service (QoS) designs. Router architectures include asynchronous [19], circuit-based, packet, and wormhole switching [14, 21]. The NoC approach uses computer networking concepts to achieve a similar networking structure on hardware. The key benefit from using the NoC is scalable connectivity; higher levels of connectivity can be achieved without incurring a large ratio of interconnect-to-device area. Researchers have demonstrated performance benefits from incorporating NoC structures in FPGA application designs [14, 22–24]. The success of NoCs has seen the release of commercially available technologies from Silistix [25] and Arteris [26].

The SNN interconnect problem is similar to that of SoCs; SNNs have large numbers of neurons (typically in excess 1000 for complex applications), which exhibit high inter-neuron connectivity requirements.

Current attempts to realise SNNs using multiprocessors in hardware have included limited numbers of neurons due to connectivity issues [3, 6, 8, 9, 27–29]. An FPGA-based multiprocessor SNN [6], incorporating reduced interneuron connectivity, has been demonstrated. However, this implementation strategy does not scale efficiently since large FPGA SNN networks are inherently limited by the interconnect requirements between large numbers of processors [10]. Similarly, multiprocessor approaches to support the accelerated simulation of SNNs are also under investigation [27, 28] and use network-type communication structures. However, in general these approaches do not accommodate the dedicated computational requirements of the biological neurons and the temporal dynamics of SNN. Reported implementations provide suboptimal strategies which attempt to "force-fit" SNNs into current multiprocessor architectures which typically target regular data-path computational applications.

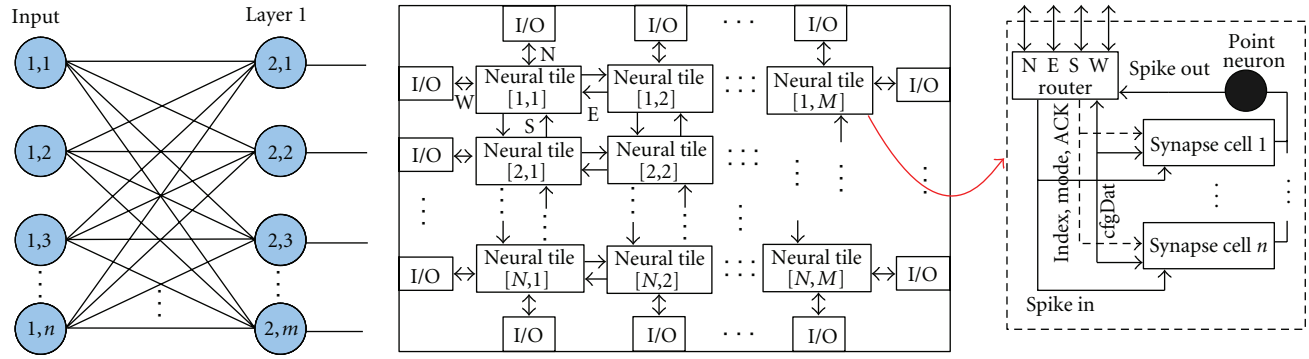
Several full-custom neuromorphic architecture devices have been proposed [13, 30, 31] which aim to address the inefficiencies of FPGAs by including optimised synaptic cells and Address Event Representation (AER) routing [32]. However, these architectures cannot scale due to limited connectivity provided by AER between neurons. More importantly, their "fixed" full-custom nature has a significant impact on the level of reconfigurability and prohibits the computational benefits afforded by programmable SNN interconnect.

2.4. Current Emulation Challenges. A major challenge is the development of bioinspired platforms which can support scalable low-power realisations and reprogrammable interconnect. In particular, the problem of SNN interneuron connectivity is the dominant obstacle that prohibits the implementation of biological scale NNs. The rapid increase in the ratio of fixed connections to the number of neurons self-limits the network size [1]. If scalable, reconfigurable networks are to be realised on SNN, domain-specific programmable devices, the interconnect issue must be addressed. This sets the primary focus of the paper. To address the large scale implementation issues of programmability, synapse weight storage, power consumption, and inter-neuron connectivity, a new device architecture tailored to the requirements of SNNs is required.

This paper proposes EMBRACE, a custom field programmable neural network architecture (EMBRACE) which merges the programmability features of FPGAs and the scalable interconnectivity of NoCs with low-area/power spiking neuron cells that have an associated training capability. The EMBRACE architecture supports the programmability of SNN topologies on hardware, providing an architecture which will enable the accelerated prototyping and hardware-in-loop training of SNNs. Earlier investigations by the authors in using NoC router strategies to implement large scale artificial neural networks [33] have demonstrated benefits in network scalability. In addition, the authors have developed an initial custom low-area/power programmable synapse cell with characteristics similar to real biological synapses [34, 35]. The following sections introduce the proposed EMBRACE architecture.

3. Embrace Architecture

The EMBRACE architecture is illustrated in Figure 1(b) as a 2-dimensional array of interconnected neural tiles surrounded by I/O blocks. The authors recognise that approaches to implementing the brain, which is a 3D structure, are currently limited to 2D because current fabrication techniques are not mature enough to reliably support large scale 3D SNN architectures. Therefore, the neural tiles are connected in North, East, South, and West directions forming a nearest neighbour connect scheme. Each neural tile can be programmed to realise neuron-level functions which collectively implement an SNN. An SNN is realised on the EMBRACE architecture by programming the tile functionality and connectivity. Consider the interconnectivity requirements of a feed-forward (FF), 2-layer $n \times m$ SNN network with each neuron in layer 1 connected to m neurons



(a) 2-layer SNN network (b) EMBRACE: Architecture (c) EEMBRACE: Neural Tile

FIGURE 1: Architecture overview.

in layer 2; see Figure 1(a). When a neuron in layer 1 fires (spikes), its pulse signal is propagated to the target neurons in layer 2 via dedicated individual lines. Using an NoC strategy, the same pattern of connectivity between layers is achieved through time-multiplexing of the communication channels. This significantly reduces interconnect density as the proposed architecture of Figure 1(b) includes a reduced number of fixed, regular-layout communication lines, and a network of NoC routers.

EMBRACE runtime and configuration data is propagated from source to destination via time-multiplexing using the NoC routing lines. For example, during runtime, spike events occurring in layer 1 are forwarded to the associated synapses of layer 2 via several router transmission steps. The proposed NoC strategy uses individual routers to group n synapses and the associated neuron, using a novel structure referred to as a *neural tile*, illustrated in Figure 1(c); the neural tile is viewed as a macro-block of EMBRACE. The novelty of the tile resides in the merging of analogue synapse/neuron circuitry with NoC digital interconnect to provide a scalable and reconfigurable neural building block. Other approaches [28, 29] do utilise NoC routers but with software models of synapse and neurons running on microprocessors. Figures 2 and 3 illustrate how the $n \times m$ network can be realised using the NoC strategy, whereby m neural tiles are required, with each one corresponding to one of the m postsynaptic neurons of layer 2; a feed-forward (FF) network with 10^3 neurons per layer would require 10^3 neural tiles each containing 10^3 synapses. The m neural tiles are arranged in a 2D array structure, as shown in Figure 3, with i number of rows and j columns, for example, $i = 200$ and $j = 5$ for $m = 10^3$. Note that Figure 2 highlights which synapses are connected to neuron (2, 1), and Figure 3 illustrates how the synapse and neuron functionality are mapped to tile number 1; for example, each synapse maps to one of the n synapse cells in the tile. The mapping process is repeated for neurons (2, 2) through to (2, m), where the unique n synapses for each neuron are allocated to tiles 2 through to m, respectively.

3.1. Distribution of Computational Resources. The aim of the NoC strategy is to use the router of each tile to communicate spike events to its group of n synapses. This enables a reduced

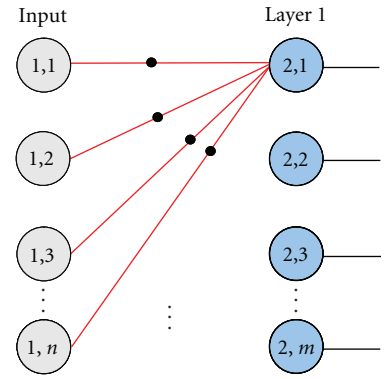


FIGURE 2: n x m network showing only the synapse inputs to neuron (2,1). Each of the m neurons in layer 1 has n synapse connections.

number of connections; for example, an SNN interconnect density of 10^6 ($n \times m$) can be implemented using 4×10^3 ($4 \times m$ connections). The 4 term stems from the N, E, S and W router connections. The n individual synapses in a neural tile are referred to as *synapse cells* and are combined with the point neuron to form the neuron cell. The synapse cell is analogue in nature and captures the pertinent biological features of real synapses [34]. The output responses from each synapse cell are connected together to produce the desired biological response from the point neuron.

Due to the time multiplexing manner of the interconnections between neurons, the data transfer time (spike-interval) between neurons is increased; however this is not a significant degradation to the speed-performance of EMBRACE as the biological speed (spike-interval) of the human brain is typically in the order of 10 milliseconds [2, 4].

3.2. Spike Event Communication. The inputs and outputs of the synapse cells are controlled via the NoC router. The events of a spike train are received as data packets from neighbouring routers where each spike-event packet includes a source address (indicating the SNN neuron/tile in which the event originated) and destination address (of neural tile and synapse). Figure 4 illustrates the packet format where the

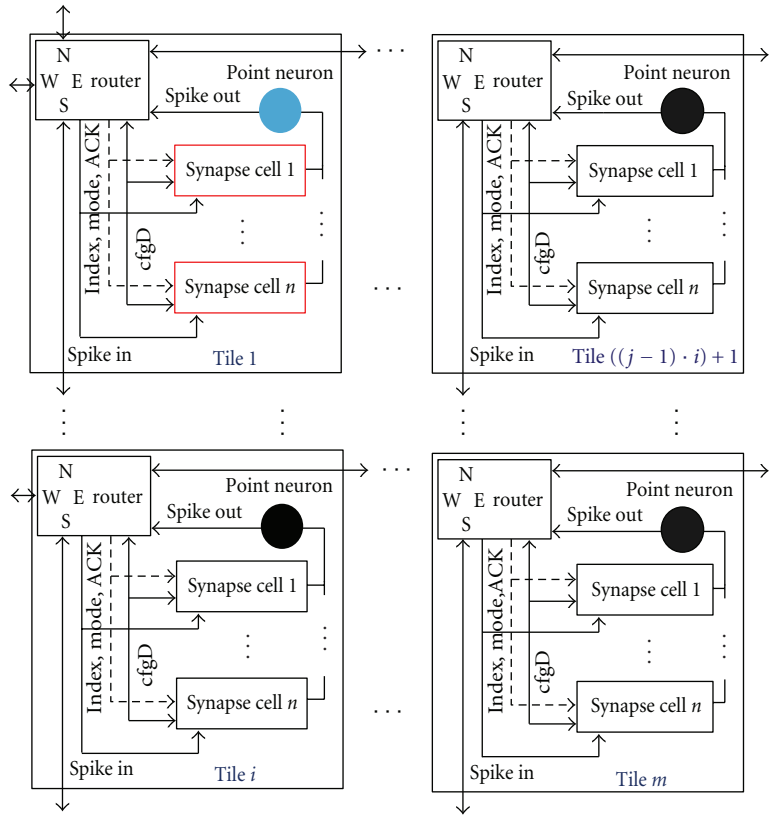


Figure 3: Mapping SNNs to EMBRACE: realisation of the neuron (2,1) and its n synapse connections to tile 1.

header contains information on the type of data contained in the payload (i.e., a packet can be contain runtime, configuration, or error data). The maximum payload size per packet is 22-bits where in runtime mode it contains the addresses of the source and target neuron. Each packet is transmitted as two 12-bit flits in runtime mode; in configuration mode the number of flits can verify as discussed later in Section 4.1. Tile routers send an output packet each time a spike-event occurs within the tile. Exploiting the relatively lowfrequency of biological spike trains (~Hz) [2], EMBRACE can use a regular and scalable NoC structure. The time-multiplexing of spike data along router paths between SNN layers enables large parallel networks and high levels of routability, without the need for an overwhelmingly large number of connections; see Figure 1(a). The proposed method of connecting neural tiles enables various SNN topologies to be realised for example, multilayered feed-forward and recurrent networks can be implemented.

4. Reconfigurable Neural Tile

The EMBRACE neural tile is illustrated in Figure 5 and highlights the connections between the NoC router, synapse cells, and point neuron. The packet-switched router implements 12-bit communication paths with buffer support where a round-robin scheduling policy is used for arbitration at the buffer input/outputs. The NoC router uses an XY routing scheme where all flits are transmitted in the same path direction (wormhole operation).

The intra-tile communication buses include the following signals: *Spike I/P* and *O/P*, *Mode*, *ACK*, *Config Data* and *Indexing*.

(i) *Spike I/P* initiates a spike on an individual synapse cell.

(ii) *Spike O/P* receives spike events from the neuron.

(iii) *Mode* specifies the tile operation, namely, runtime or configuration programming

(iv) *Indexing* bus is used to address individual synapse cells (via address decoders) for receiving spike events or configuration data.

(v) *ACK* acknowledges the correct synapse addressing.

(vi) The *Config Data* bus is used to transmit configuration data to the cells. Cells configure the connections to the q global voltage lines, V, which are common to tiles and run throughout the device.

Each neural tile has a unique address, and the synaptic connectivity is also specified using an Address Table (AT) within each router. The AT is programmed during EMBRACE configuration to specify the desired connectivity between tiles and enables spike events to be routed. A spike event is detected at the *Spike O/P*, and the AT identifies which target neural tiles and synapses must receive event notification.

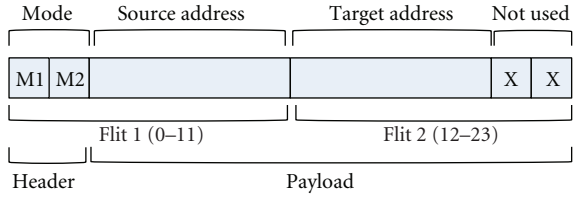


Figure 4: General Packet Format.

Table 1: Packet header definitions.

Header Information	M1, M2
Runtime mode	00
Configuration mode (Start)	01
N/A	10
Configuration mode (End)	11

4.1. Reconfigurable and Runtime Operations. The neural tiles operate in one of two modes: *runtime* or *configuration*. This is defined by the information in the packet header (see Table 1) where "00" and "01" define the payload to be either runtime or configuration, respectively. In runtime mode, EMBRACE routes spike events (data packets) and computes the programmed SNN functionality. In configuration or programming mode, EMBRACE is configured to realise particular synapse models and the desired SNN topology. Configuration data is also delivered to the EMBRACE device in the form of data packets where each packet is addressed to a particular neural tile and contains information on the configuration of the router's AT, the selection of cell synapse weights via the programmable voltage lines, V_q, and other neural tile parameters. However, in this mode, a packet typically contains more than two flits as the payload exceeds the 22-bit limit shown in Figure 4. The additional flits are required as several synapses are often reconfigured within a single tile. In this case, *Configuration (Start)* and *(End)* information is specified in the packet, header as shown in Table 1. For example, when a router detects "01" in the header of a packet it is alerted that the following packet payload contains configuration data and to expect more flits. When it detects the value "11" in the header, it recognises that the transmission of configuration data is complete for the tile. A wormhole routing scheme is used to forward all flits in the payload to the target tile.

This strategy fully exploits the flexibility of the NoC structure by additionally using it to select and distribute configuration data to EMBRACE tiles. It is envisaged that this type of configuration mechanism would also be able to partially support the repair implementation of a tile neural structure in the event of faults.

4.2. Synapse Weight Storage. Each synapse is analogue in nature and models key pertinent biological features of real synapses [34], such as long and short term plasticity. Spike stimuli are received by synapse inputs. Synapse outputs within a neural tile are combined as inputs to the point neuron of the tile. In keeping with biological plausibility, synapse weight updates for long term plasticity should be governed by a Hebbian-based rule [35], and the authors envisage an off-line training procedure that uses this rule.

Table 2: Example of synapse weight configurations.

S6	S5	S4	S3	S2	S1	Synapse cell weight
1	1	1	1	1	1	5.04 V
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	0	0	0	0	0	2.56
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	0	0	0	0	1	0.08 V
0	0	0	0	0	0	0 V

While floating gate transistors with submicron feature sizes would yield compact analogue weight storage, there are significant issues to be resolved with this technology if it is to be a viable analogue memory storage capability for large scale neural networks: dynamic range, sensitivity, and stability [36]. In view of this we propose the novel weight distribution and storage architecture, shown in Fig.6, where for each synapse cell, any number or combination of p identical (smaller) charge-transfer synapses (CTSs) can be hardwired to programmable weight voltage levels during configuration. V_1 to V_q rails provide a range of supply weights (voltages), selected using digitally controlled analogue switches (S_1 to S_p). Figure 6 illustrates how the current outputs of the CTSs are summed when weight voltages V_1 and V_q are applied to synapse 1 and p. Table 2 illustrates an example synapse cell weight selection where $q = p = 6$, $V_1 = 0.08$, $V_2 = 0.16$, $V_3 = 0.32$, $V_4 = 0.64$, $V_5 = 1.28$, and $V_6 = 2.56$ volts. Varying the number of voltage rails (q) increases the weight range, and varying the weight rail voltage values modifies the weight resolution. Selecting combinations of rail voltages using S_1–S_6 provides 2^6 possible synapse weights where a sample of these combinations is shown in Table 2.

If we consider the first row in the table where all switches are closed, then each CTS is activated to a level dictated by the associated supply rail (weight). Therefore, each CTS will output a postsynaptic current whose magnitude is proportional to its weight voltage, and the net postsynaptic response is the aggregate of all CTS outputs. This is further illustrated in the remaining rows of the table where S_6 and S_1 are activated in rows two and three, respectively. In the former only one CTS is activated with a postsynaptic response proportional to a weighting of 2.56 volts whereas for the latter we have again a single CTS activated by S_1 with an output postsynaptic current proportional to a weight value of 0.08 volts. Note, it is envisaged that $q \neq p$, where more than one switch can be connected to a single voltage rail.

The authors recognise the need for a nonvolatile memory capability and are currently investigating the replacement of the latches/switches of Figure 6 with flash memory techniques using floating gates. What is novel about this approach to nonvolatile weight storage is that it only

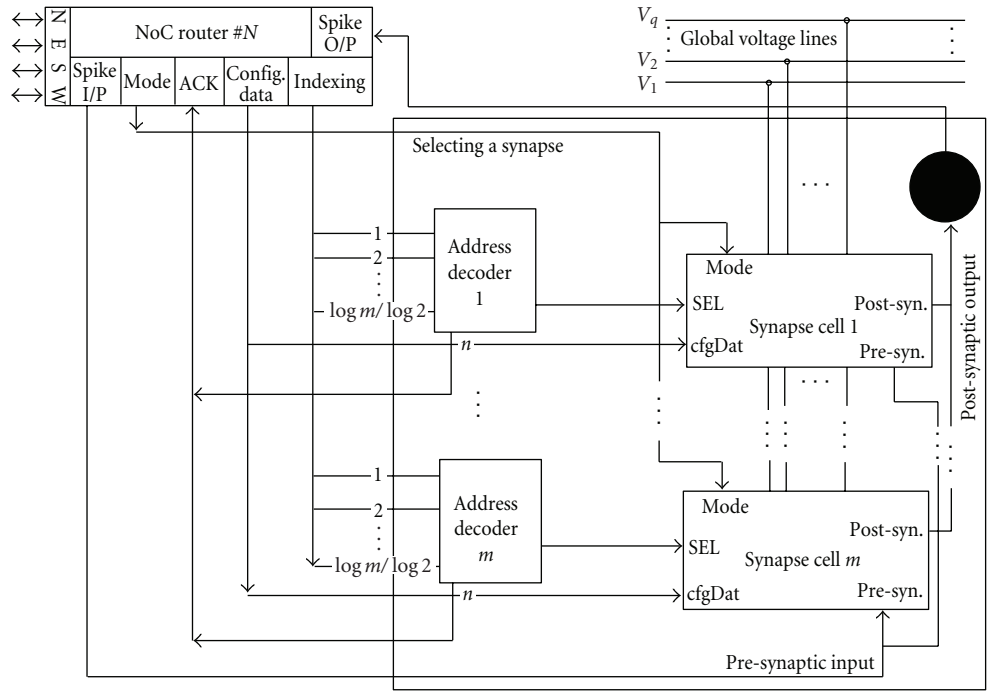


Figure 5: EMBRACE: neural tile structure.

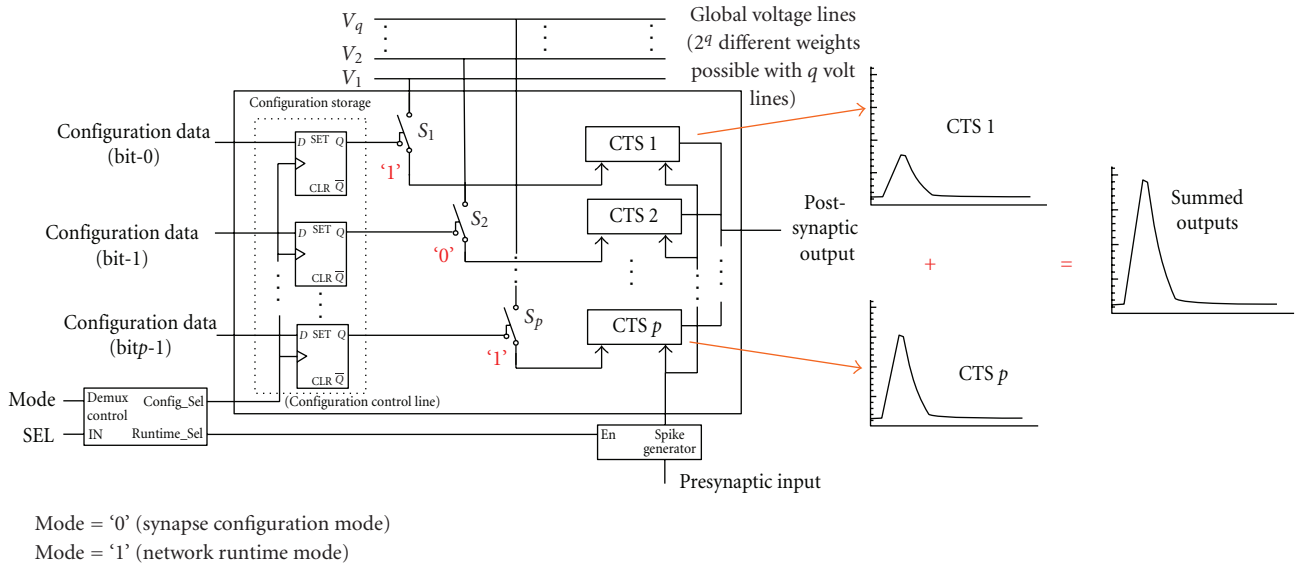


Figure 6: Synapse cell with p CTS.

requires that the transistors associated with each floating gate operate in either a fully on or off mode (binary operation) thereby avoiding the problems associated with dynamic range, sensitivity and, stability.

5. Embrace Performance

This section presents results on the functional evaluation of EMBRACE for the exemplar eXclusive-OR (XOR) problem. Results on the area performance of the neural tile are presented, and optimisations of the architecture for scalability are illustrated.

5.1. Demonstrative Application. The XOR (eXclusive-OR) problem has been used by researchers as a standard benchmark to verify the operation or evaluate the performance of artificial NNs [37] and more recently SNNs [38]. The XOR problem is based on the principle of logical pattern matching

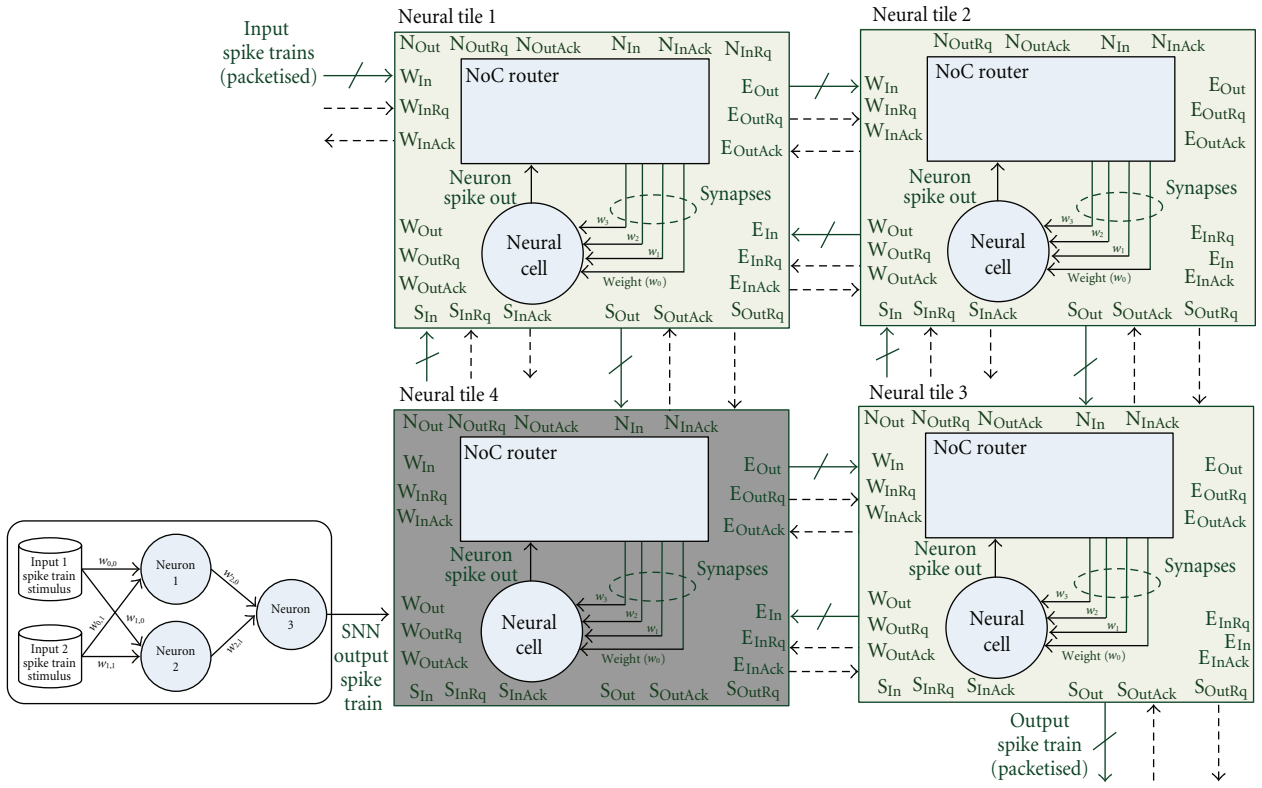


Figure 7: Embrace configuration for SNN-based XOR function.

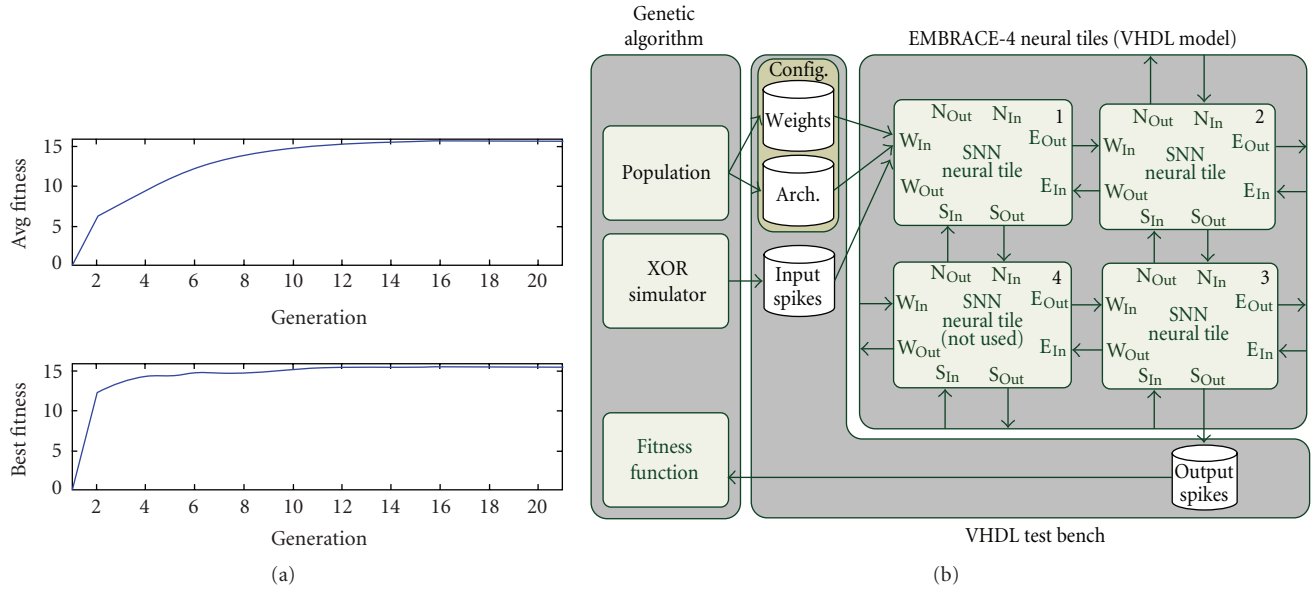


Figure 8: Evaluation setup of SNN-based XOR function (realised on 2×2 EMBRACE).

whereby a system must decipher whether or not an input pattern has an equal or unequal number of logic "0" or "1". For example, take the simple case where the XOR is applied to the binary pattern "11100100"; the system must be able to detect that the pattern has an equal number of zeros to be classed as an XOR solver. The XOR problem is not linearly separable and therefore an appropriate test case to exercise SNNs.

A 2-layer feed-forward SNN consisting of 3 neurons (2 input/1 output) was created to solve the XOR; whereby each neural cell utilised one point neuron and two synapse cells. Figure 7 illustrates the layout of the SNN-based XOR

configuration across a 2×2 array of neural tiles. In this example only 3 tiles are required, however; four programmable synapses (the maximum number expected for a 2 × 2 array) are included in each tile, one for each of the neurons in the architecture (5 CTSs per synapse cell). Note that although four synapse cells are shown in Figure 7, only two synapses per tile are utilised in the XOR problem. For larger problems additional tiles can be included by appending to the N, E, S and, W directions of the four example tiles, as depicted in Figure 1(b).

A hardware model of EMBRACE was created to evaluate its functionality in emulating the SNN-based XOR problem and, in particular, the configuration of the synapse weights and runtime switching of packet data between the NoC routers. Synthesisable VHDL was created for the digital NoC router and behavioural VHDL for the analogue synapse cells [39]. The XOR mapped EMBRACE model was simulated using commercial VHDL software tools running on a standard PC. Note: The SNN-based XOR was mapped to EMBRACE by allocating each single neuron and its multiple synapses to different tiles. The architecture was programmed by configuring the synapse weights and neuron threshold of each tile via a test bench. Figure 8 illustrates the test bed used to simulate the model of the XOR problem on the 2 × 2 tile array; a Genetic Algorithm (GA) was used to train the SNN to recognise the correct XOR input patterns over 20 generations. The GA fitness measure was based on the XOR function where a maximum of 16 was evaluated for all correct matched input patterns. The left-hand side of Figure 7 shows the convergence rate in generations and both the average and maximum fitness measures. Note that the GA-driven training process defines the inputs of known spike train patterns to neural tile 1 where each router forwards the target spike packets to their destination synapses in layer 1 (tiles 1 and 2).

For each of the postsynaptic neurons in layer 1 that fired, additional packets were generated by tiles 1 and 2 and then routed to the neuron in layer 2 (located in tile 3). Figure 9 illustrates the runtime mode of the tile with two packets being received and a spike train consisting of two spikes being directed to the 5 CTS, that is, one synapse cell.

During each GA generation, new synapse weights were reconfigured in the synapse cells using the NoC router of each tile and evaluated with the XOR fitness measure. This GA-driven training process was repeated until EMBRACE accurately performed the XOR function, that is, until an optimal or near-optimal set of weights were identified. The XOR evaluation of the 2 × 2 EMBRACE model proved successful as it was able to reconfigure weights, route packets, and perform the XOR operator on input patterns.

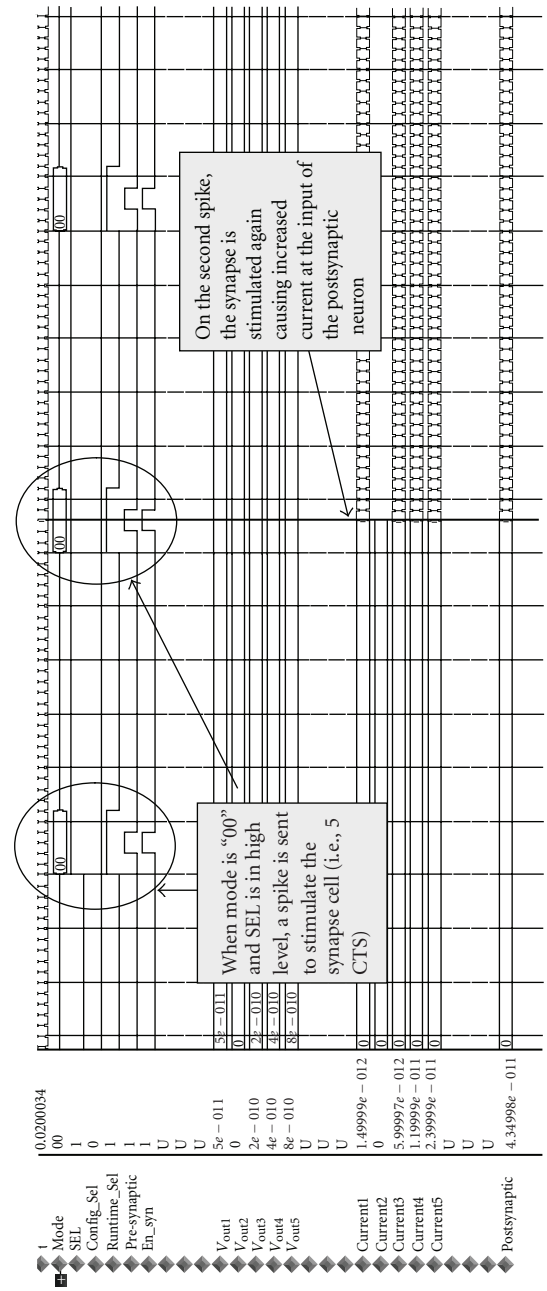


Figure 9: Runtime mode of the tile showing two packets being received. A spike train consisting of two spikes is directed to the 5 CTS. The total postsynaptic current of the individual CTS currents (current 1, 3, 4, and 5) increases in value with each spike stimulus.

5.2. Neural Tile Resources and Performance. The custom layout for an indicative neural tile with 10 synapse cells (each cell containing 10 CTSs) and a single point neuron [34] is illustrated in Figure 10. Note that the number of synaptic cells is set by the fan-in, and to avoid limiting the response time of the point neuron, the cells density was set to 10: a higher fan-in can easily be achieved by buffering the input of the point neuron circuit. The configuration storage, synapse voltage selection, and NoC router are not included in the layout. The 10 programmable synapse cells and neuron occupy a compact area size of 3 × 11 μm using 90 nm CMOS technology (test circuits for the programmable synapse and neuron are currently being fabricated by Europractice). The small synapse area indicates the potential to realise compact neural tiles. Moreover, the synapse operates in transient mode, and consequently the associated power

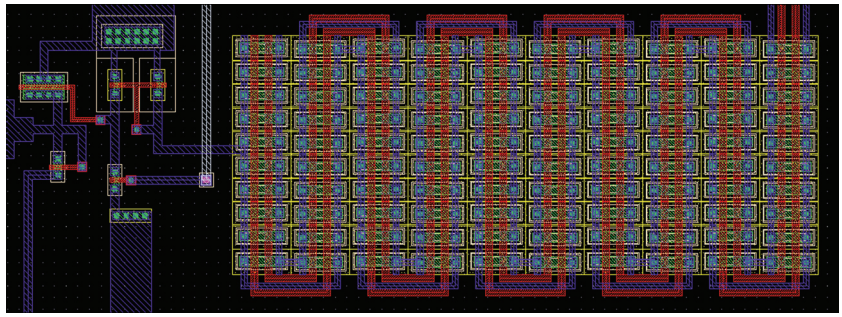


Figure 10: Neural Cell: layout of single point neuron and 10 synapse cells (10 CTS per cell).

consumption is small. Consider the extreme case where the CTS is stimulated by a 1 MHz presynaptic spike train and assume that the applied weight voltage to the CTS is 5 V (maximum weight). For each presynaptic spike, an average current spike of amplitude 10^{-7} amps is produced at the output of the CTS for a duration of 2 nanoseconds [34]. This yields a power consumption/cycle of 1 nW, which is orders of magnitude improvement when compared to circuit-based implementations of dynamic synapses [40]. Although the estimation does not include power consumption due to the tile's configuration circuitry, it does illustrate the application of the proposed synapse for low-power SNN implementation.

The initial NoC router design used in the XOR application has been implemented on a Xilinx XC2V1000 FPGA [33]. Data from [22] estimates the *AEthereal* NoC router implementation of 2658 LUTs on the Virtex-4 to be equivalent to 2.28 mm^2 using 90 nm CMOS technology. Using this data, a conservative estimate of 0.201mm^2 is determined for the area of the proposed router comprising 234 LUTs (synthesized for Virtex-4). This analysis provides an early area estimate of the proposed neural tile (single router, neuron and 10 synapse cells) at 201×10^{-3} mm^2. This initial estimate suggests promising results for EMBRACE networks, since the tile area is relatively small.

In terms of NoC router latency or performance, the router can process incoming data packets every 10 clock cycles with source packet generation requiring 12 cycles. The router was verified at 200 MHz on the Xilinx XC2V1000 providing 50 and 60 nanoseconds latencies, respectively. In the example of XOR problem the latency is negligible (~2.9 microseconds :10 spikes in the input train); however, for large scale network problem sizes the overall latency of EMBRACE is still faster than the biological spike-interval time, *ts*. Consider the network of Figure 1(a) with ($n = m = 10^3$) requiring 10^6 connections giving $m = 10^3$ (such network sizes are used in image processing tasks such as object tracking [3]).

Assume $ts = 10$ millisecondss [2], a clock frequency of 200 MHz ($tc = 5$ nanoseconds), and a 10 clock cycle period to receive and forward a packet between each NoC router; the time for the longest routing path, tp, can be estimated at $(m \times tc) = 5$ microseconds. Now assume the condition when the upper bound on data traffic load occurs with all neurons firing at the same. In consideration that each tile operates in parallel and multiple paths exist, the total latency of the 10^3 neuron layer is not an accumulation of $n \times m$ individual path times but rather an amortisation of path, times across the number of routing paths available. As each router can route data in all 4 co-ordinate paths the maximum number of parallel routes is $4 \times m = 10^3$. For the example network of 10^6 connections, the total latency with the upper bound condition will be $[10^6 \times tp/4\,m] = 5$ milliseconds. This demonstrates that the time required to time-multiplex the 10^6 connections between layers is significantly less than the 10 milliseconds interspike period and can therefore be performed in real-time. It should be noted that not all neurons spike at the same time [2, 4] so, a further speed improvement is possible.

5.3. Optimising EMBRACE. Research is currently underway to optimise the tile area further and so support the creation of dense EMBRACE architectures. For example, the addition of larger numbers of neurons and synapses within individual neural tiles will reduce the number of routers required for the SNN implementations; this has the effect of creating multicast groups where many neurons are assigned a single unique router address within the architecture. This strategy will allow reduction of the overall SNN area requirements as shown in Figure 11 (estimated by summing the router, synapse, and neuron areas). For example, EMBRACE can implement a fully connected 2-layered SNN with 10^4 neurons per layer, using a neuron/router ratio of 5 with an associated area of 642 mm^2 (shown by the dashed line in Figure 11). It can be seen that this scaling trend for increased neuron to router (N/R) ratios reduces the total device area of EMBRACE. The trade-off for minimising the number of routers (i.e., creating multicast groups) increased complexity in the router interface due to addressing and arbitration functions. This will increase the latency performance of the NoC router; however, the multicast groups will minimise the number of packet transmissions between routers and therefore minimise the overall network latency incurred.

Further optimisation of EMBRACE can be achieved by exploring the minimum number of CTS, p, per synapse cell. Figure 6 illustrated p CTS per cell where the dynamic range of the cell is defined by the number of CTS p and

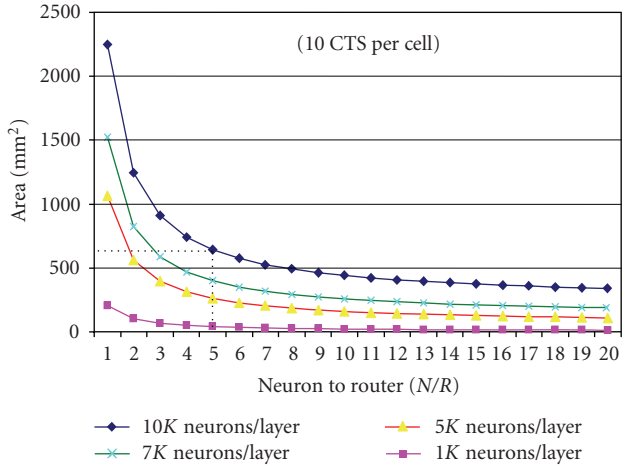


Figure 11: Optimisation: increasing number of neurons per tile.

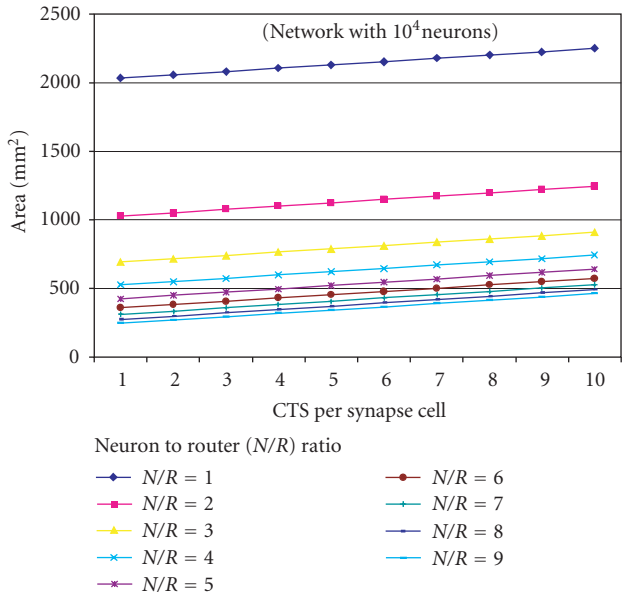


Figure 12: Optimisation: number of CTS per synapse cell.

voltage values V_q. Reducing the number of CTS per cell can reduce the EMBRACE area requirements. For example, in Figure 12 the total area for a network, with 10^4 neurons per layer and a N/R ratio of 5, can be reduced from ~642 to ~522 mm^2 by reducing the number of CTS per cell from 10 to 5, respectively. This linear reduction in area is consistent for other N/R ratios. Further research is currently underway to identify the optimum number of p CTS per synapse cell.

5.4. Scalability. Figure 13 illustrates the total EMBRACE area as a function of neuron density with a neuron/router ratio of 5. Since the proposed NoC supports a regular layout of the tiles and neuron communication, the interconnectivity between layers using EMBRACE does not limit the network size that can be implemented, unlike existing strategies. The

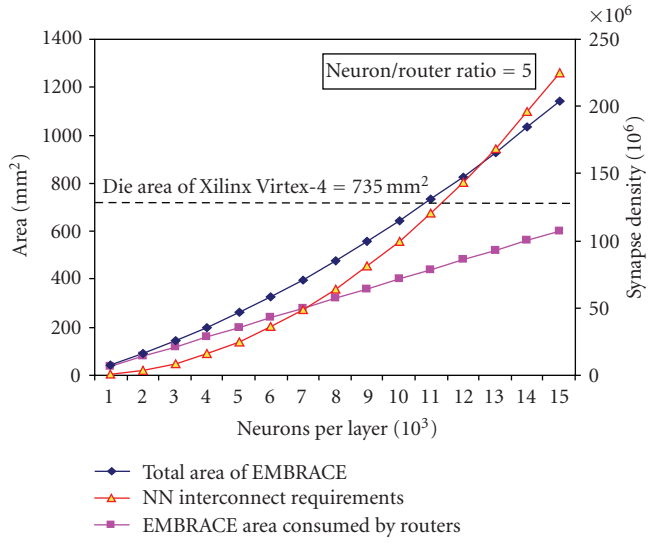


Figure 13: EMBRACE scalability.

figure also illustrates the linear growth of the area occupied by the routers with the SNN interconnect requirements. Although Figure 13 illustrates a synaptic density of the order of 10^8 for a die area equivalent to the Xilinx Virtex-4 FPGA [15], the calculations underestimate the total EMBRACE area due to configuration circuitry and global chip wiring. Results do however highlight a scaling trend, indicative of the proposed architecture.

6. Discussion

The adaptability of EMBRACE provides a framework to exploit the ability to reconfigure itself and also provide alternative routing paths for damaged areas using NoCs. For example, two levels of fault tolerance could be supported, namely, at the synapse and tile levels.

The abstract basis of SNNs is the strengthening and weakening of synaptic weights, whereby training algorithms are used to derive appropriate weight values to reflect a mapping between the input and desired output data. Faults occurring in individual synapses could be tolerated by using such algorithms to appropriately retrain the network when the output deviates from the "golden" patterns. This process can be achieved via the strengthening and/or weakening of neighbouring synapse weights within the tile.

At a more coarse level, complete tiles could be remapped or relocated to fault-free tiles, whereby the configurable data of a damaged tile is reconfigured to a new tile with updated router address contents and synaptic weights. For example, the adaptability of each neural tile could be exploited to allow its contents to be updated during runtime by any of its four coordinate neighbours. When a fault is detected in one of the neighbouring tiles, using a similar fault detection scheme such as [41] the centre tile could take control and relocate the configuration data of the faulty tile to a new available tile. An address update packet could then be broadcast to all tiles indicating the new location of the repaired tile. This

approach would therefore provide a more robust distributed repair mechanism as opposed to a centrally controlled strategy.

Such a fault tolerant computing paradigm has the potential to be used in aerospace avionics (e.g., safety systems in space-craft or aeroplanes) or automotive electronics (e.g., engine management systems) where exposure to harsh environmental conditions requires adaptive computing systems.

7. Summary and Future Work

The challenges for large scale SNN implementations on reconfigurable platforms have been highlighted. A novel EMBRACE hardware architecture has been presented which utilises NoC routers and novel synapse cells to provide the programmable and scalable interconnect and biocomputational resources, required in large scale SNNs. Results have been presented, which demonstrate the functionality and performance of EMBRACE using the benchmark XOR problem. Similarly, results on the scalability of EMBRACE in terms of area and power have been presented and the capability of the EMBRACE architecture to support dense SNNs has been illustrated. Overall, the approach has been proven to be very promising.

Future work will explore NoC routing polices and topologies for SNN spike traffic. Optimising the number of neurons per neural tile will be tested, and the application of asynchronous NoCs will be investigated to support the QoS for spike-event traffic. Approaches to on-chip training will also be investigated to support the programming of the synapse cells and router. Integration of the neural tile configuration architecture and router with the analogue neural cell will be realised with the aim of developing a full-scale EMBRACE implementation. Additionally, the adaptability of EMBRACE provides an ideal framework to further explore bioinspired fault tolerant computing paradigms, whereby on-chip SNN learning and reprogrammability of EMBRACE neural structures could potentially emulate the repair behaviour of the brain. Future work will also explore the requirements to support on-chip, self-adaptation for fault tolerance computing applications.

References

[1] M. A. Arbib, *The Handbook of Brain Theory and Neural Networks*, A Bradford Book, Bradford, UK, 1995.

[2] W. Maass, *Computation with Spiking Neurons: The Handbook of Brain Theory & NNs*, MIT, Cambridge, Mass, USA, 2001.

[3] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin, "Challenges for large-scale implementations of spiking neural networks on FPGAs," *Neurocomputing*, vol. 71, no. 1–3, pp. 13–29, 2007.

[4] H. Markram, "The blue brain project," *Nature Reviews Neuroscience*, vol. 7, no. 2, pp. 153–160, 2006.

[5] A. Ghani, T. M. McGinnity, L. P. Maguire, and J. Harkin, "Area efficient architecture for large scale implementation of biologically plausible spiking neural networks on reconfigurable hardware," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 939–940, Madrid, Spain, August 2006.

[6] B. Glackin, et al., "Novel approach for the implementation of large-scale spiking neural networks on FPGAs," in *Proceedings of the Artificial Neural Network Conference*, pp. 552–563, 2005.

[7] A. Upegui, C. A. Peña-Reyes, and E. Sanchez, "An FPGA platform for on-line topology exploration of spiking neural networks," *Microprocessors & Microsystems*, vol. 29, no. 5, pp. 211–223, 2005.

[8] M. J. Pearson, A. G. Pipe, B. Mitchinson, et al., "Implementing spiking neural networks for real-time signal-processing and control applications: a model-validated FPGA approach," *IEEE Transactions on Neural Networks*, vol. 18, no. 5, pp. 1472–1487, 2007.

[9] E. Ros, E. M. Ortigosa, R. Agis, R. Carrillo, and M. Arnold, "Real-time computing platform for spiking neurons (RT-spike)," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 1050–1063, 2006.

[10] A. DeHon and R. Rubin, "Design of FPGA interconnect for multilevel metallization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 10, pp. 1038–1050, 2004.

[11] S. Furber, "Future trends in SoC interconnect," in *Proceedings of the IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test (VLSI-TSA-DAT '05)*, pp. 295–298, Hsinchu, Taiwan, April 2005.

[12] L. Benini and G. DeMicheli, "NoCs: a new SoC paradigm," *IEEE Computers*, pp. 70–78, 2002.

[13] R. J. Vogelstein, U. Mallik, J. T. Vogelstein, and G. Cauwenberghs, "Dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 253–265, 2007.

[14] C. Schuck, S. Lamparth, and J. Becker, "artNoC—a novel multi-functional router architecture for organic computing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 371–376, Amsterdam, The Netherlands, August 2007.

[15] Xilinx, Virtex-4 User Guide, Ver. (V1.3), 2005.

[16] C.-S. Bouganis, P. Y. K. Cheung, and L. Zhaoping, "FPGA-accelerated pre-attentive segmentation in primary visual cortex," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 199–204, Madrid, Spain, August 2006.

[17] A. Ye and J. Rose, "Using bus-based connections to improve field-programmable gate-array density for implementing datapath circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 5, pp. 462–473, 2006.

[18] G. S. Snider and R. S. Williams, "Nano/CMOS architectures using a field-programmable nanowire interconnect," *Nanotechnology*, vol. 18, no. 3, 2007.

[19] W. J. Bainbridge, L. A. Plana, and S. B. Furber, "The design and test of a smartcard Chip using a CHAIN self-timed Network-on-Chip," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, pp. 274–279, Paris, France, February 2004.

[20] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda, "CuNoC: a scalable dynamic NoC for dynamically reconflgurable FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 753–756, Amsterdam, The Netherlands, August 2007.

[21] I. Nousias and T. Arslan, "Wormhole routing with virtual channels using adaptive rate control for network-on-chip (NoC)," in *Proceedings of the 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS '06)*, pp. 420–423, Istanbul, Turkey, June 2006.

[22] K. G. W. Goossens, et al., "Hardwired NoCs in FPGAs to unify data and configuration interconnects," in *Proceedings of the International Symposium on NoCs*, pp. 45–54, 2008.

[23] C. Hilton and B. Nelson, "PNoC: a flexible circuit-switched NoC for FPGA-based systems," *IEE Proceedings: Computers & Digital Techniques*, vol. 153, no. 3, pp. 181–188, 2006.

[24] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda, "CuNoC: a scalable dynamic NoC for dynamically reconflgurable FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 753–756, Amsterdam, The Netherlands, August 2007.

[25] Silistix, "CHAINworks," 2008, http://www.silistix.com/.

[26] Arteris, "Danube NoC IP," 2005, http://www.arteris.com/.

[27] M. M. Khan, D. R. Lester, L. A. Plana, et al., "SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '08)*, pp. 2849–2856, Hong Kong, June 2008.

[28] A. D. Rast, S. Yang, M. Khan, and S. B. Furber, "Virtual synaptic interconnect using an asynchronous network-on-chip," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '08)*, pp. 2727–2734, Hong Kong, June 2008.

[29] B. Glackin, J. Harkin, T. M. McGinnity, and L. P. Maguire, "A hardware accelerated simulation environment for spiking neural networks," in *Proceedings of the Applied Reconfigurable Computing Workshop*, pp. 336–341, 2009.

[30] P. A. Merolla, J. V. Arthur, B. E. Shi, and K. A. Boahen, "Expandable networks for neuromorphic chips," *IEEE Transactions on Circuits and Systems I*, vol. 54, no. 2, pp. 301–311, 2007.

[31] J. Schemmel, et al., "Mixed-mode analog NN using current-steering synapses," *Analog Integrated Circuits & Signal Processing*, vol. 38, 2004.

[32] G. Indiveri, "A neuromorphic VLSI device for implementing 2D selective attention systems," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1455–1463, 2001.

[33] J. Harkin and M. McElholm, "Novel interconnect strategy for large scale implementations of NNs," *IEEE Soft Computing in Industrial Applications*, July 2007.

[34] Y. Chen, L. McDaid, S. Hall, and P. Kelly, "A programmable facilitating synapse device," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '08)*, pp. 1615–1620, Hong Kong, June 2008.

[35] Y. Chen, et al., "A silicon synapse based on a charge transfer device for SNNs," in *Proceedings of the 3rd International Symposium on Neural Networks (ISNN '06)*, Chengdu, China, May 2006.

[36] D. E. Johnson, J. S. Marsland, and W. Eccleston, "Neural network implementation using a single MOST per synapse," *IEEE Transactions on Neural Networks*, vol. 6, no. 4, pp. 1008–1011, 1995.

[37] H. M. El-Bakry, "Modular neural networks for solving high complexity problems," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '03)*, vol. 3, pp. 2202–2207, Portland, Ore, USA, July 2003.

[38] O. Booij and H. Tat Nguyen, "A gradient descent rule for spiking neurons emitting multiple spikes," *Information Processing Letters*, vol. 95, no. 6, pp. 552–558, 2005.

[39] P. Rocke, B. McGinley, et al., "Investigating the suitability of FPAAs for evolved hardware spiking neural networks," in *Proceedings of the International Conference on Evolvable Systems (ICES '08)*, pp. 118–129, 2008.

[40] G. Indiveri, E. Chicca, and R. Douglas, "A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity," *IEEE Transactions on Neural Networks*, vol. 17, no. 1, pp. 211–221, 2006.

[41] J. Harkin, P. Dempster, T. M. McGinnity, and B. Cather, "Fault Detection Strategy for Self-Repairing Systems," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics UK Chapter (SMC '07)*, pp. 23–28, Dublin, Ireland, September 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 161458, 9 pages
doi:10.1155/2009/161458

Research Article

Enabling Self-Organization in Embedded Systems with Reconfigurable Hardware

Christophe Bobda,[1] Kevin Cheng,[1] Felix Mühlbauer,[1] Klaus Drechsler,[2] Jan Schulte,[2] Dominik Murr,[2] and Camel Tanougast[3]

[1] *Computer Engineering Group, University of Potsdam, 14482 Potsdam, Germany*
[2] *Kaiserslautern University of Technology, Kaiserlautern, Germany*
[3] *P. Verlaine University of Metz, 57000 Metz, France*

Correspondence should be addressed to Jan Schulte, jan@janschulte.com

Received 31 December 2008; Accepted 23 July 2009

Recommended by J. Manuel Moreno

We present a methodology based on self-organization to manage resources in networked embedded systems based on reconfigurable hardware. Two points are detailed in this paper, the monitoring system used to analyse the system and the Local Marketplaces Global Symbiosis (LMGS) concept defined for self-organization of dynamically reconfigurable nodes.

1. Introduction

Increasing complexity due to rapid progress in information technology is making systems more and more difficult to integrate and control. Due to the large amount of possible configurations and alternative design decisions, the integration of components from different manufacturers in a working systems cannot be done only at design time anymore. Systems must be designed to cope with unexpected runtime environmental changes and interactions. They must be able to organize themselves to adapt to changes and avoid nondesirable or destructive behaviors. In heterogeneous environments devices with different abilities are available. An example would be a sensor network consisting of powerful nodes called beacons doing all the computational work and collecting environmental information from less powerful sensor nodes, which of course could also do some in-network processing of the collected data. The ongoing progress in chip technology provides us with steadily increasing computation power whereas the accompanying miniaturization shrinks the device sizes to unprecedented measures at rock bottom prices. It is obvious that even the weakest sensor node will have enough computation power to share its resources with other nodes in the near future by offering its resources as services to them.

In this work we try to answer the question how nodes in networks may utilize reconfigurable hardware and share their resources with other nodes to optimize load and dependability, thus adapting to a changing environment. To answer this question we investigate self-organization to implement a distributed resource management in these kind of networks. Self-organization is naturally found in biological systems and can be defined as a process in which pattern at the global level of a system emerges solely from numerous interactions among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern [1]. If we translate this definition of self-organization into embedded systems we get something which has been characterized as the vision of autonomic computing [2]. There an autonomic system is a collection of so-called autonomic elements which is an individual system that contains resources and delivers services to humans and other autonomic elements. Figure 1 shows the building blocks of an autonomic element. It consists of an autonomic manager which controls one or

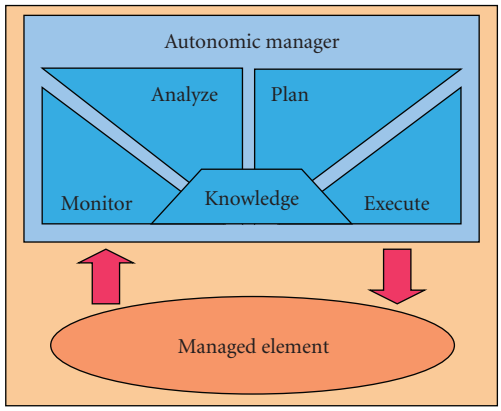


Figure 1: An autonomic element as presented in the vision of autonomic computing [2].

more managed elements. The autonomic manager gathers information about the managed element and its surrounding environment by using a monitor and constructs and executes plans based on an analysis of this information. Prior learned knowledge can be used in these steps to construct more intelligent plans, thus behaving more intelligent. Not all of these blocks are necessary to build an autonomic element. It depends heavily on the application to decide which block should be implemented and how.

In our system the *monitor* is implemented as a standalone application which gathers information from the local system and offers these information to neighbour nodes and in turn also gathers information from them. Because the monitor is extensible, the type of information a monitor can offer and collect is manifold and ranges from system information like CPU load, memory usage, used/unused reconfigurable area to positioning information, and offered services. A service a node may offer is, for example, its computational power. These information together with information from a knowledge database, that is built upon antecedent experiences, is used by a resource manager, that implements the *analyze* and *plan* blocks (see Figure 1), to distribute tasks and optimize resource utilization or power consumption in the network. In order to enforce flexibility in such systems and allow single nodes to adapt their behavior to disturbances, we use reconfigurable units, in this case FPGAs. Those devices pose a decent trade-off between computation power, energy consumption, and flexibility. They can be configured and reconfigured to provide hardware acceleration for highly specialized services. Furthermore, they can be partially reconfigured while keeping the rest of the system working. Some are even capable of initiating their reconfiguration themselves keeping the part of their logical circuits that hosts the local operating system up and running while only a small partition that hosts specialized accelerators is being reconfigured [3]. The *execute* block (see Figure 1) implements something like a task manager or reconfiguration manager. It is able to take a piece of hardware (we call it hardware module) either from a local storage or from any other node in the network and reconfigure itself to execute a computation-intensive task.

To investigate the principles of self-organization in networked embedded systems, we have developed a flexible and extensible monitor to track the state of the surrounding neighbours, which serves as the basis for a powerful and flexible approach to manage the available resources in networked embedded systems with reconfigurable hardware. A framework is shown that enables nodes to (re)distribute tasks in a marketplace-like manner, which delivers the basis technology to elevate implementations of collective task completion to a new level. Here a node that recognizes the need for a certain task to be done formulates it as a query to itself and its neighbors. Every node that offers the execution of a task replies to a query with its cost for fulfilling the job. The inquirer is now able to choose whether it is more appropriate to maybe reconfigure and do the task by itself or to delegate.

Both worlds, namely, self-organization in embedded systems and embedded systems built around reconfigurable hardware, lead to highly adaptive distributed systems.

The remaining part of this paper is structured as follows. The next sectiongives an introduction into the technological basis we have used to develop and test our system (Section 1). It follows a description of the implemented monitor (Section 2) and an explanation of a framework, which enables nodes to redistribute tasks in a marketplace-like manner (Section 3). Finally we conclude this article and discuss future work.

2. Technological Basis

2.1. FPGA and Partial Reconfiguration. The essential resources that must be available on each node in a distributed cooperative system are a minimal local computation power, a resource management, the ability to communicate and a system to distribute or gather jobs in the network.

Concerning the computation power, to propose a flexible and powerful node, FPGA technology is used. As described by Xilinx [4], their S-RAM-based FPGAs are configured by loading application-specific configuration data: the bitstream into an internal memory. The bitstream is a binary encoded file, that is, the equivalent of the design in a form that can be downloaded into the FPGA device. This basic idea of FPGAs is the most important. It means that it is possible to create an electronic design based on an FPGA remotely through the network.

To develop and test our concepts we use a Virtex-II Pro FPGA sitting on the XUP development board, a Xilinx ML403 board as well as the hydraXC-50 module which are both equipped with a Virtex-4 FX12 FPGA, and a Xilinx ML505 board equipped with a Virtex-5 LX50T.

Besides the configurable logic cells, present inside the FPGA, that allow custom hardware accelerators, the Virtex-II Pro, Virtex-4, and Virtex-5 supply a certain amount of basic, hard-wired circuits (primitives) to extend the device's speed and effectiveness. These are, for example, multipliers, block RAM, and especially a module named ICAP: Internal Configuration Access Port (see Figure 2). This module is the key for a node to change its own reconfigurable logic; it is an internal access to the configuration memory where the

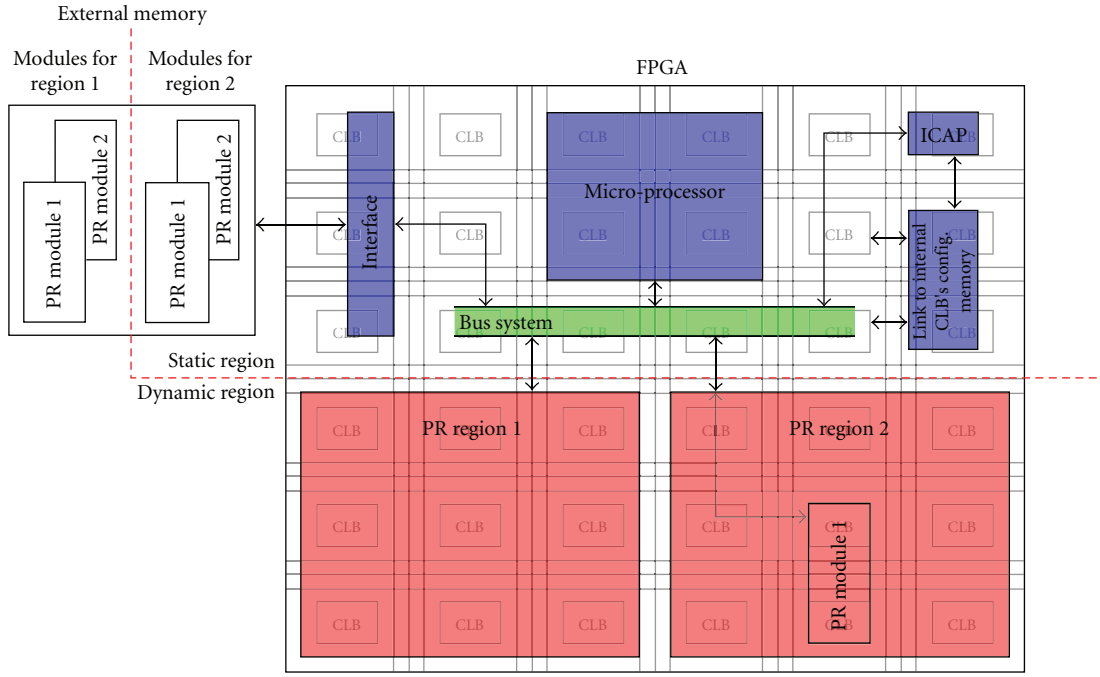


Figure 2: Description of Dynamically Reconfigurable SoC with ICAP interface.

current bitstream is stored. In addition, Xilinx FPGAs are capable of partial reconfiguration since the Virtex-II series. That means that single hardware modules can be exchanged while vital parts of the node, like the memory controller or the network interface controller, keep on operating uninterruptedly.

To realize a partial reconfigurable system [5], Partial Reconfigurable Regions (PRRs) have to be defined inside the FPGA. This PRRs are regions where Partial Reconfigurable Modules (PRMs) can take place. PRMs represent the electronic circuit of functional units which can be placed according to the application. As for a standard design that uses bitstreams, a PRM is represented by a partial-bitstream. This partial-bitstream can be stored in the system memory (Compact flash card or PROM) and transferred into the configuration memory when it is necessary. Then, it is easy to consider partial-reconfiguration management as a file management. Indeed, bitstreams can be manipulated like common file and send via a network.

All PR Regions and Modules must have exactly the same interface to be connected with the static part of the design. This particularity is a limitation in the design for partial reconfigurable hardware but it is the key to be able to change a part of the design at run-time. This interface between PR Regions and the Static Region (SR) is designed using Bus Macros (BMs) which are used to create a link between the reconfigurable and the fixed logic of the design. They are also fixed to be able to exchange PR Modules and to know where to connect I/Os precisely.

When designing a partial reconfigurable system, the border between static and dynamic parts must be defined. Inside the dynamic part, the area size and the position of each PR Region are also determinant [6]. The development of such system has a direct influence on the quantity and the size of bitstreams.

PR Modules are synthesized specially for a certain PR region. In consequence, each PR Module is synthesized for each region. This allows the relocation of a module in all PR Regions, locally or in another node. In such systems, the number of partial bitstreams can rapidly increase.

The size of a bitstream a direct impact on the systems performance. Moreover, FPGA resources like CLBs or BRAMs [7] are directly related with the configuration memory [4]. As explained before, the configuration of an FPGA is done by loading a bitstream into the internal configuration memory (see Figure 2). Thus, the time necessary to reconfigure the FPGA depends on the bitstream's size.

Virtex-4 and Virtex-5 boards have been used to test the partial-reconfiguration, in a network context, and to implement a self-organisation concept. The first implementation is the control of the partial reconfiguration of one board by another. On each board, a System on Chip (SoC) is implemented. It includes a MicroBlaze soft-core processor used to run a software that controls the reconfiguration via the ICAP port. Using a custom link between boards, a command is sent to request a dynamic reconfiguration. On each board, bitstreams and partial-bitstreams are stored in a compact flash card.

Beside that, a hydraXC board has been used in conjunction with a MicroBlaze soft-core CPU and an uClinux operating system to rapidly develop the monitoring system as described in the next section.

2.2. System Development and Linux. Linux was also used as general purpose operating system on the other development

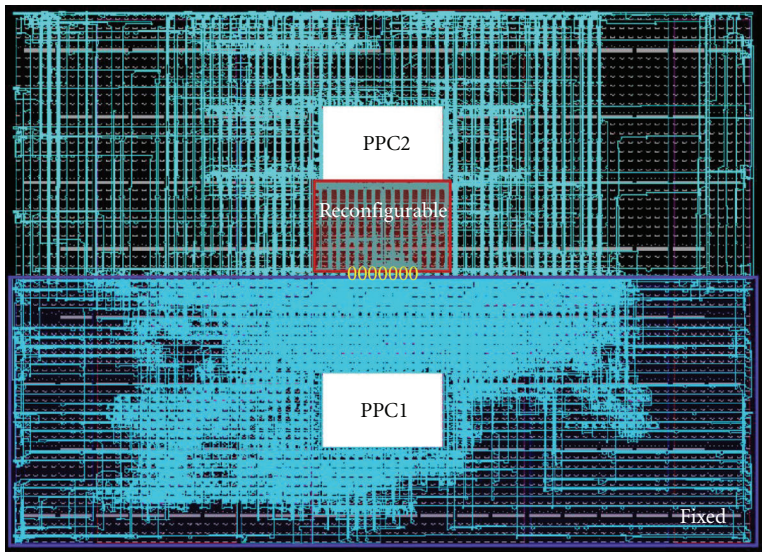


Figure 3: Layout of our sample system: the small highlighted area is reconfigurable.

boards. Linux serves our needs in several ways. It encapsulates the difficulties to access hardware and thus facilitates resource deployment through its abstract driver interfaces. For example, the ICAP is incorporated into the Linux system with John Williams' device driver [3]. Especially the fully developed networking abilities are a convenient way to realize communication. The innumerable quantity of applications for any kind of need and furthermore adaption and development of software in high-level programming languages pose a big plus. The use of a standard Linux kernel and applications also speeds up development for another reason: the large community that exists and the vast database of solutions to standard problems.

Linux provides us with a fully featured TCP/IP-stack that we have used as a basis for developing and prototyping our applications. We use tools like *telnet* to control devices remotely and *wget* to transfer data. It is clear that certain application fields have to adapt the used communication protocols to fulfill their needs, for example, energy awareness.

Using partially reconfigurable hardware devices we are in the position to equip our devices with a virtually adaptive behaviour. To keep development easy at the beginning our nodes provide only one region of fixed size that can be reconfigured separately from the rest of the system as shown in Figure 3.

With this, we are able to realize a hardware-software codesign for optimized performance. Tasks that can efficiently be computed in hardware may be executed in a specialized hardware module. Other tasks that are not available as a hardware module or are more efficiently solved in software are then executed on the local CPU. We use the ICAP to allow the CPU to reconfigure the device itself to utilize free reconfigurable space or to replace unused hardware modules. The individual situational behavior of a device is determined by the controlling application which decides whether to replace a local hardware accelerator or to employ a neighbouring node according to a set of parameters stored locally in simple files. These decisions are based on the analysis of the surrounding neighbourhood and constructing plans based on these information as explained in the introduction.

3. Targeted Application

Sensor networks used to consist of many very small sensor nodes to collect data and at least one data sink that gathers all the information and offers an interface to other networks. In scenarios with more complex data aggregations beacons are introduced which provide more computation power, collect data, and transmit preprocessed information to the destination.

Our targeted application is a sensor network that is able to track an object optically over a large area where several cameras have to cooperate to keep the target within sight. A sensor node is now equipped with a video camera and has a certain computation power to extract features of object recognition from the taken pictures. Information about what the cameras detect will be send to the user's terminal.

For this, we built up a sample application for the XUP development board that captures the video signal from a camera, digitizes it, and applies a filter to the stream to provide it for further processing. The whole task is solved in hardware on the FPGA with the filter being partially reconfigurable at run-time. Figure 3 shows the physical layout of our sample system. There the small boxed area is reconfigurable, whereas the bigger region encloses fixed logic like the Ethernet controller or the SD-RAM controller. BusMacros are represented between the reconfigurable and the static part (yellow). The lighter lines on the picture are wire connections between logic blocks.

Since our system is flexible and expandable, the loss of camera nodes or computation nodes (beacons) can be compensated to a certain degree in terms of surveyed

area as well as computation power for picture analysis. This is achieved by distributing the work the failing node contributed to other units. Also new camera nodes or beacons can be added as the task of the network and therefore the demand for computation power changes.

4. Monitoring System

We developed a monitoring system that consists of three parts. The main component of the system is the monitor daemon. This daemon permanently monitors the neighbourhood. The second part is an interface, which we have implemented in c++ and java for simplifying access to the daemon. The third and last component is a java application used to log into a daemon to display its knowledge about its neighbourhood.

4.1. Daemon. The monitor daemon runs in an uClinux environment as a background service. Other applications may access the daemon by using a simple interface. The task of the daemon is to permanently collect information from and about other devices in the neighbourhood and simultaneously broadcast its own state. Information a device may gather is

(i) devices in the neighbourhood,

(ii) their state (e.g., CPU load, memory usage, position, etc.),

(iii) what capabilities they have (e.g., services they offer).

4.1.1. Device Discovery—Recognition of Surrounding Devices. The daemon periodically broadcasts a ping message to inform other devices, that it is still alive. This is done by every device in the network. When the daemon receives a ping message from another device A, it updates its list of known neighbours. If device A is not yet in the list, it is added to the list and a counter for that device is set. If device A is already in the list, the counter is reset to a standard value. Therefore the daemon keeps a list with all known neighbours with a counter value for each node. The daemon now periodically walks through the list in a specified interval (e.g., every 10 seconds) and decrements each counter. When a counter of a device, let us say device B, has reached a given limit, a ping message is sent to node B. If device B does not respond with a ping message within a given time, device B is removed from the list of known neighbours. In this case an event can be generated to inform other devices of this situation.

4.1.2. Service Discovery—Retrieving a Node's Capabilities. As well as the ping messages, each device in the network periodically broadcasts a list of information, that it is offering. For example, device A may have a list of its own hardware accelerators. Device A then broadcasts, that it is capable of sending a list of its hardware accelerators. Another device B may be able to give information about its state, like CPU load, memory usage, and so forth. Then device B periodically sends a message that other devices can retrieve the system state of device B. Therefore each node in the network knows which information it can obtain from its surrounding nodes.

While ping messages are sent in quite a short interval, the lists are broadcasted less often, since we do not expect them to change very often in stable environments. It is thinkable that the frequency of broadcasting these lists is gradually adapted when a node detects an unstable environment (e.g., moving nodes).

4.1.3. Being Communicative—Obtaining Information. Obtaining information is done in a published/subscribed manner. If a device A is interested in receiving information from a device B, for example, the CPU load, device A subscribes to this information. Device B then periodically broadcasts the requested information. If another device C is interested in the same information, device C also subscribes to this information; however, device B recognizes the subscriptions being the same and only sends the information once every period. This mechanism minimizes network traffic.

Basically the daemon consists of two interfaces, a node-to-node (NTN) and a node-to-application (NTA) interface, a tiny database, and a set of plug-ins. The daemons communicate with each other using the NTN interface (see Figure 4).

If an application wants to get information from a daemon, it uses the c++ or java interface (as described in the next section) to connect to the NTA interface and collect the desired information. Using two interfaces, the complexity of the protocol for node-to-node communication can be kept small. The database is needed for storing information from neighbours and allows simple search queries. So far, we have explained how distribution of information is done in the network, but where do the information come from? Therefore the daemon uses a plug-in mechanism. For instance, there may be a plug-in that provides information about the available hardware accelerators, or a plug-in to catch system information from the underlying operating system. Additional plug-ins can easily be integrated in the system to enhance the daemons functionality.

4.2. The c++/Java Interface. The interface provides access to the daemon for information interchange. This is basically a c++ library which can be included by other applications, but also a lightweight java version has been implemented. Therefore the interface provides a basic set of functions to retrieve information from the daemon and to configure the way it monitors the surrounding environment. This, for instance, makes it possible to retrieve the list of known neighbours by a single function call. The communication between the interface and the daemon service is done by using UDP sockets. This allows connections to daemons that are more than one hop away by using a routing service or a lower level routing protocol. The complexity of retrieving information from nodes that are far away can hereby be taken out of the monitor daemon.

Getting information from the monitor daemon using the c++/java interface is done in a polling way. Therefore, if an application is interested in information (e.g., the hardware

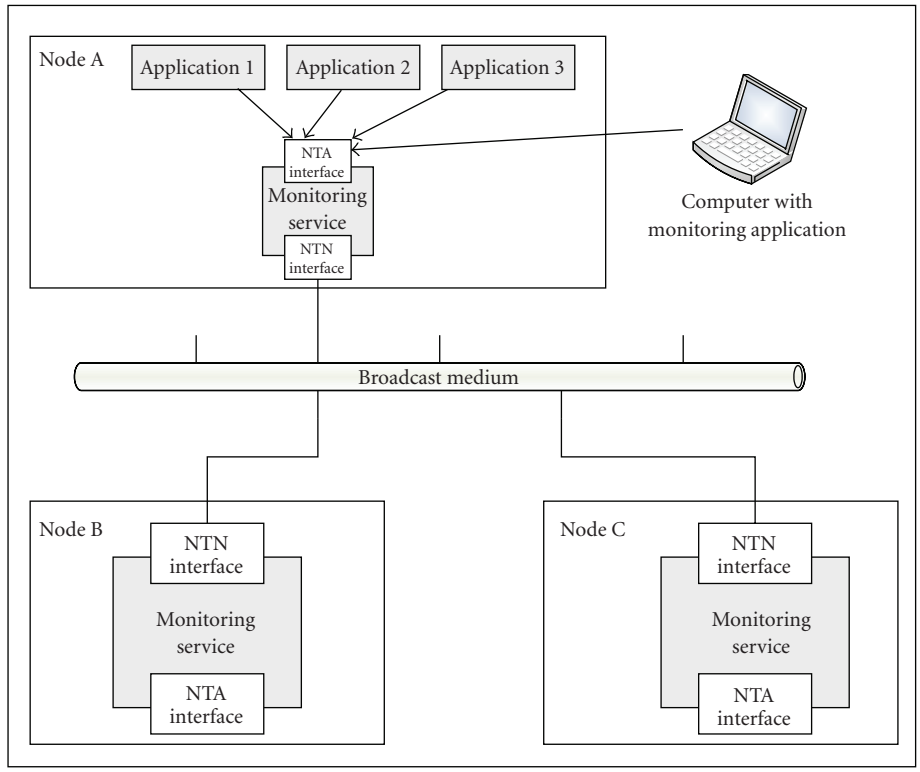


Figure 4: Communication between monitor daemons using the NTN interface and with applications using the NTA interface.

accelerators of node A), it must explicitly call a function to retrieve the information. However, we have also implemented an event-mechanism to push critical information to connected applications. This might be the disappearance of a neighbour device A, which causes the system to reorganize itself to adapt to the changing environment. Therefore the daemon system may send crucial system events to any connected application.

4.3. Monitoring-Application. The last component of the monitor system is a java-based application. Using this application, it is possible to log into a monitor-daemon and display the daemon's knowledge of its environment (see Figure 5).

By graphically displaying the neighbouring devices with their information, we have developed a simple tool to observe the system-behaviour during run-time.

5. Local Marketplaces Global Symbiosis

We defined a simple concept to deploy our new features: LMGS—Local Marketplaces Global Symbiosis. The work is distributed according to principles of supply and demand within the network.

The simple idea is that every device does exactly what it deems to be the best according to its stored parameters. The minimal software to actively "take part in the game" consists of two elements: a customer which issues requests to the network to have a certain job done and a purveyor that answers requests for jobs with the costs it charged if it would be commissioned. The effort a device makes to create the answer can vary widely. According to its computation power, knowledge, and storage capacity this can reach from a simple return of standard values to a complex measuring where the load, the utilization of the node's components, or the probability for the effectiveness of anticipated reconfiguration might be taken into account. In sensor networks, for example, communication is the most expensive action; so a distribution of work should be chosen that minimizes overall traffic, maybe by executing a lot of tasks locally through reconfiguring the node.

5.1. Requests. Requests issued by the customer component of a node consist of a tuple as follows:

$$Request = \{Source, Target, Requester, Data\_Volume, Task, Max\_Hops\}, \quad (1)$$

where *Source* contains the data source for the task, *Target* the data sink, and *Requester* the device which posted the inquiry. *Data_Volume* holds the quantum of data to be processed with *Task*. *Max_Hops* specifies the maximum number of times a request may be relayed by a node's neighbors.

Source, *Target*, and *Requester* may be partly or completely identical, given they are not, we included mechanisms to distribute jobs within a channel between data source and target. The values can be one-to-one identifiers of nodes or

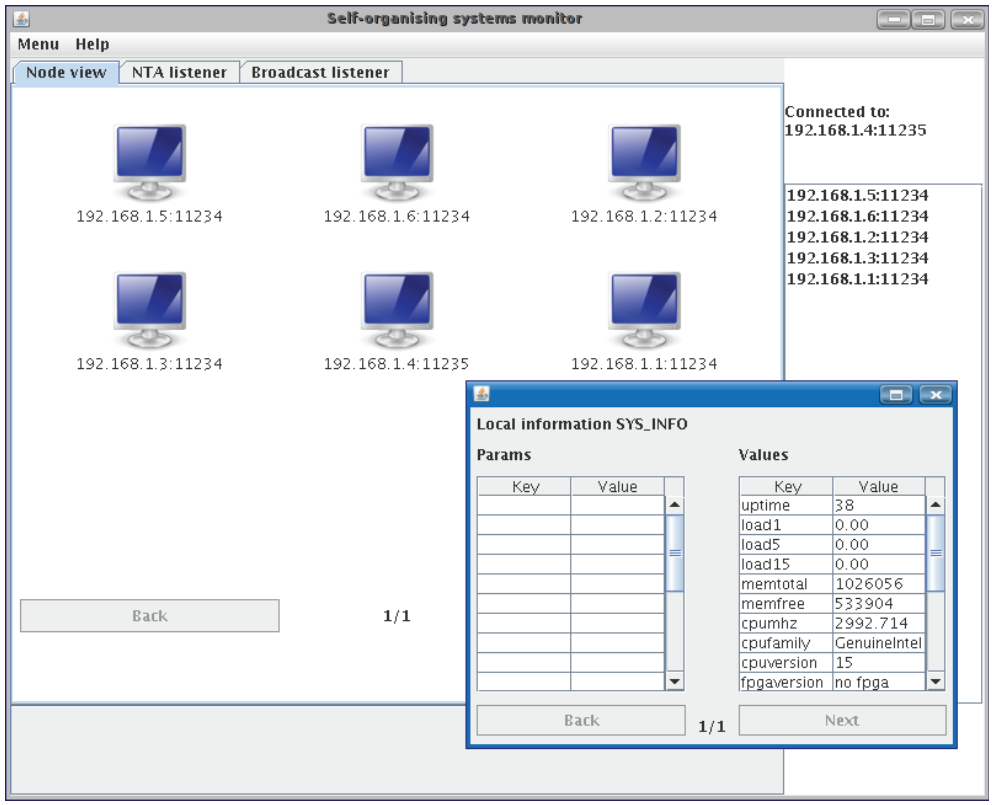


Figure 5: Communication between monitor daemons using the NTN interface and with applications using the NTA interface.

might as well contain wild-cards to address groups of nodes so that, for example, modifications that have to be applied to a set of devices can be launched by a single command.

The *Data_Volume* will be taken into account when an offer is generated for the request. It usually influences the decision whether it is more appropriate to solve a task in software or in hardware and if the data may be processed remotely or if the communication costs for that would be too high.

In order to be able to specify a *task* or a whole group of tasks, we suggest an hierarchical nomenclature which comprises every single service that can be rendered in the network under one root node.

As *Task* in the request structure is not limited to one atomic job, it may be a list of tasks. These lists may contain jobs that are to be processed sequentially or in parallel reaching from a single data-source to a single target, to complex data flows with multiple sources and targets.

Finally the restriction to *Max_Hops* ensures that inquiries are not simply flooded through the net but stay local working off jobs at a kind of in situ marketplace when sensible. This is of course only if the local neighbors provide appropriate solutions to tasks at a reasonable price. The composition of this "price" will be presented in Section 4.2. The idea behind that is obvious: since we are able to reconfigure devices to serve virtually any need, even small localized groups are highly adaptable and will be able to cope with most challenges with optimal efficiency. Thus data will in general be kept in a spatially narrow cloud, and communication is minimized. In the current and the next section we will explain how this is reconcilable with the claim of superregional cooperation.

To publish and find services to fill in a valid request basically three mechanisms can be deployed. First one central directory server knows which device offers which services and has to be prompted for every job to work off. If it fails, the whole network is paralyzed. New services have to be entered, causing additional traffic.

Second, searches for certain services are flooded through the whole network. This is very flexible, but rather inefficient.

In this work, we developed a third approach, a hybrid solution between the two previously mentioned ones. Here data are flooded only within the closest neighborhood. Only if the answers from them are insufficient, say because none of the next nodes wants to execute the requested task, the job is advertised again, this time with a higher number of maximum relays. Additionally devices keep a more or less extensive list of other remote hosts, that satisfy a certain request. In this manner not only local offers but also more distant ones will be taken into account. Distant ones possibly suit the current situation better. The maintenance of this list is closely related to the structure of offers replied to a request which will be covered in Section 4.2.

Generally, requests will be posed to the direct neighbors and to the issuer itself. A neighbor that finds the *Max_Hops* greater than one reduces that counter and sends the request to all its neighbours. Especially the answer from the node itself is interesting in this regard: with the possibility to

reconfigure, scenarios can be managed where communication cost is very high and nodes have to cut back on transporting lots of data through the net.

5.2. Offers. The response to a concrete query contains two elements: the cost-vector that the replying device is estimating for supplying the service and the local list of known providers that are also capable of satisfying this particular request:

$$Offer = \{CostVectors, List\_of\_known\_providers\}. \quad (2)$$

Costs mean figures given as multipliers of a base cost. For example, the transmission of one byte of data over a wireless bluetooth connection will be a lot more expensive than over a wired Ethernet link.

We identified three dimensions of costly actions: time consuming, energy consuming, and space consuming. Thus a device's purveyor calculates the cost vector for a task A according to

$$C_A = \begin{pmatrix} Z_K & E_K & P_K \end{pmatrix}^T \cdot W, \quad (3)$$

where Z_K denotes the cost for the local effort concerning time, E_K for energy, and P_K for required space. W contains the willingness of the device to spend part of the specific resource to locally execute task A as a diagonal matrix. A battery driven device might, for example, want to lower its willingness to accept very energy consuming jobs as it runs low on battery power. The final cost-vector C_A is passed to the issuer as part of the offer.

The list of additional service providers is being built up through logging of messages indicating the commission of a node for a particular task or through deliberate writing. On the one hand devices that relay an acceptation message (basically a request with a specially formulated task) to a device store the target device and task together with an expiration time. On the other hand devices can advertise their capabilities by giving out a request to every other participant of the net to amend its local list of providers. If it intends to stay in the community for a longer period, the expiration time may be set accordingly, attracting all sorts of requesters, locally and remote.

5.3. Negotiation Example. The flow of a negotiation bases on sending requests, retrieving responses, determining the most appropriate solution, and commissioning a purveyor (Figure 6). When evaluating the replies, the contained lists of alternative, maybe remote, providers may be taken into account and selected ones may be prompted for a bid. This enables the system to incorporate both: decentralized and distributed computing as well as central services like the storage of gathered and processed data.

When enough answers came back in or after an amount of time, the customer determines the optimal partner to commission. The weighted cost including communication is therefore calculated according to

$$A : CC_A = \vec{C}_A \cdot \vec{G}_A = \sum_i C_{A_i} \cdot G_{A_i}. \quad (4)$$

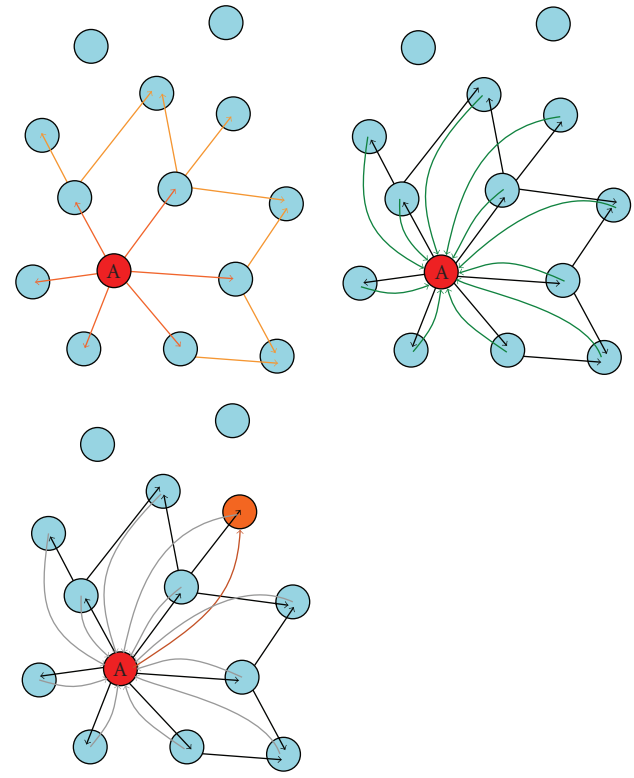


Figure 6: LMGS: issue a request (top left picture), retrieve answers (top right picture), and commission job (bottom left picture).

Manipulating the components of G_A allows the device to emphasize a rather fast, energy-efficient or space preserving execution of a task. Depending on the computation-power and -willingness the node might accept the earliest offer or may run a multigoal optimization on the plenty of data retrieved to find the best trade-off.

6. Conclusions

In this work we have shown an approach for a distributed resource management based on self-organization which utilizes dynamically reconfigurable hardware to optimize the resource utilization in constrained networks.

Our work also contributes to dependability in networked embedded systems as disappearing nodes are easily detected, so that tasks can be reassigned to nodes willing to utilize their spare resources. To implement our approach we have developed a monitor system that collects information from the surrounding neighbourhood, that can be used for device discovery, service discovery, and the detection of disappearing nodes due to failures. For example, this system can be deployed to prototype self-organizing routing algorithms, load balancing, resource management, and so forth. Furthermore we have described our targeted application, which is a distributed intelligent camera system which consists of nodes built upon reconfigurable hardware. Our approach for a self-organizing resource management is based upon local marketplaces (LMGSs) where work is distributed

according to the principles of supply and demand within the network to optimize the resource utilization, especially that of the reconfigurable hardware.

References

[1] S. Camazine, J.-L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau, *Self-Organization in Biological Systems*, Princeton University Press, Princeton, NJ, USA, 2003.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] J. W. Williams and N. Bergmann, "Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, T. P. Plaks, Ed., pp. 163–169, CSREA Press, Las Vegas, Nev, USA, June 2004.

[4] Xilinx, "Virtex-5 FPGA Configuration User Guide," June 2009, http://www.xilinx.com/support/documentation/user_guides/ug071.pdf.

[5] Xilinx, "Early Access Partial Reconfiguration User Guide," September 2008, http://kom.aau.dk/~ylm/rc/mm4-5/ug208_92.pdf.

[6] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "Design of homogeneous communication infrastructure for partially reconfigurable fpgas," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '07)*, Las Vegas, Nev, USA, June 2007.

[7] Xilinx, "Virtex-5 FPGA User Guide," June 2009, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 273791, 11 pages
doi:10.1155/2009/273791

Research Article

An Interface for a Decentralized 2D Reconfiguration on Xilinx Virtex-FPGAs for Organic Computing

Christian Schuck, Bastian Haetzer, and Jürgen Becker

Institut für Technik der Informationsverarbeitung (ITIV), Universität Karlsruhe (TH), Vincenz-Prießnitz-Straße 1, 76131 Karlsruhe, Germany

Correspondence should be addressed to Christian Schuck, schuck@itiv.uni-karlsruhe.de

Received 31 December 2008; Accepted 3 August 2009

Recommended by Peter Zipf

Partial and dynamic online reconfiguration of Field Programmable Gate Arrays (FPGAs) is a promising approach to design high adaptive systems with lower power consumption, higher task specific performance, and even build-in fault tolerance. Different techniques and tool flows have been successfully developed. One of them, the two-dimensional partial reconfiguration, based on the Readback-Modify-Writeback method implemented on Xilinx Virtex devices, makes them ideally suited to be used as a hardware platform in future organic computing systems, where a highly adaptive hardware is necessary. In turn, decentralisation, the key property of an organic computing system, is in contradiction with the central nature of the FPGAs configuration port. Therefore, this paper presents an approach that connects the single ICAP port to a network on chip (NoC) to provide access for all clients of the network. Through this a virtual decentralisation of the ICAP is achieved. Further true 2-dimensional partial reconfiguration is raised to a higher level of abstraction through a lightweight Readback-Modify-Writeback hardware module with different configuration and addressing modes. Results show that configuration data as well as reconfiguration times could be significantly reduced.

1. Introduction

Modern VLSI circuits are getting more and more complex as CMOS technology scales down into deep submicron (DSM) domains. In near future it will be possible to integrate hundreds of predesigned IP-cores, such as general-purpose processors, DSP, reconfigurable arrays, memory subsystems, and so forth, on a single chip. State-of-the-art FPGAs, such as the Xilinx Virtex series, are growing in complexity almost in the same order of magnitude, opening new promising possibilities for future systems, but at the same time the resulting complexity of such systems leads to big challenges for both the system designer, system programmer, as well as the user of the device. Reliability, especially MTBF, power consumption, controllability, and programmability are the main drawbacks.

Therefore, a new paradigm of system design is necessary to efficiently utilize the available processing power of future chip generations. To address this issue in [1] the *Digital on Demand Computing Organism* (DodOrg), which is derived from a biological organism, was proposed. Decentralisation of all system instances is the key feature to reach the desired goals of self-organisation, self-adoption and self-healing, in short the self-x features. Hence, the hardware architecture of the DodOrg system consists of many heterogeneous cells, so-called *Organic Processing Cells* (OPCs), as shown in Figure 1. They provide services for computation, memory, IO, and monitoring. A completely decentralized acting middleware layer [2] is able to dynamically assign upcoming tasks to the OPCs and group OPCs together to form virtual organs that are working closely together to solve a given task. All cells are connected by the "*artNoC*" [3] router network, which efficiently fulfils the special communication requirements of the organic system.

In general, all OPCs are made of the same blueprint, see Figure 2. On the one side they contain the cell specific functionality and on the other side they contain several common units, that are responsible for the organic behaviour of the architecture.

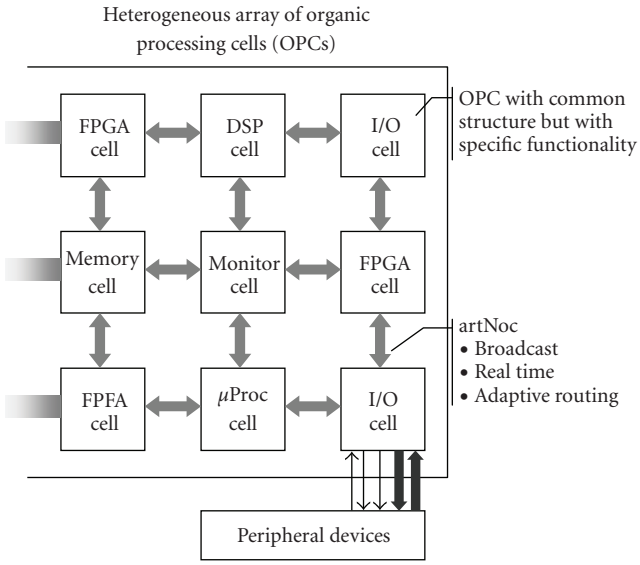


Figure 1: DodOrg organic hardware architecture.

Most OPCs of the array contain a highly adaptive datapath. Possible scenarios for the datapath range from μProc-cells, DSP-cells, coarse grained datapath units (DPUs), to completely flexible FPGA-like structures. This means, that some cells come prepacked with fixed functionality and others are more flexible. Therefore, it is possible to adapt the datapath during runtime to the needs of the current application tasks.

Within every OPC a *Configuration Management Unit* is responsible for the configuration of a single cell only and serves as a hardware protocol handler for the configuration exchange protocol with other OPCs.

During the distributed reconfiguration process, responsibilities might be shifted from one cell to another and configuration, as well as application data is exchanged. Each OPC forms its own local clock island, which is controlled by a *Clock and Power-Management Unit,* by applying dynamic frequency scaling (DFS). Therefore, it can control and adjust performance and power consumption of the cells datapath according to the actual computational demands of the application and the critical path accordingly. DFS has a high potential, as it decreases the dynamic power consumption by decreasing the switching activity of flip flops, gates in the fan-out of flip flops and the clock trees.

Xilinx Virtex-FPGAs offer the possibility of dynamic and partial online reconfiguration through an *internal reconfiguration access port* (ICAP). Therefore, they are ideally suited to serve as a test bed for the proposed organic cell based hardware architecture. However, three main requirements must be met the following:

(1) The OPC internal *Configuration Management Unit* needs to have access to the ICAP, in-order to realize a true decentralized reconfiguration of the cells datapath.

(2) For an effective floor plan, a fast two-dimensional reconfiguration must be possible.

(3) The FPGA area must be divided into clock islands that can be controlled by the *Clock and Power Management Unit*.

Therefore, in this paper we present the *artNoC-ICAP-Interface,* a lightweight hardware module that builds a bridge between the "*artNoC*" network and the ICAP. It is able to execute the *Readback-Modify-Writeback* (RMW) method in hardware, to fulfil the beforehand mentioned requirements. In the following sections the basics of Virtex-FPGAs and the RMW method are reviewed in brief and related work in the field of partial and dynamic reconfiguration and DFS on FPGAs is summarized. The rest of the paper is structured as follows: Section 4 introduces the basic architecture and the features of the implemented *artNoC-ICAP-Interface*. In Section 5 an overview of the DodOrg system architecture on Virtex-FPGAs is given, while Section 6 gives implementation results and performance and power figures. Finally, Section 7 concludes the work and gives an overview over future tasks.

2. Reconfiguration Basics

As this work is based on the Xilinx Virtex series reconfiguration architecture, the basics will be reviewed in brief.

2.1. Xilinx Reconfiguration Architecture. Xilinx FPGAs are configured by a so-called *bitstream*, which is written into the SRAM configuration memory. Basically, a Virtex-FPGA consists of two layers, the configuration memory layer and the hardware layer. The hardware layer contains the reconfigurable hardware structures, like configurable logic blocks (CLBs), RAM blocks (RAMBs), IO blocks (IOBs), digital clock managers DCMs and configurable wiring resources. The hardware units are aligned into a 2D grid.

The configuration memory layer is organized into columns, spanning from top to bottom of a device, see Figure 3. There are six different kinds of columns present, such as CLB-columns. Each column determines the function of the underlying hardware resources. Within such columns, several different hardware resources are included. For example, in every CLB-column, there are IOBs in the top and bottom of that column. For some hardware resources, like DCMs, no dedicated configuration columns exist. Instead, their configuration is included in one of the other columns.

Each column is further divided into subcolumns, called frames. One CLB-column for example is covered by 22 configuration frames. Within the Xilinx reconfiguration architecture, a frame is the smallest addressable unit [4].

Partial and dynamic online reconfiguration in this context means, that parts of the content of the configuration memory can be changed during runtime of the system, while the reminder of the configuration remains unchanged. However, as the smallest addressable unit is one frame, a single resource, for example, a CLB, cannot be addressed independently. Instead, several resources are accessed altogether. Furthermore, the division into large units, like CLBs or IOBs is too coarse grained—there are smaller units like

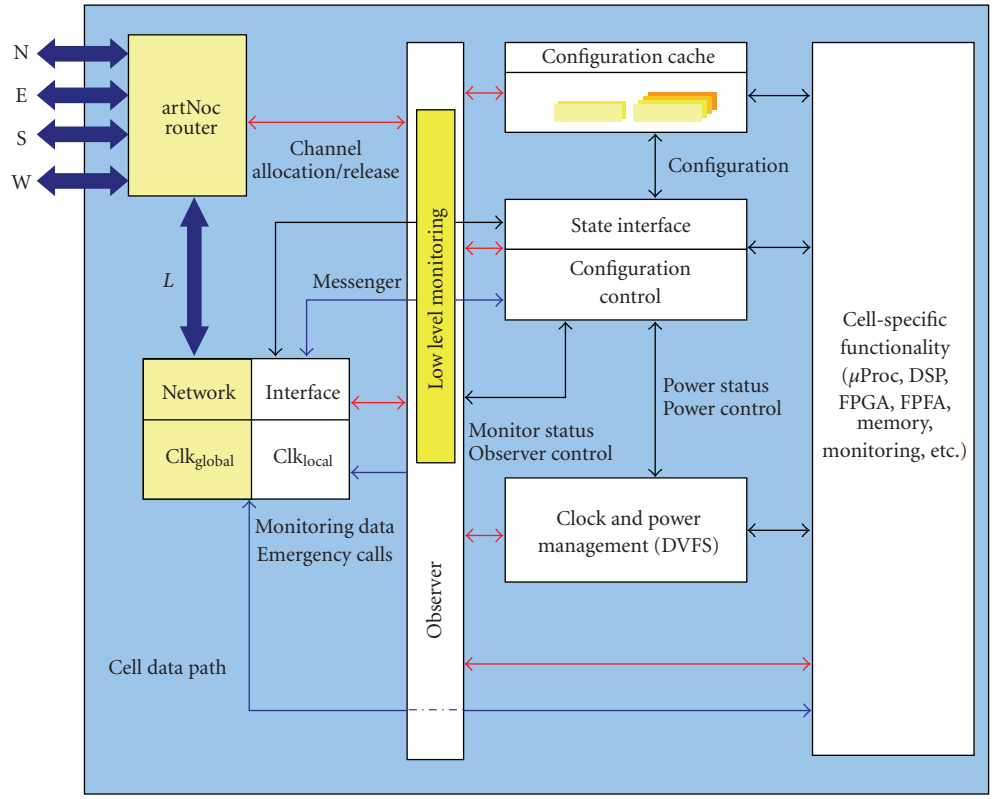


Figure 2: OPC block diagram view.

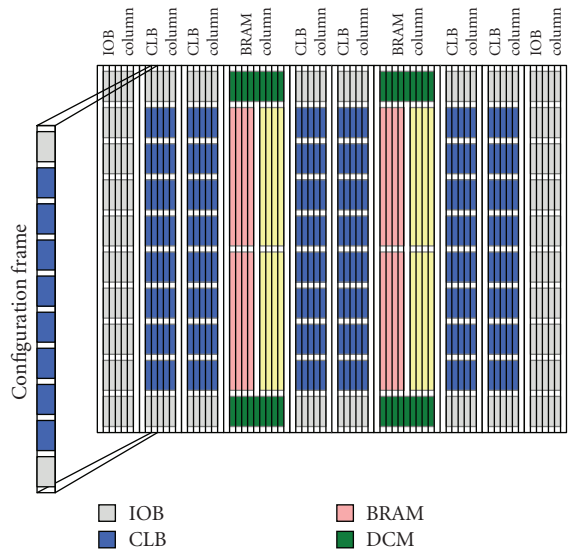


Figure 3: Xilinx virtex configuration architecture [4].

clock distribution buffer or look-up tables, which are not covered by the addressing scheme. A direct addressing of these units may be useful.

As discussed, the hardware units are arranged in a 2D grid, but the configuration architecture gives no direct access to the configuration space. However, for the proposed OC-application, a true 2D access is needed. Therefore, a special RMW [5] method has to be applied, as explained in the subsequent section.

2.2. Readback Modify Writeback Method. Besides writing configuration data to the configuration memory, Virtex-FPGAs allow that configuration can be read back. This feature, combined with the fact that a glitch less switching of configuration logic occurs, once a part of the configuration is overwritten with an equal content, can be used to alter every single bit of the configuration memory instead of whole frames. The procedure is applied as follows. First, the content of the frames to be modified is read back from the configuration memory and stored in a proper secondary memory. In a second step, the new configuration is merged into the read back frames. This entails, that parts of the frames stay the same, while other parts are overwritten by the new configuration. Finally, the content of the secondary memory, with the modified configuration, is written back to the configuration memory. The method significantly improves the flexibility and the area usage of the resulting system level design. However, the following two main drawbacks have to be taken:

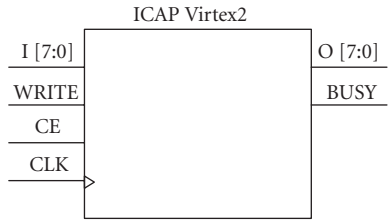


Figure 4: ICAP interface [4].

(1) The reconfiguration times for the same amount of chip area (e.g., measured in CLBs/second) are substantially increased compared to the classical one-dimensional approach [6].

(2) The interfacing with the configuration logic is considerably more complicated, as the merging of new configuration data into the read back data has to be controlled, as well as the single steps including generation of partial bitstream headers and addresses have to be accomplished [5].

Therefore, these major drawbacks are addressed by the presented approach in Section 4. Beforehand, the ICAP interface and related work will be introduced.

2.3. ICAP Interface. Besides other configuration interfaces Xilinx Virtex-FPGAs feature an internal configuration access port. It allows the on chip logic to have read, as well as write access to the *configuration control logic* (CCL) and to alter the current configuration during runtime. Combined with the presented methods, it is the key to a self-configuring and self-organizing hardware system.

Figure 4 shows its interface. Two separate 8-Bit data ports one for read (O) and one for write (I) are available. The WRITE signal indicates if data has to be written to or read from the ICAP. It has to be set by the controller. Once the ICAP is enabled with the CE line, it expects or rather provides valid data at every rising edge of the clock signal, given that the BUSY signal is held low. Once the ICAP interfaces raises the BUSY signal, it indicates, that it is no longer able to send/receive data at the data ports (O/I).

The CCL features a set of registers, which are used for communication. Each write access to the configuration memory is pipelined through an internal frame buffer. So, after a configuration frame is written to the CCL, an additional, so-called dummy frame, has to be written to flush the internal buffer to memory. The same is valid for read access.

The communication with the CCL consists of 5 basics steps.

(1) Sending of synchronisation sequence.

(2) Sending of operation commands to set the CCL registers.

(3) Sending/Receiving actual configuration data

(4) Sending/Reading of one dummy frame to flush CCL internal buffer.

(5) Sending desynchronisation sequence.

2.4. artNoC Local Port Interface. The local interface of the used artNoC [3] basically consists of two ports with mirrored signals: an input port for injecting data packets into the network, and an output port for receiving data packets from the network. The concept of virtual channels (VCs) allows a time interleaved simultaneous transmission of different data packets over the physical data input and output lines. The number of simultaneous virtual channels can be tailored during design time to fit the needs of the connected client. Further, for each VC a real-time feedback signal exists, that is routed in the opposite direction of the established datapath, from receiver to the sender, during VC reservation. Once the channel from sender to receiver is established, guaranties for throughput and latency can be given, and an in-order delivery of data can be assured. As we will see, this is important for the design of the proposed *artNoC-ICAP-Interface* module.

3. Related Work

3.1. Partial and Dynamic Online Reconfiguration. Most of the systems using partial and dynamic online reconfiguration make use of the Xilinx OPB-ICAP peripheral, which can be either connected to the "*Microblaze*" soft core processor or via a PLB to OPB-bridge to the Power PC cores available at the Virtex-II-Pro devices. Various systems have been implemented making use of 1-dimensional slot-based reconfiguration [6, 7], while others utilize the RMW method for a 2 dimensional placement [5]. Common to all these systems is that reconfiguration is controlled by a single processor. The reported configuration speeds are far away from the maximal possible speed provided by the ICAP. To overcome the speed limitations, in [8] an approach is presented that connects the ICAP to the PowerPC PLB bus with direct memory access (DMA) to a connected BRAM, lowering the processor load and speeds up writing of configuration data close to its maximum value. However, no support for the RMW method is provided, nor is it possible to access the ICAP in a virtual fashion. In [9, 10] the ICAP is connected to a NoC to allow a distributed access, but no support for the RMW method is provided, which is a key feature of the presented *artNoC-ICAP-Interface*. Further, the reported reconfiguration speeds suggest, that both interfaces do not operate at the optimal operation point.

3.2. Dynamic Frequency Scaling. Recently several works has been published dealing with power management and especially clock management on FPGAs. All authors agree, that there is a high potential for using DFS method in both ASIC and FPGA designs [11, 12].

In [13] the authors show, that even because of FPGA process variations and because of changing environmental conditions (hot, normal, cold temperature), dynamically clocking designs can lead to a speed improvement of up to 86% compared to using a fixed, statically, during design time estimated, clock. The authors use an external programmable

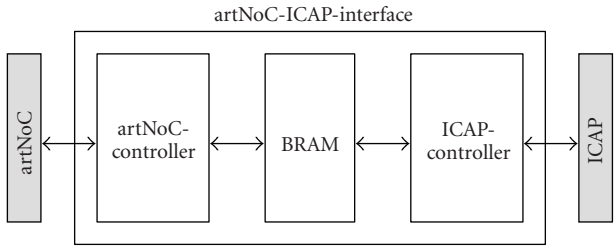


Figure 5: artNoC-ICAP-interface.

clock generator that is controlled by a host PC. However, in-order to enable the system to self-adapt its clock frequency, on chip solutions are required.

In [11] the authors proposed an online solution for clock gating. They use a feedback multiplexer with control logic in front of the registers. So it is possible to keep the register value and to prevent the combinatorial logic behind the register to toggle. But simultaneously, they highlight that clock gating on FPGAs could have a much higher power saving efficiency, if it would be possible to completely gate the FPGA clock tree. To overcome this drawback, in [14] the authors provide an architectural block, that is able to perform DFS. However, this approach leads to low speed designs and clock skew problems, as it is necessary to insert user logic into the clock network.

We show that on Xilinx Virtex-II no additional user logic is necessary to efficiently and reliably perform a fine grained self-adaptive DFS. This is possible because digital clock managers (DCMs) are available, that can be reconfigured during runtime with the presented interface. All advantages of the high speed clock distribution network could be maintained.

4. Architecture Details

To overcome the mentioned drawbacks and to realize a virtual decentralisation of the local ICAP interface an approach as shown in Figure 5 was taken.

The *artNoC-ICAP-Interface*, acts as a bridge between the artNoC router network and the ICAP interface of the Virtex-FPGAs.

The interface is able to receive configuration commands from all OPCs connected to the network. Further, from OPC view (client view) it provides different levels of abstraction. Clients can operate as close as with the natural ICAP commands, as well as sending their configuration on a higher level of abstraction, for example, on CLB basis. The actual interfacing in this case is taken over by *artNoC-ICAP-Interface*. This simplifies the handling of configuration significantly and the amount of configuration data to be sent to the interface is reduced (see Section 5.2) Furthermore, through a local handling of configuration data within the interface, important speedups could be achieved.

4.1. Basic Concepts. As mentioned before, the Xilinx Virtex configuration hardware works on a frame basis. When employing the RMW method every reconfiguration operation consists of a frame read and a frame write. To achieve this basic function, the interface consists of a BlockRAM, that can hold the content of one frame, and a unit that controls the ICAP, called *ICAP-Controller*. This unit implements all functionality to read a frame from ICAP, store it in BRAM and write it back to ICAP. To process the requests from network clients and to modify the configuration frame content, a second unit is implemented, the *artNoC-Controller*. To give this unit access to memory a dual ported BRAM is used. The advantage of this architecture is that the two controllers are decoupled by memory, making it possibly to operate both at different clock rates. Because the ICAP appears to be the bottleneck in the system the controller can work with the highest possible clock frequency (see Section 6.2). Further, a simultaneous loading of BRAM with both new configuration data and read back data (merging) can be performed resulting in remarkable speedups. The block diagram of the architecture is shown in Figure 5. The basic functions of the three units are summarized in the following listing.

artNoC-Controller

(i) processes configuration requests from clients,
(ii) controls the overall interface operation,
(iii) generation of frame address(es),
(iv) controls operation of ICAP-Controller,
(v) merging data into BRAM.

Dual Ported BRAM

(i) holds data of one configuration frame,
(ii) decouples artNoC-Controller and ICAP-Controller.

ICAP-Controller

(i) controls ICAP-Interface signals,
(ii) sends command sequences to ICAP,
(iii) reading/writing data from or to BRAM.

4.2. Design Considerations. FPGA resources, especially BRAMs, are strongly limited when thinking of a complete system with many OPCs that implement a certain functionality. As the interface implements the RMW method, at least data for one configuration frame has to be held in memory. There is a possibility to achieve an advantage by spending more memory than for one frame. As mentioned in Section 2.3, each read or write operation requires an additional dummy frame to be written. When consecutive frames have to be processed, the number of dummy frames can be reduced by reading more than one frame into BRAM, modify the frames and write them back to ICAP with only one dummy frame at the end of the read and write operation. Through this, the number of dummy frames can be reduced from n to [n/m] (with n # frames

to modify, m # memory for one frame), which leads to a higher reconfiguration performance. Unfortunately this is only possible for consecutive frames, with the cost of more memory and advanced control logic.

The used technique that configuration data from clients is directly merged in the BRAM saves resources, because no requests have to be stored and the control logic is simple (see Section 6.1). The disadvantage is that clients have to wait till complete configuration operations have finished, before another request can be accepted.

In this implementation at one time only requests from one client can be processed. The reason for this restriction is motivated as follows. Every configuration operation must do a complete RMW sequence as the ICAP cannot be interrupted. Only requests from other clients which would operate at the same frame have a chance to be processed in parallel. When expecting that requests from different clients are statistically independent, the possibility that requests belong to the same configuration frame is very small, due to the big FPGA area. Even if this occurs, special care has to be taken, as the configuration regions may overlap, which leads to a more complex control logic. In scenarios, where some clients are correlated with each other, an enhanced approach may be used. Otherwise, it is not worth to process more than one client request simultaneously.

The implementation presented here uses the simplest and least resource consuming approach.

4.3. Operation Modes. The *artNoC-ICAP-Interface* features five different operation modes, which allow the client cells to access the interface according to their current needs. One OPC, for instance, wants to configure its whole cell internal datapath, spanning several CLB blocks, the other cell just wants to change a few bits within a LUT, to adjust the coefficients of an implemented filter task. The different operation modes both insure that a minimum of configuration data has to be sent over the network, as well as the actions for controlling the reconfiguration, that have to be taken by the client cells, are kept low. Hence, the amount of additional memory needed to store the new partial configuration data as well as the complexity of the configuration managers within each OPC can be reduced.

The following listing describes the different modes and explains their purpose.

(1) Transparent-Operation-Mode. The interface works as a virtual extension of the physical ICAP hardware. Clients can send native ICAP commands [4], which are tunnelled through to the ICAP. The clients have full access to all ICAP features. This mode is intended to serve as a debugging feature or in case later system enhancements need this low level access.

(2) Read/Modify Frame-Operation-Mode. The client can read back or modify configuration data of one frame. This mode is suited, if just parts of modules have to be reconfigured, for example, changing a LUT.

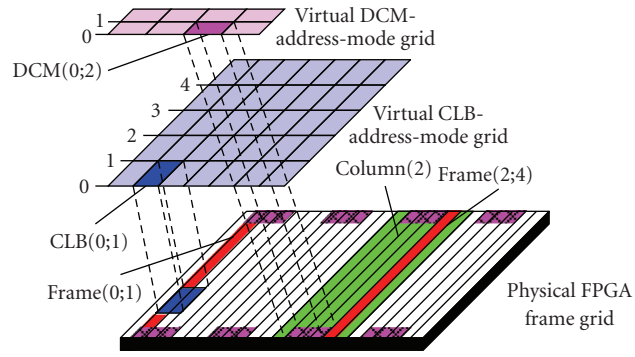


Figure 6: Virtual address grids.

Different address modes are available to select resources within a frame (see Section 4.4). Every desired frame can be addressed and several units within the frame can be processed.

(3) Read/Modify Module-Operation-Mode. The configuration data of whole modules can be read or modified. The module width can span several columns and the height can range from one unit to the maximum count a frame can hold (see Section 4.4). All frames within these columns are processed (e.g., CLB column: 22 frames/column).

4.4. Address Modes. As shown in Figure 6 the physical FPGA hardware resources are covered by a frame grid. Thereby, every single Virtex configuration frame contains the configuration information for different FPGA resources, such as CLBs, IOBs or DCMs. To take the burden from a client who, for example, wants to operate on CLBs, to calculate the addresses of the frames that have to be processed and the positions within these frames, where the configuration lies, the *artNoC-ICAP-Interface* features different addressing modes.

In this example the client can operate in CLB address mode and just needs to transmit the CLB coordinates of the desired CLB (cf. Figure 6, e.g., CLB $(0;1)$) where the new configuration has to be inserted. The interface automatically performs the mapping down to the physical FPGA frame grid. As with the RMW method every single bit can be changed independently, the access to the configuration memory can be more fine granular. This means, that new addressing modes can easily be added to give individual access to smaller hardware resources like clock distribution buffer or look-up tables, which is inherently not given. So, for each hardware resource an individual address grid can be introduced.

From a clients view with this method reconfiguration can be done on a much higher level of abstraction without knowing the exact underlying configuration architecture.

Moreover, with this approach complete hardware systems can be easily ported to different device families such as Virtex 4- or Virtex 5-FPGAs by just adjusting the address mapping parameters within the artNoC-ICAP-Interface package. At the moment CLB-, BRAM- and DCM-addressing modes are supported.

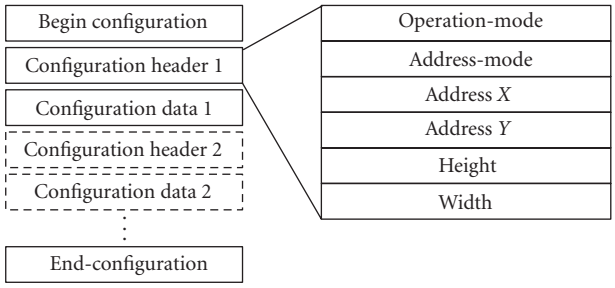


Figure 7: Configuration protocol.

4.5. Implemented Protocol. Each configuration request is started with the *begin-configuration* command. After that, different configuration operations are available, till an *end-configuration* command is received. Each configuration operation is preceded by a header followed by the data necessary for that operation (e.g., native ICAP command (transparent mode), module configuration data (module mode)). Figure 7 shows the protocol. The header main fields are *operation-mode* and *address-mode*. The *operation-mode* selects one of five different operations such as modify-module see Section 4.3. In every *operation-mode*, except transparent mode, all presented address modes are available, see Section 4.4. The *address-mode* determines on which FPGA resources the configuration works. The address fields are interpreted in related coordinates, so every resource is addressed with its own 2D grid. As discussed in the operational modes section, clients can configure whole modules (e.g., in CLB address mode 22 frames) or single frames. When a module is configured in module mode, one header has to be sent, followed with the complete information of that module. This especially includes data that has to be merged into all frames spanning the module, even if the data remains unchanged within several frames, so-called dummy data. When only few frames have to be changed it can be more efficient not to configure in module mode, but to address the frames in single frame mode. For every operation a header is needed followed by a data section. Dependent on the module size one of the two possibilities is more efficient. The client can decide which alternative is the appropriate for his needs. The following equation evaluates the number of frames n till the module operation is more efficient:

$$n = \frac{\text{header size} + \text{module height} \times 22}{\text{header size} + \text{module height}}. \quad (1)$$

If the number of frames to be modified is bigger than n, module mode is more efficient in terms of configuration data to be sent over the network, otherwise frame mode should be chosen.

4.6. RMW Configuration Procedure. The complete RMW configuration procedure is shown in Figures 8(a) and 8(b). First a network client establishes a real-time-channel with the *artNoC-Controller*. This ensures exclusive access for this

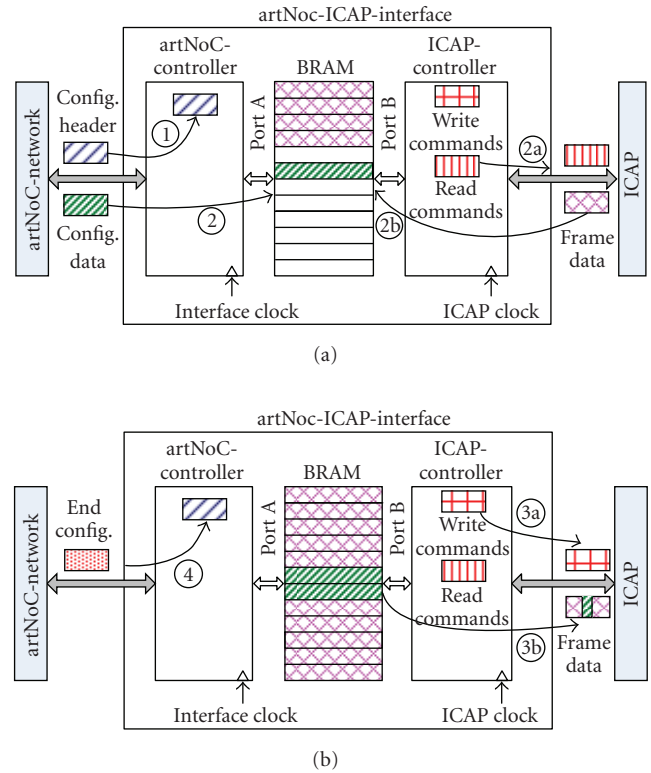


Figure 8: artNoC-ICAP-interface configuration procedure.

client and all other request from different clients will be rejected. Thereafter the *begin-configuration* command and the header are sent over the network (1). The *artNoC-Controller* processes the header to generate the proper frame-address(es) based on the chosen address mode, and triggers the *ICAP-Controller* to read back the frame from configuration memory. The readback consists of two steps: first a read command-sequence is written to the ICAP (2a) and the frame data from ICAP is continuously stored in BRAM (2b) on a byte basis (PORT B). Simultaneously, the configuration data, that has to be merged into the frame, is received from the network. It is directly written to the proper position in BRAM (2) over the second port (PORT A). At the same time PORT A has exclusive write access to the portion of BRAM it writes to, which results in the fact that the data written back from ICAP targeting this portion is discarded and the new configuration data is merged into the read back frame. When the whole frame is in memory, the write back procedure starts with the write command-sequence (3a) and the modified frame data (3b) are both send to ICAP.

This also includes automatic generation of the dummy frame after the frame has been written. Finally, the client sends an *end-configuration* command to the *artNoC-Controller* (4), which releases the established real-time channel and gives other clients the possibility to access the interface.

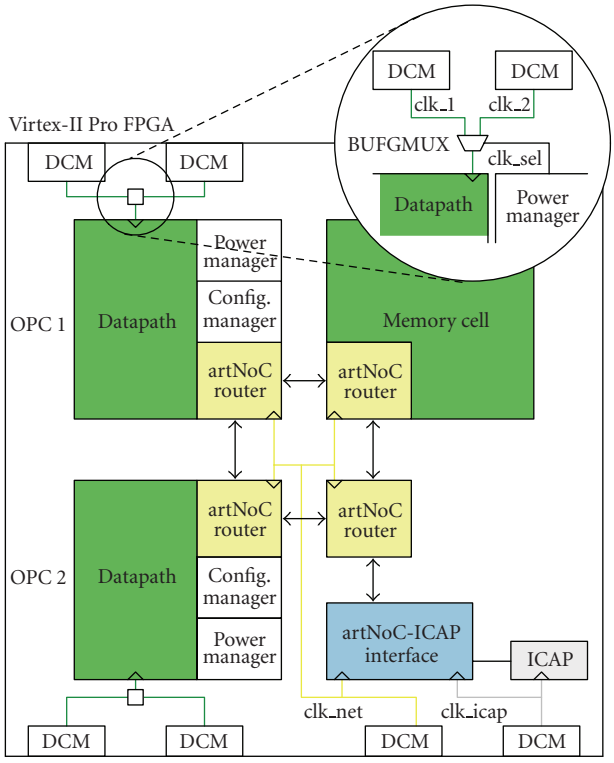


Figure 9: DodOrg FPGA floor plan/clock architecture.

5. Organic System Architecture

5.1. Overall Floorplan. As mentioned in the introduction, we propose to implement the OPC-based organic computing organisms on a Virtex-II Pro FPGA. The floor plan of the complete system is shown in Figure 9. Because of the two-dimensional reconfiguration capabilities, it is possible that every OPC occupies a self contained rectangular area. Depending on the device size several OPCs can be implemented onto a single FPGA. Every OPC is composed of a *configuration management unit* (CMU), a *power management unit* (PMU), a runtime reconfigurable datapath area (DPA) and the artNoC router. The artNoC router allows all OPC units to communicate with units of the other OPCs and the *artNoC-ICAP-Interface*. The *artNoC-ICAP-Interface* is located in the lower right corner of the FPGA and is connected to the artNoC as well as described in Section 4. All modules, except the datapath area, belong to the static design area and are not changed during runtime.

The clock net of the reconfigurable DP module of every OPC is connected to a BUFGMUX, which is driven by a pair of DCMs. The DP module clock is decoupled form the artNoC clock by using a dual ported, dual clock FIFO buffer. By using the features of the *artNoC-ICAP-Interface* every PMU can dynamically adapt the DCM output clock frequency, as described in Section 5.3.

5.2. Datapath Reconfiguration. The reconfiguration data of each DP module is stored in dedicated memory cells. Memory cells use the available on chip BRAM and provide an interface to be accessible over artNoC network. Every DP module is identified by a unique ID. A specific characteristic of the stored reconfiguration data is that it contains no absolute placement information and only the actual configuration data is stored. That means, that the represented DP module can be loaded into every OPCs reconfigurable datapath area. This is in contrast to the EAPR-flow, where the complete frames of all columns have to be stored. For example, the used "*Microblaze*" DP module requires 136 CLBs ($\rightarrow$ reconfigurable datapath area 4*34 CLBs) which results in a storage requirement of:

(i) 4Col*22Frames/Col*34Rows*10Byte/(Row*Frame) = *29920 Bytes*,
Compared to the partial bitstream size of:

(ii) 4Col*22Frames/Col*824Byte/Col= *72512 Bytes*
In case of the EAPR flow (for a XC2VP30 device). This is a reduction of *58.7%*.

Derived from the biological model the basic idea is the distributed organisation of the whole system. That means, in each OPC the CMU is responsible for the DPA of this cell. To carry out a reconfiguration process the cells CMU generates the configuration header and sends it to the *artNoC-ICAP-Interface*. It contains the placement information as explained in Section 4.5. Based on the ID, it then fetches the actual DP module configuration data from a memory cell and redirects it to the *artNoC-ICAP-Interface*. Finally, after the new DP module is configured the CMU triggers a reset and the module is ready to operate.

In general, each DP module has a specific maximum clock frequency according to its critical path. As a consequence, it is necessary to adjust the clock frequency after reconfiguration in-order to operate at maximum speed, or rather to operate in save conditions. This adjustment can be done by the PMU of the cell.

5.3. Power Management. Like DP reconfiguration, power management is performed on OPC basis. Therefore, the digital clock managers (DCMs) of Xilinx Virtex-FPGAs are used. Besides others, frequency synthesis is an important feature of the DCMs. Two main different programmable outputs are available. CLKDV provides an output frequency that is a fraction ($\div 1.5$, $\div 2$, $\div 2.5 \cdots \div 7$, $\div 7.5$, $\div 8$, $\div 9 \cdots \div 16$) of the input frequency CLKIN.

CLKFX is able to produce an output frequency that is synthesised by combination of a specified integer multiplier $M \in \{1 \cdots 32\}$ and a specified integer divisor $D \in \{1 \cdots 32\}$ by calculation CLKFX = M $\div$ D*CLKIN. By using the *artNoC-ICAP-Interface* both multiplier and divisor can be reconfigured during runtime and hence the clock frequency at the CLKFX output can be adjusted. However, reconfiguring a DCM involves a minimum delay of 60 μs (see Section 6.2) to change the clock frequency, if the *artNoC-ICAP-Interface* is not blocked by an active request.

This means, that the method is appropriate for reaching long term or intermediate term power management goals, that is, a new datapath is configured and the clock frequency is adapted to its critical path and then stays constant until a

new datapath is required. But if a frequent and immediate switching is necessary, for example, when data arrives in burst and between burst the OPC wants to toggle between shut off ($f_{DP} = 0$ Hz) and maximal performance ($f_{DP} = f_{max}$) the method needs to be extended.

In this case, a setup consisting of two DCMs and a BUFGMUX, as shown in Figure 9 can be chosen. The select input of the BUFGMUX is connected to the PMU of the OPC. Therefore, it is able to toggle between two frequencies immediately without any delay.

6. Result

6.1. Synthesis Results. The presented *artNoC-ICAP-Interface* has been synthesised and implemented targeting a Virtex2VP30 device by using the Xilinx ISE 9.1 toolchain. Table 1 shows the required resources. As we see only a small portion of FPGA resources are used. When considering complex multi core systems with many heterogeneous cores, like the DodOrg system, these figures can even get more neglected on bigger devices, but in turn the performance of the 2D reconfiguration can be significantly improved and the complexity of client's configuration logic, which is multiplied by the number of clients, can be reduced.

6.2. Reconfiguration Performance. One goal was to maximise reconfiguration performance of the RMW method. As a reference the time needed to perform a complete RMW cycle for one configuration frame on a Virtex2VP30 (824 Bytes) with 10 Byte configuration data was measured. As a test setup a hardware configuration client was connected to the artNoC, as shown in Figure 10. This client operates in *Modify-Frame-Mode* with *CLB-Address-Mode*. It repeatedly sends 10 Byte configuration data to toggle the function of a LUT from an AND- to an OR- gate. The correct function could be verified by applying two push buttons as input and a LED as an output to the gate.

Xilinx Chip Scope was connected to the signals representing the current internal state of the *artNoC-Controller*. So the number of clock cycles, starting with the receiving of the first byte of the config header until the RMW cycle was completed, could be measured. Thereby, the *artNoC-Controller,* as well as the "*artNoC*" where operated at a constant clock frequency of 50 MHz, whereas the clock frequency of the ICAP and the *ICAP-Controller* were altered from 50 to 110 MHz, as shown in Figure 11.

The measured data is divided into readback and write-back portion and is expressed in terms of clock cycles of the *artNoC-Controller*. As we can see, the maximum throughput is reached at an ICAP speed of 105 MHz with 2048 clock cycles which is equal to a configuration time of 40 μs/frame.

For example, the "*Microblaze*" DP module, spanning 4 clb-columns with an height of 34 clb-rows, can be modified within 3,5 ms. The width of the module determines the reconfiguration time, as it also determines the number of frames to be modified. The same "*Microblaze*" with a layout of 8 clb-columns and 17 clb-rows requires twice the time. This has to be considered during the design phase.

Table 1: Resource utilization.

Resource	Number	Percentage
Slices	387	2%
Slice FlipFlops	183	0%
4 input LUTs	709	2%
BRAMs	1	0%
MULT18x18s	2	1%

Table 2: Component power consumption.

	Passive power (mW)	Active power (mW)
static_offset	—	11
DCM	—	37
CMU	<1	<1
artNoC-ICAP-IF	<1	9
ICAP	69	76

One would expect that the performance can be further improved with increased ICAP clock frequency, but the measurements show, that the ICAP itself is limited in speed. With increased clock frequency the number of ICAP busy cycles also increases, due to ICAP internal processing. Especially in read-mode, the ICAP-busy cycles causes the *ICAP-Controller* to stall.

The speed of the *artNoC-Controller* itself is not critical, as in 1217 clock cycles, which are need for read-back, the new configuration of a whole frame can be easily merged into the read back frame (assuming 8 Bit router links).

6.3. Power Consumption. In the preceding section results for reconfiguration times and tradeoffs have been presented. This section evaluates the potential of power savings and performance enhancements in the context of module based partial online reconfiguration. Especially, the overhead in terms of area and power consumption introduced by the approach (CMU, *artNoC-ICAP-Interface*, DCM) is taken into account.

We calculated the power consumption by measuring the voltage drop over an external shunt resistor (0.4 Ohm) on the FPGA core voltage (FGPA_VINT). As a test system again the Xilinx XUP board, with a Virtex-II Pro (XC2VP30) device, was used. For all measurements the board source clock of 100 MHz was used as an input clock to the design.

To isolate the portions of power consumption, as shown in Table 2, several distinct designs have been synthesised. For DCM power measurement, an array of toggle flip-flops at 100 MHz with and without a DCM in the clock tree has been recorded and the difference of both values has been taken. For extracting ICAP power consumption, a system consisting of CMU, *artNoC-ICAP-Interface* and ICAP instance and a second identical system, but without ICAP instance, have been implemented. After activation, the CMU sends bursts of two complete alternating configuration frames, targeting the same frame in configuration memory. The ratio of toggling bits between the two frames is 80% and is considered to be

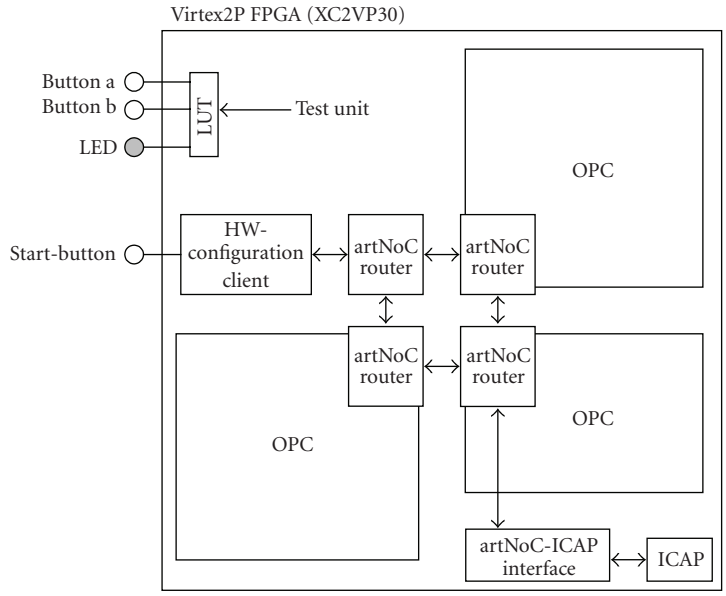


Figure 10: Measurement test setup.

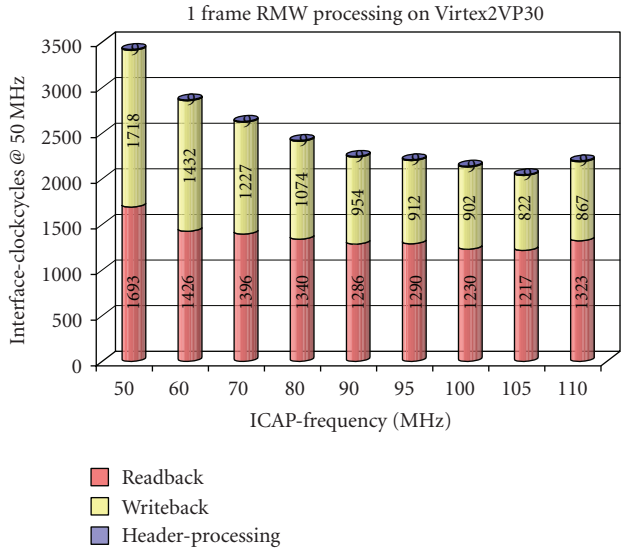


Figure 11: RMW performance.

representative for a partial reconfiguration. Therefore, before CMU activation, the "passive" power and after activation the "active" power could be measured. Again, the difference in power consumption of the two systems was taken to extract ICAP portion. The other components were measured with the same methodology. Therefore, for example, all components necessary to implement the approach presented in Section 5.3, with two DCMs + BUFGMUX consume 196 mW when active, that is, 180 mW when passive. But it has to be considered, that *artNoC-ICAP-Interface,* as well as ICAP is also used for partial 2D reconfiguration.

7. Summary and Future Work

In this paper we have presented the *artNoC-ICAP-Interface,* which is a small and lightweight hardware module to support a fast RMW method for a true 2 dimensional online reconfiguration of Xilinx Virtex-FPGAs. The reconfiguration time could be reduced to 40 μs per frame, which at the same time determines the physical limit of the ICAP. Different operation modes, which can be combined with various addressing modes, optimize the amount of traffic that has to be sent over the network for reconfiguration and raise the abstraction level of the reconfiguration interface to a higher level. The innovative concept can be easily ported to different device families, also featuring an internal configuration access port. With slight changes, it should also be possible, to modify the network interface in-order to suit the needs of different NoC architectures, so they can also benefit from the achieved virtual decentralisation of the ICAP. Similarly, not only distributed reconfiguration systems can take effort from the presented concepts. Because of the FIFO based interface, the *artNoC-ICAP-Interface* can also be easily connected to single configuration controller, like the "*Microblaze*" processor (e.g., over the FSL-bus). Hence, it can take advantage of the addressing modes and performance improvements in 2D reconfiguration.

In this work we determined the optimal operation point to reach a throughput for RMW, which is just limited by the physical ICAP instance.

Further we showed how the *artNoC-ICAP-Interface* can be used to reconfigure the Digital Clock Managers during runtime, to perform a Dynamic Frequency Scaling. A concept for clock net partitioning, using the example of the organic system architecture, was shown. Both long term as well as short term power management goals can be achieved.

All advantages of the high speed clock distribution network could be maintained.

Finally, we provide measurement results for power consumption of all relevant components involved.

To the best of our knowledge the *artNoC-ICAP-Interface* is the first approach to support the RMW method with all properties described above. Future work is targeted towards the self-adaptation and self-healing capabilities of the presented organic hardware architecture by employing suitable monitoring or observer techniques.

References

[1] J. Becker, et al., "Digital on-demand computing organism for real-time systems," in *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS '06)*, W. Karl, et al., Ed., Karlsruhe, Germany, February 2006.

[2] U. Brinkschulte, M. Pacher, and A. von Renteln, *An Artificial Hormone System for Self-Organizing Real-Time Task Allocation in Organic Middleware*, Springer, Berlin, Germany, 2007.

[3] C. Schuck, S. Lamparth, and J. Becker, "artNoC—a novel multi-functional router architecture for organic computing," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 371–376, Amsterdam, Netherlands, August 2007.

[4] "Xilinx Virtex-II Platform FPGA User Guide," UG002(V1.3).

[5] M. Hübner, C. Schuck, and J. Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, p. 8, Rhodes Island, Greece, April 2006.

[6] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms," in *Proceedings of the IEEE International Conference on Field Programmable Technology*, pp. 37–42, December 2005.

[7] M. Ullmann, M. Hübner, B. Grimm, and J. Becker, "An FPGA run-time system for dynamical on-demand reconfiguration," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '04)*, vol. 18, pp. 1841–1848, Santa Fe, NM, USA, April 2004.

[8] C. Claus, F. H. Müller, J. Zeppenfeld, and W. Stechele, "A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS '07)*, pp. 1–7, Long Beach, Calif, USA, March 2007.

[9] R. Koch, T. Pionteck, C. Albrecht, and E. Maehle, "An adaptive system-on-chip for network applications," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, p. 8, April 2006.

[10] L. Möller, I. Grehs, E. Carvalho, et al., "A NoC-based infrastructure to enable dynamic self reconfigurable systems," in *Proceedings of the 3rd International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC '07)*, Montpellier, France, June 2007.

[11] Y. Zhang, J. Roivainen, and A. Mämmelä, "Clock-gating in FPGAs: a novel and comparative evaluation," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD '06)*, pp. 584–588, August-September 2006.

[12] I. Brynjolfson and Z. Zilic, "FPGA clock management for low power," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGAs '00)*, Monterey, Calif, USA, February 2000.

[13] J. A. Bower, W. Luk, O. Mencer, M. J. Flynn, and M. Morf, "Dynamic clock-frequencies for FPGAs," *Microprocessors and Microsystems*, vol. 30, no. 6, pp. 388–397, 2006.

[14] I. Brynjolfson and Z. Zilic, "Dynamic clock management for low power applications in FPGAs," in *Proceedings of the Custom Integrated Circuits Conference (CICC '00)*, pp. 139–142, Orlando, Fla, USA, May 2000.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 390167, 7 pages
doi:10.1155/2009/390167

Research Article

Reducing Reconfiguration Overheads in Heterogeneous Multicore RSoCs with Predictive Configuration Management

Stéphane Chevobbe and Stéphane Guyetant

CEA, LIST, PC94, 91191 Gif-sur-Yvette, France

Correspondence should be addressed to Stéphane Chevobbe, stephane.chevobbe@cea.fr

Received 24 December 2008; Accepted 9 August 2009

Recommended by Gilles Sassatelli

A predictive dynamic reconfiguration management service is described here, targeting a new generation of multicore SoC that embed multiple heterogeneous reconfigurable cores. The main goal of the service is to hide the reconfiguration overheads, thus permitting more dynamicity for reconfiguring. We describe the implementation of the reconfiguration service managing three heterogeneous cores; functional results are presented on generated multithreaded applications.

1. Introduction

Classical embedded application domains, such as wireless networking, video encoding/decoding, or still pictures enhancement, typically need high computing performance and a high level of flexibility. They can take a great benefit of implementations on dynamically reconfigurable hardware: actually, it allows accelerating most applications thanks to explicit parallelization and optimized hardwired implementation. The flexibility is also achieved by modifying at runtime the datapath or the operators themselves. The ratio between the configuration time and the actual execution time is a key figure, because it defines the programming model and the dimensioning of the configuration control logic of such architectures: on one hand, dynamic reconfiguration at application level needs to adapt the operators to new functions from time to time, for example, to launch a recognition engine when a moving object is detected in front of a camera; on the other hand, dynamic reconfiguration at the instruction level is able to change the context every few cycles, for example, to switch a condition inside nested loops.

Through the high potential of dynamically reconfigurable architectures, only a few commercial successes exist, which is mainly due to the following shortcomings.

(i) Reconfiguring a core implies to load some configuration data during the application execution, which leads to a nonnegligible time overhead.

(ii) Compared to static reconfiguration during which the execution is halted, the dynamic reconfiguration needs more control logic and is eventually more complex than static reconfiguration.

(iii) The lack of programming model is critical for most application designers: taking into account the dynamic reconfiguration complexities the design flow and using dedicated tools are necessary.

This article will first review in Section 2 the services that are commonly found in order to manage the reconfigurations of RSoCs. Section 3 will present the reconfigurable platform on which the work is based. Then in Section 4 we will explain the principles that were selected for this platform's reconfiguration. Validation and experiments are presented in Section 5, and finally we conclude on the status of the study.

2. Related work

Several examples can be found in literature concerning reconfiguration management services. Figure 1 depicts the

most common services that can be sorted under three main classes.

(i) The temporal optimization services have the goal to minimize the reconfiguration time by reducing or masking the loading of the configuration bitstreams.

(ii) The spatial optimization services seek to maximize the number of present configurations on the hardware resource, for example, by defragmenting hardware tasks on a reconfigurable array; thus, more tasks can be executed within the same configuration.

(iii) Finally, the functional optimization services allow more flexibility for the reconfigurable architectures: inserting breakpoints to facilitate preemption and allowing migration of tasks on several resources are examples of such services.

The dynamic behavior of a service is based on exchange of information with the scheduler that can provide the status of the running application. Thus the service can adapt its behavior by tuning its parameters, leading to better reconfiguration performance.

Most of the time, the reconfigurable systems use several of these services together. Figure 1 presents the way how all possible run-time services are related: for example, configuration prefetching and caching are complementary to reduce the reconfiguration time. Another example is the spatial optimization of reconfigurable resources that involves a bitstream processing toolchain in order to partition, place, and route the operators.

Anyway, it would be unfeasible yet to implement all these services in a run-time reconfiguration manager, because of the very high complexity of some of these services. Even if some algorithms can be downsized with strong simplifications, such as placement considering only tiled architectures, we will now focus on the four most common services, namely compression, prefetching, caching, and allocation.

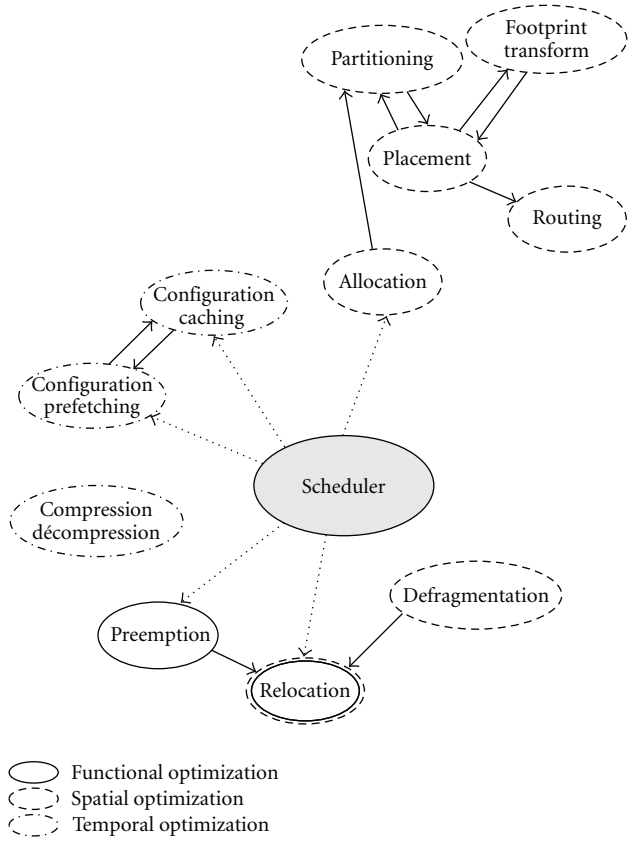


Figure 1: The existing reconfiguration services can be classified into three classes, according to the kind of optimizations they provide.

2.1. Bitstream Compression. The first way to reduce the reconfiguration overhead is to decompress the configuration bitstreams during run-time: this is useful when the configuration bottleneck is not the loader of the actual reconfigurable core but is located instead in the interconnect hierarchy; compression also reduces the occupation of configuration memories and eventually caches. This is a particular relevance for FPGAs [1, 2], whose bitstream size is typically over hundreds of kilobytes. Such an approach is profitable, but in the case of heterogeneous RSoCs, dedicated decompression engines have to be used to conform to each bitstream format.

2.2. Configuration Prefetch. The configuration prefetch is quite similar to the instruction prefetch found in general purpose processors: configuration data are meant to be present in an internal configuration memory before being actually called, so that all the transfer time (or at least a part of it) is hidden. Figure 2 explains the principle with three reconfigurable cores (IPs): *Ln* slots correspond to the loading of a task, and *Ex_n* blocks correspond to its execution. When no prefetch is performed (see Figure 2(a)), the reconfiguration implies a time overhead; on the contrary, Figure 2(b) shows that the prefetch service reduces the prefetch time or ideally makes it disappear.

An interesting example of prefetching was developed in [3] that targets heterogeneous multiprocessor platforms integrated inside an FPGA. The application is split into tasks, themselves divided into subtasks; the service must schedule the loading of subtasks in order to mask the configuration latencies, using static and dynamic information. The generation of static data is done when designing the application: a set of critical tasks is manually extracted, which correspond to the subtasks that are to be loaded in priority. The dynamic data are provided by the scheduler: the critical subtasks are updated according to the current execution status. In this example, the off-line generation of some prefetch choices permits to lower the computational load of the prefetch service, still keeping a dynamic behavior. The work in [4] presents another example that concerns techniques useful for FPGAs that can support dynamic partial reconfiguration, at least on rows; it shows that prefetching is always better that simple configuration caching. The raw-based reconfiguration permits easier relocation and defragmentation in order to enhance prefetching.

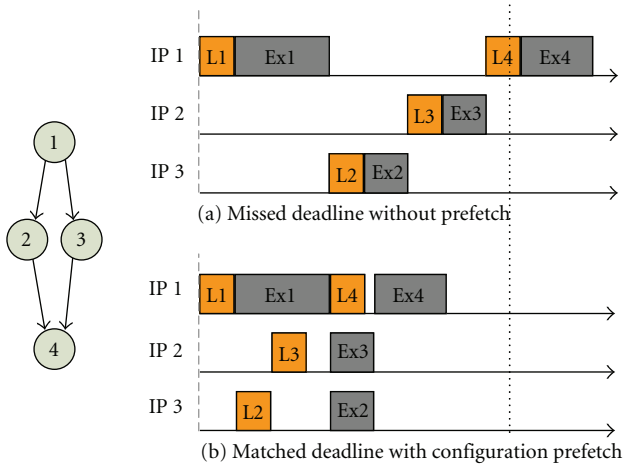


Figure 2: Principle of configuration prefetch for a graph of four tasks that execute on three cores denoted IP1 to IP3.

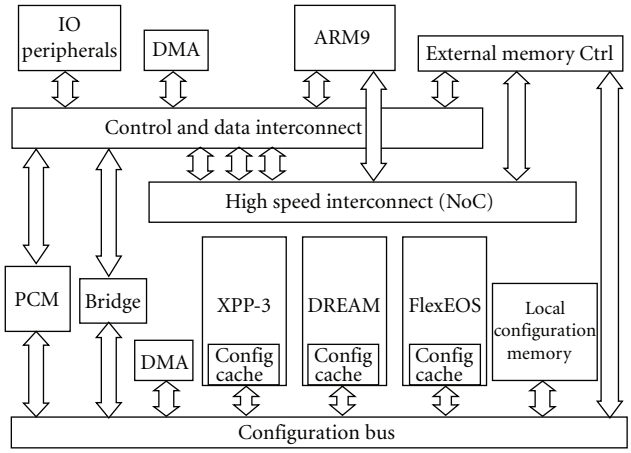


Figure 3: The considered platform is architected around three heterogeneous reconfigurable cores interconnected by an NoC.

2.3. Configuration Caching. Again, the configuration caching is similar to the instruction caching found in processors: the goal is still to have the reused control information present in on chip memories when needed, and thus, the cost of loading bitstreams on a reconfigurable core is lower. The approach differs from the processors because of the difference of size (one or so words for an instruction, several dozens to hundreds of words for configurations), and the size of partial bitstreams can be very irregular: in the processor approach, most reused instructions are kept, but for a reconfigurable approach, the bigger size of bitstreams and thus the loading time make the cache misses very costly; therefore it might be more interesting to favor a large bitstream instead of a frequently used small bitstream.

Several algorithms for configuration caching are developed in [5], targeting various models of FPGAs: in particular, a multicontext FPGA and partially reconfigurable FPGAs are studied. The algorithms have been designed in C++, and the experiments show that for small configurations, the multicontext model leads to a reconfiguration overhead 20 to 40% smaller than for the partially reconfigurable model.

2.4. Allocation. This service is basically in charge of allocating the hardware resources to the software tasks, by maintaining the list of all available resources. Anyway, more complex services can also have the capacity to relocate running tasks in order to defragment the mapped operations on the platform. In [6], the authors enhance the allocation service with run-time partitioning, placement, and routing, judged as the four mandatory services for an OS dedicated for reconfigurable computing. As the algorithms associated with such services can be quite complex, a compromise is done between their efficiency and their own computing time.

3. Case Study Platform

3.1. Platform Presentation. The platform developed in the European project MORPHEUS (Multipurpose dynamically Reconfigurable Platform for intensive Heterogeneous processing) was used as a case study [7]: the architecture, represented on Figure 3, is build around three reconfigurable cores with heterogeneous computational grain: XPP3 [8], a coarse grain matrix partially reconfigurable enhanced with several VLIW cores, DREAM [9], a middle grain architecture that expands the functional units of an RISC processor with a multicontext middle grain reconfigurable matrix, and finally an FLEXEOS [10] embedded FPGA. Anyway, the service presented in this article is meant to be used with any kind of reconfigurable cores, in any number and hierarchical pattern. At the system level, the execution control is here performed by an ARM926 processor and an NoC is responsible of high bandwidth data transfers between the various computational resources.

One of the MORPHEUS project's goals is to promote the use of the reconfigurable technology, thanks to the help of a complete toolset to port any application on the platform [11]. Demonstrations will be performed in the field of image processing, video surveillance, network processors, and wireless stations [12].

3.2. Configuration Subsystem. A dedicated bus and memories are used to transfer and store the bitstreams. All the bitstreams that are used during an application execution are packed inside a configuration library, which is located in external memory, either a flash memory or a faster volatile memory initialized at boot time. The main processor executing the PCM service or the dedicated PCM component is in charge of programming DMA transfers from this external memory to the internal hierarchy of configuration memories: the first level is the local configuration memory that is shared between all the cores. Each core can be associated or not to a dedicated configuration cache, which is a dual-port memory accessible concurrently by the core's loader and from the configuration bus. Finally, the last level of configuration memories consists of the reconfigurable core's context itself.

The predictive reconfiguration management service can be implemented fully in software as one of the OS modules

running on the ARM core, or with a dedicated hardware component. In the silicon implementation of the MORPHEUS platform, the PCM component can be considered as a coprocessor that will not only leverage the processor with low-level bitstreams transfers and configuration memory hierarchy management when the reconfiguration directives are issued but will also offer the high-level prefetch prediction services, with an almost immediate answer to the OS requests at the cost of a reasonable area overhead.

In order to simplify the memory transfers and addresses calculation, all the configuration bitstreams are split into chunks of two kilobytes; obviously, this leads to a small occupation overhead due to the incomplete last block, but the eFPGA bitstreams and other libraries of small bitstreams used in the project size between 20 and 60 Kbytes.

4. Implementation of a Reconfiguration Management Service

4.1. Configuration Mechanism. The dynamic reconfiguration mechanism is based on the Molen [13] programming paradigm at the thread level: pragmas in the applicative source code explicit the functions that will be accelerated on the reconfigurable cores. The Molen compilation analyses these pragmas and inserts the following commands:

(i) SET: starts the prefetch of the configuration;

(ii) EXEC: starts execution of the accelerated function when ready;

(iii) BREAK: the execution is stopped, but the operation will be reused;

(iv) RELEASE: the function will not be executed any more, and associated hardware resource can be freed.

In the original Molen approach, the SET is statically scheduled *as soon as possible*, to ensure that loading the configuration can be entirely done before the execution is requested. In the case of the studied platform, the operating system is responsible for managing the multithread execution; the SET is then statically scheduled *as last as possible* in order to maximize the availability of the hardware resources; they can be dynamically rescheduled by the OS and sent to the PCM service.

In addition to the configuration control commands, the PCM service interacts with the scheduler thanks to a set of requests issued by the scheduler. Typically, these requests provide the scheduler with status information of the memory contents, reflecting the prefetched configurations. Some complex requests are computed by the PCM: for example, the *TimeToExecute* request returns the remaining time needed before a configuration is ready to be executed, computed by the PCM; this time is bounded by zero if the bitstream was already prefetched to the maximal time to access the full bitstream for the external memory.

As described in Figure 4, the PCM services receives dynamic information from the OS, that not only mainly the configuration commands, but also the thread priorities that are used by the prefetch service. The static information,

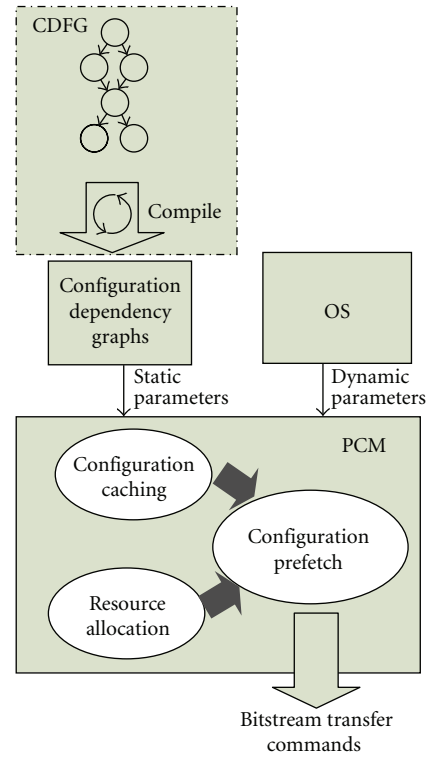


Figure 4: Principle of the reconfiguration management.

such as execution probability (extracted from application profiling and that annotate conditional branches) and implementation priority (given by the application designer to differentiate several implementations of the same software function, e.g., on heterogeneous cores), is embedded in the graph representation; so that is easily retrieved by the PCM. The allocation, caching, and prefetch services are then ultimately translated into commands to transfer the bitstreams between the levels of the configuration memory hierarchy.

4.2. Loader Interface. The configuration service is meant to deal with every reconfigurable core, and not only those selected for the MORPHEUS implementation; this explains why it does not provide specialized decompression service nor is it intrusive with the internal configuration mechanisms: the goal is to provide a unified access to the configuration interfaces at the system level. All existing reconfigurable cores have their own protocol; anyway, they can be classified in two main categories. The first includes the loaders that are passive (memory-mapped or decoding frames). Active loaders, that are autonomous in retrieving their configuration data, belong to the second category.

For the MORPHEUS platform, the PCM service is able to prefetch configurations internally to the passive loaders but restricts the prefetch for active loaders at the cache memory level.

4.3. Predictive Reconfiguration. In an autonomous way from the scheduler, the PCM does not only select the next tasks that are to be queued for scheduling but also does a broader

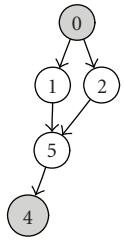


Figure 5: Subgraph example.

search inside all configuration call graphs: let us consider the example presented in Figure 5; suppose that only the tasks 0 and 4 have an FlexEOS implementation. At the end of task 0, this core is released, but task 4 is too deep in the graph to be selected for immediate prefetch. On the contrary, the PCM will walk the graph looking for the next tasks that have an existing implementation for the just freed core.

When all tasks that can be prefetched have been selected by the first stage of the PCM, they are affected a dynamic priority calculated from the static and dynamic parameters: a polynomial function is implemented with coefficients that can be later fine-tuned to different application behaviors. These dynamic priorities are sorted together so that the most relevant prefetch actions can be sorted inside a FIFO. Then, following this order, the bitstream transfers can start until a new command is issued by the scheduler. Arbitrary thresholds select for full or partial bitstream loadings. Obviously, if the execution time between two consecutive schedules is too short, the prefetch cannot take place, but at least the service does not create additional overhead. Contrarywise, an arbitrarily long execution time leads to a perfect prefetch that hides all reconfiguration latencies, as we will verify in the next section.

5. Experiments and Results

5.1. Test Cases. The behaviour of the Predictive Configuration Management service has been validated by automation of application graphs execution and metrics monitoring. First, configuration call graphs are generated: they do not represent real applications but are created randomly and include up to 30 tasks; an example is showed in Figure 6. For each simulation run, 16 such graphs represent many threads of an application. Then a bitstream library is generated, with one to three different implementations for each software task. Obviously, this library does not contain real configuration data, but the structure is relevant according to the generated application. Each bitstream has a random size between 2 KB and 64 KB (representing coarse grain or partial fine grain configurations). The scheduling of function calls between all the threads is also randomly created.

5.2. Simulation Environment. Several simulation environments have been developed to validate the functionality of the service: RTL models were used to simulate the PCM hardware component, and the software implementation was tested at two levels of integration: it was first ported as

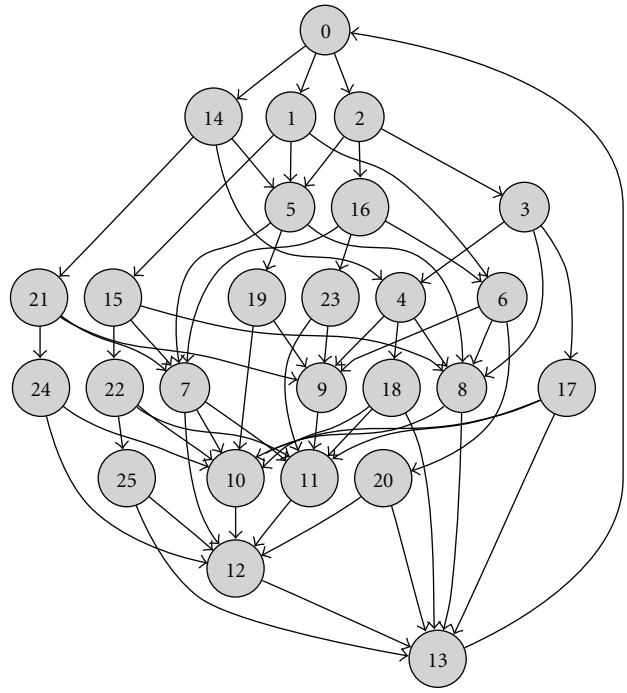


Figure 6: Example of a generated configuration call graph.

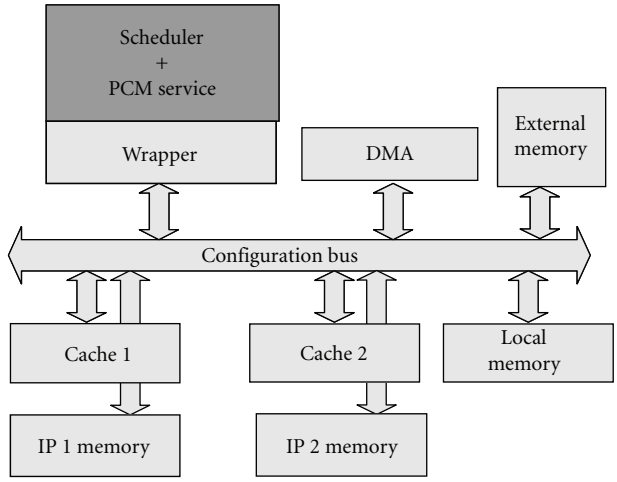


Figure 7: Simulation environment.

an eCos driver executed on the eCos synthetic target (a Linux process emulating eCos), cosimulating with a SystemC model of the platform. It has the advantage of the real behavior of the OS, but due to limitations in the number of tasks with the cosimulation with the synthetic target, a fully SystemC simulation was developed: the PCM service and the scheduler are encapsulated into TLM models coupled to the aforementioned SystemC platform (see Figure 7).

5.3. Results. In order to assess that the configuration overhead was masked, we measure during the execution of a generated application for each SET command issued the number of blocks that were prefetched for the considered

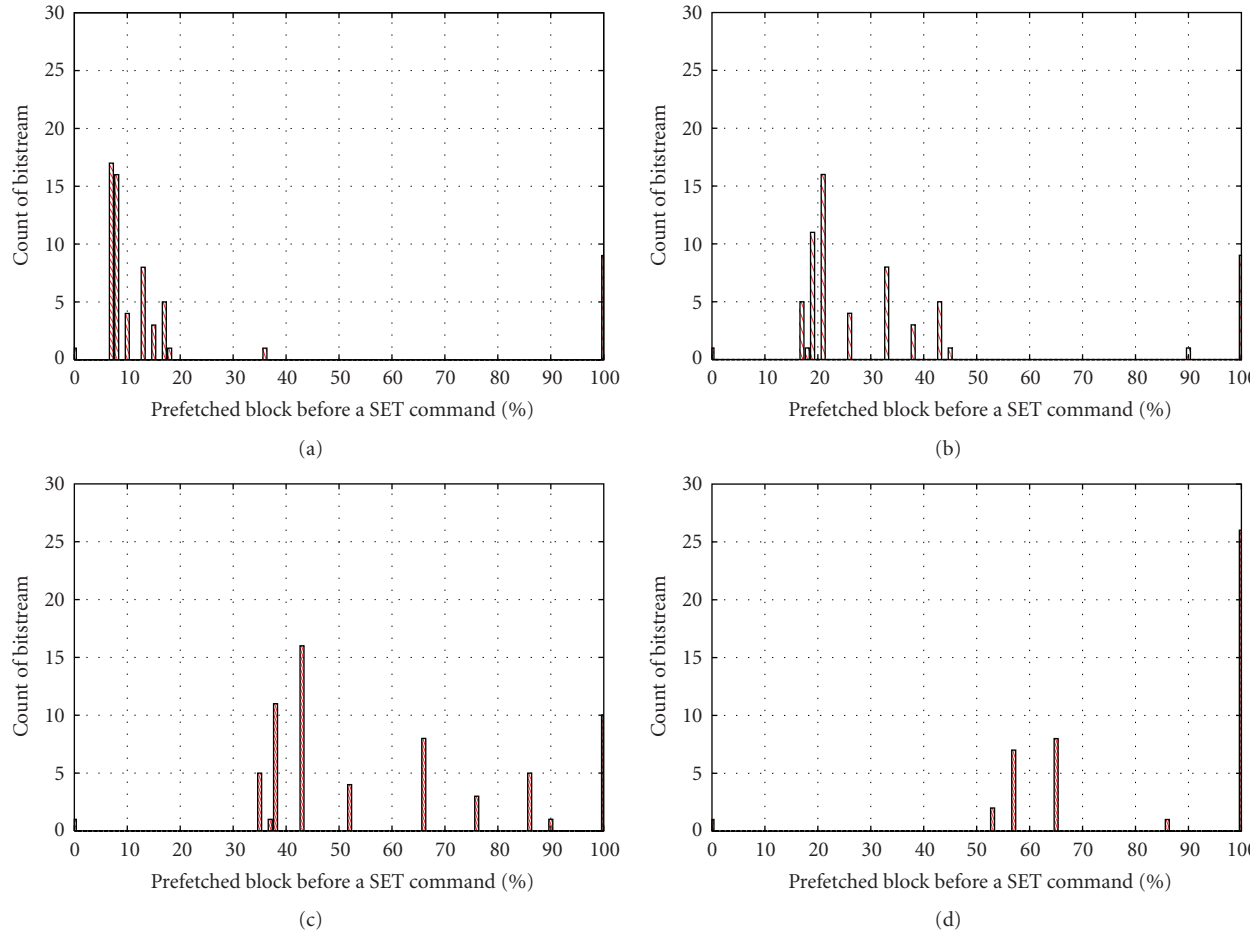


Figure 8: Histograms showing the prefetch results; from top (a) to bottom (d), the execution time is 1, 2.5, 5, and 7.5 milliseconds.

bitstream. This number is normalized by the size of the complete bitstream and is presented in percentage as a histogram. For example, Figure 8(a) shows that during a particular execution, 17 bitstreams were prefetched at 7% and 9 bitstreams were prefetched at 100%. All histograms of Figure 8 are generated at the end of the execution of always the same application. The average execution time of hardware operation varies from 2 to 15 times prefetch prediction time, on Figure 8(a) to 8(d), respectively. The prediction time corresponding to the software implementation of the PCM is estimated to 500 microseconds on an ARM9 processor.

When all the tasks are prefetched at the SET command (i.e., to say, there is only one bar at 100%), the reconfiguration overhead is totally masked. On the contrary, when the prefetch could not take place, the overhead is around 100 microseconds per SET, for tasks that last less than 1 millisecond.

What emerges from the histograms of Figure 8 is that when the execution time grows, the percentage of prefetched blocks increases, and thus the overhead decreases respectively. Indeed, hardware operation for which average execution time is around 1 millisecond (Figure 8(a)) the PCM prefetches 10% to 20% of the bitstreams of the application. Then for execution time around 2.5 milliseconds (Figure 8(b)) the PCM prefetches 20% to 40% of the bitstreams. All experiments over 9 milliseconds show only one bar at 100%, except the very first call which is missed. If the shape of the curve is dependant from each application, the behavior is similar from one application to another.

The PCM performance is very dependant of the ratio between the time it takes to perform the prefetch prediction and the actual execution time of the task. Typical applications studied have reconfiguration needs from 1 milliseconds down to dozens of microseconds. The software version of the PCM, that can run around 500 microseconds for each prediction update, performs poorly with sub-millisecond reconfiguration requirement (see Figure 8(a)). Thus, using the hardware component, that was measured to be almost 20 times faster, will permit to execute applications with a more dynamic behavior. Also architectures that include more than three cores could be targeted, because the prefetch prediction will last longer with more cores.

By comparison with the Molen paradigm, we differ by the execution of concurrent threads. Molen deals with monothreaded applications with a reasonably regular behavior; it performs excellent prefetch with an "as soon as possible" policy, computed at compile-time after profiling.

Our position is that multithreaded applications with non-predictable behavior need dynamic scheduling and flexible allocation mechanisms.

6. Conclusion

A predictive configuration management service dedicated to multicore heterogeneous reconfigurable SoCs was described in this article; this service is used to reduce the reconfiguration overhead and also to provide a unified view of the heterogeneous resources at the system level.

Hiding the reconfiguration overhead is achieved by computing at run-time the heterogeneous allocation, the prefetch of configurations bitstreams by blocks in a hierarchy of three levels of memories, and caching the reused tasks in these same memories.

The service is based on some static information in order to reduce the computational load at run-time; anyway, a hardware implementation is proposed to leverage the main control processor.

The functional validation of the service is presented as well as dimensioning and prefetch policies evaluation method.

In the future, the availability of the silicon platform will allow to test the reconfiguration service with true applications in real conditions and to measure accurately the benefits of using such service. Also the availability of a complete toolset [11] will permit to deal with real application cases, that are simpler than the generated use cases used in this article.

Acknowledgment

This research was partially supported by the European Commission under project MORPHEUS no. FP6-2004-IST-027342. See http://www.morpheus-ist.org/.

References

[1] S. H. Z. Li, "Configuration compression for Virtex FPGAs," in *Proceedings of 9th Annual Symposium FPGAs for Custom Computing Machines*, pp. 147–159, 2001.

[2] A. Dandalis and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 12, pp. 1394–1398, 2005.

[3] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 106–111, 2005.

[4] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *Proceedings of the 10th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '02)*, pp. 187–195, 2002.

[5] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, 2000.

[6] G. Wigley and D. Kearney, "The development of an operating system for reconfigurable computing," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 249–250, 2001.

[7] MORPHEUS, http://www.morpheus-ist.org/.

[8] PACT, The XPP III white paper.

[9] A. Lodi, M. Toma, and F. Campi, "A pipelined configurable gate array for embedded processors," in *Proceedings of the 11th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '03)*, pp. 21–30, 2003.

[10] M2000, http://www.m2000.com/.

[11] G. Edelin, P. Bonnot, W. Gouja, et al., "A programming toolset enabling exploitation of reconfiguration for increased flexibility in future system-on-chips," in *Proceedings of the Design, Automation and Test in Europe (DATE '07)*, 2007.

[12] F. Thoma, M. Kühnle, P. Bonnot, et al., "Morpheus: heterogeneous reconfigurable computing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 409–414, 2007.

[13] E. Moscu Panainte, K. L. M. Bertels, and S. Vassiliadis, "The Molen compiler for reconfigurable processors," in *Proceedings of the ACM Transactions in Embedded Computing Systems (TECS '07)*, 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 259837, 13 pages
doi:10.1155/2009/259837

Research Article

FPGA Interconnect Topologies Exploration

Zied Marrakchi, Hayder Mrabet, Umer Farooq, and Habib Mehrez

LIP6, Université Pierre et Marie Curie, 4, Place Jussieu, 75252 Paris, France

Correspondence should be addressed to Zied Marrakchi, zied.marrakchi@lip6.fr

Received 1 December 2008; Accepted 20 July 2009

Recommended by J. Manuel Moreno

This paper presents an improved interconnect network for Tree-based FPGA architecture that unifies two unidirectional programmable networks. New tools are developed to place and route the largest benchmark circuits, where different optimization techniques are used to get an optimized architecture. The effect of variation in LUT and cluster size on the area, performance, and power of the Tree-based architecture is analyzed. Experimental results show that an architecture with LUT size 4 and arity size 4 is the most efficient in terms of area and static power dissipation, whereas the architectures with higher LUT and cluster size are efficient in terms of performance. We also show that unifying a Mesh with this Tree topology leads to an architecture which has good layout scalability and better interconnect efficiency compared to VPR-style Mesh.

1. Introduction

The work presented in this paper can be divided into two parts. In the first part, we present an improved Tree-based FPGA architecture. In the second part of the paper, the architecture presented in the first part is used for the improvement of connection blocks and intracluster interconnect topologies in a cluster-based Mesh FPGA architecture.

The motivation behind the work presented in the first part is to reduce the domination of the interconnect area in field programmable arrays (FPGAs). In FPGAs interconnect area takes up to 90% of the total area while the remaining 10% is used by the logic part of the architecture. Domination by interconnect greatly affects the delay and area efficiency of the architecture. In [1] the authors have shown that the best way to improve circuit density is to balance logic blocks and interconnect utilization. In this paper we present an improved Tree-based FPGA (MFPGA) architecture where interconnect and logic utilizations are controlled using different architectural parameters, and it is shown that by reducing the logic occupancy of the architecture, we can increase the interconnect utilization of the architecture resulting in over all area reduction. Also, in this part we investigate the effect of LUT and cluster size on the area, performance, and power dissipation of a Tree-based FPGA. Many studies in the past several years were carried out to see the effect of LUT and cluster size on the density and performance of FPGA architecture [2–5]. But all the work previously done in this context focuses on the Mesh-based FPGA architecture, and no work has been done for Tree-based architectures up to now.

The motivation behind the second part of the paper is the optimization of connection blocks and intracluster interconnect topologies in a cluster-based Mesh FPGA architecture. There are different ways to connect signals to the LUT input muxes. In Xilinx Virtex architectures [6], the routing tracks are connected directly to the input muxes. In the VPR [7] and the Altera Stratix [8] architectures, the routing tracks are connected to the input muxes via an intermediate level of muxes called connection block. VPR-style interconnect has a sparsely populated connection block and a fully populated intracluster crossbar. The fully populated intracluster crossbar is simple but takes no advantage of the logical equivalence of LUT inputs and causes a significant inefficiency. Lemieux and Lewis [9] improved the basic VPR-style interconnect in two ways. They proposed an approach to generate highly routable optimized connection block. Furthermore, they showed that the intracluster full crossbar can be depopulated to achieve significant area reduction without performance degradation. A practical example is Stratix, which depopulates this crossbar by 50% [8]. All

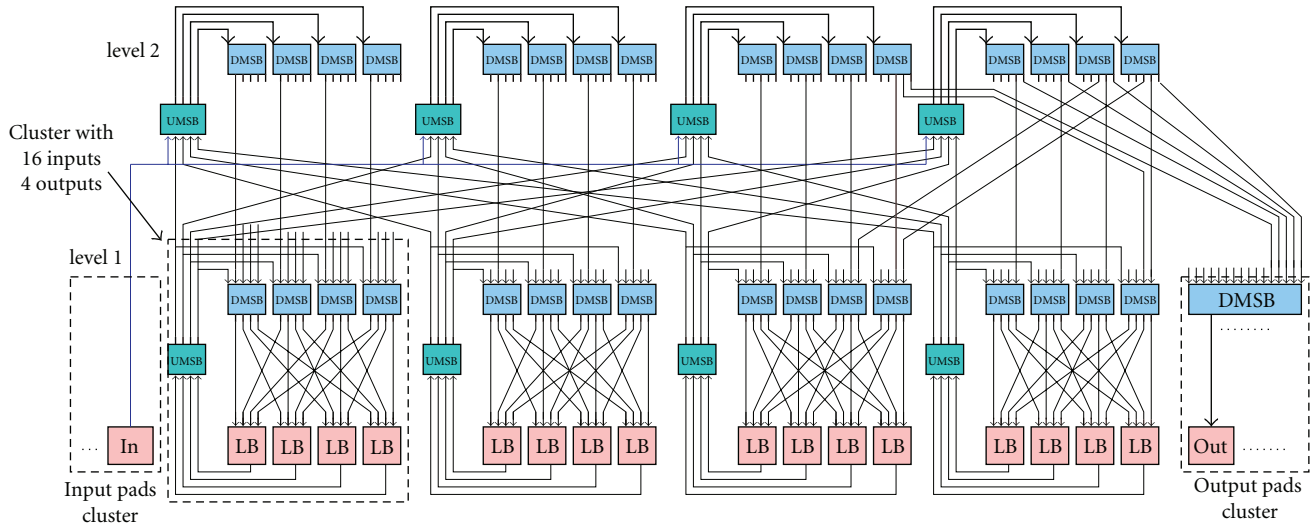


Figure 1: Tree-based interconnect: upward and downward networks.

these studies consider the connection block interconnect level and the intracluster crossbar separately. In [10], authors investigated joint optimization of both crossbars and proposed a new class of efficient topology. Nevertheless in the intracluster crossbar they optimized only the part connecting external signals to LBs inputs. Using a full crossbar to connect feedbacks (LBs outputs) to LBs inputs is very penalizing and imposes a very low bound on the cluster LBs number. For example, we assume that we have a cluster with 256 LBs and we use a full crossbar to connect feedbacks to 4-Lut inputs. This means that we need 256×1024 switches to route clusters internal signals only, which is very expensive. In this part, our first contribution corresponds to a joint optimization of connection blocks and intracluster interconnect topologies. We optimize both crossbars: (1) connecting external signals to LB inputs (2) connecting feedbacks to LB inputs. Our second contribution consists in using only single-driver interconnect based on unidirectional wires. As illustrated in [11], single-driver interconnect has a good impact on density improvement.

The remainder of the paper is organized as follows. Section 2 describes the Tree-based architecture. In Section 3 we propose suitable techniques to place and route netlists on the Tree-based architecture. In the following section, we present the effect of LUT and cluster size on Tree-based FPGA, then we evaluate architecture routability, and we compare it with the common VPR-Style Mesh architecture. Finally we present the cluster-based Mesh FPGA architecture and then we conclude this paper.

2. Tree-Based Interconnect

We propose a Tree-based architecture called MFPGA (Multilevel FPGA) where LBs (Logic Blocks) are grouped into clusters located at different levels. Each cluster contains a switch block to connect local LBs. A switch block is divided into MSBs (Miniswitch Blocks).

2.1. Interconnect Networks. This architecture unifies two unidirectional networks. The downward network uses a "Butterfly Fat Tree" topology to connect DMSBs (Downward MSBs) to LBs inputs. As shown in Figure 1, the number of DMSBs of a cluster located at level ℓ is equal to the number of inputs of a cluster located at level $\ell - 1$. The upward network connects LBs outputs to the DMSBs at each level. As shown in Figure 1, we use UMSBs (Upward MSBs) to allow LBs outputs to reach a large number of DMSBs and to reduce fanout on feedback lines. The number of UMSBs of a cluster located at level ℓ is equal to the number of outputs of a cluster located at level $\ell - 1$. UMSBs are organized in a way allowing LBs belonging to the same "owner cluster" to reach exactly the same set of DMSBs at each level. Thus positions, inside the same cluster, are equivalent, and LBs can negotiate with their siblings the use of a larger number of DMSBs depending on their fanout.

As shown in Figure 1, input and output pads are grouped into specific clusters and are connected to UMSBs and DMSBs, respectively. Thus, input pads can reach all LBs of the architecture, and output pads can also be reached by all the from different paths.

Using UMSBs and DMSBs greatly enhances routability, but it increases the interconnect switches number. However this increase is compensated by reducing in/out signals bandwidth of clusters at every level. In fact, netlists implemented on FPGA architecture often communicate locally (intraclusters) and this fact can be exploited to reduce the bandwidth of signals with inter-clusters communication. A good estimation of netlists communication locality is given by Rent's Rule [12]. Based on this estimation authors in [13] showed that most netlist Rent's parameters range between 0.5 and 0.65.

2.2. Architecture Rent's Parameter. We define Rent's parameter for an architecture as follows:

$$\mathrm{IO} = c \cdot m^{\ell \cdot p}. \tag{1}$$

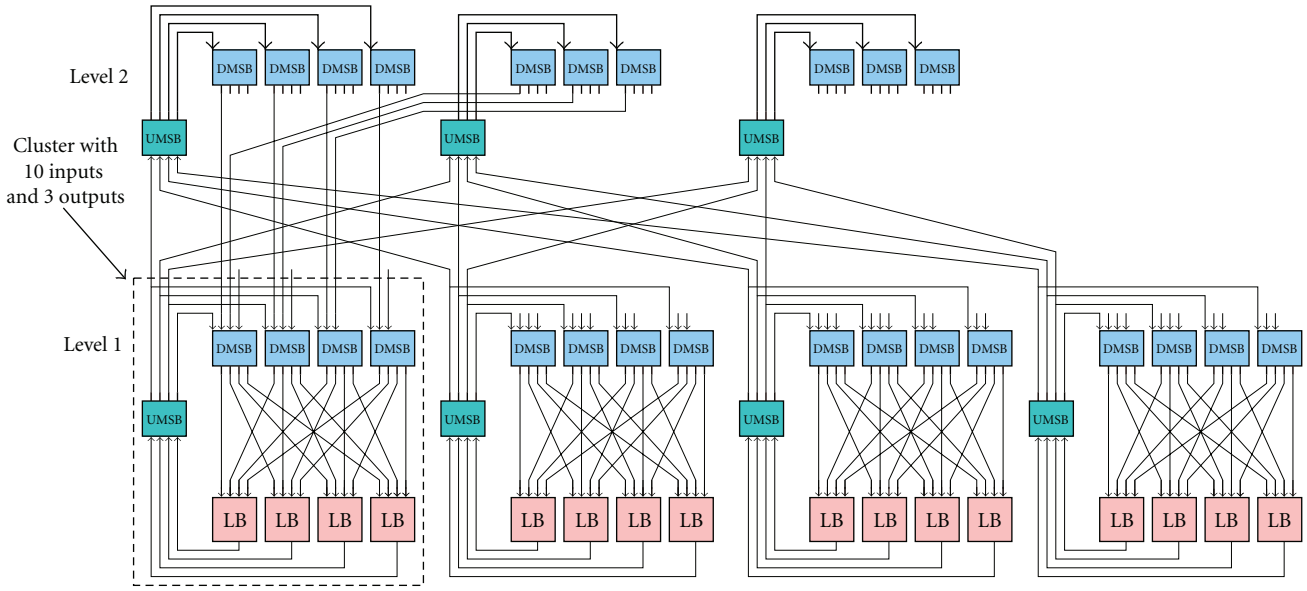


Figure 2: Tree-based interconnect depopulation using Rent's rule (level 1 with $p = 0.73$).

In this formula, ℓ is a Tree level, m is the cluster arity, c is the number of in/out pins of an LB, and IO is the number of in/out pins of a cluster located at level ℓ.

Intuitively, p represents the locality in interconnect requirements. If most of the connections are routed locally and only a few of them communicate to the exterior of a local region, p will be small. In Tree-based architecture, both the upward and downward interconnects populations depend on this parameter. As shown in Figure 2, we can depopulate the routing interconnect by reducing the number of inputs of each cluster of level 1 from 16 to 10 and outputs from 4 to 3 ($p = 0.73$). In this case, if we consider an architecture with 2 levels of hierarchy, we get a reduction in interconnect switches number from 521 to 368 (28%). However, a reduction in the value of p reduces the routability of the architecture too. Thus we must find the best tradeoff between interconnect population and logic blocks occupancy. As shown in [1], the best way to improve circuit density is to balance logic blocks and interconnect utilization. So in MFPGA architecture, interconnect occupancy is controlled by p and logic occupancy factor is controlled by N (the leaves (LBs) number in the Tree).

3. Configuration Flow

The way LBs are distributed between Tree clusters has an important impact on congestion. It is worthwhile to reduce external communications, since local connections are cheaper, not only in terms of delay but also in terms of routability, as this allows to get more levels (more paths) for connecting sources to destinations. Another way to decrease congestion consists in eliminating competition between sources to reach their sinks. This can be achieved by depopulating clusters based on netlist instances fanout. Instances with high fanout need more resources to reach their sinks. Thus in the partitioning phase, instances weights are attributed according to their fanout size. We use a top-down recursive partitioning approach. First, we construct the top level clusters, then every cluster is partitioned into subclusters, until the bottom of the hierarchy is reached. Since logic block positions inside the owner cluster are equivalent, the detailed placement phase (Arrangement inside clusters) is done randomly.

After placement, the routing process is started. Interconnect resources are presented by a routing graph with nodes corresponding to wires and LBs pins and edges to switches. We use the negotiation-based algorithm *Pathfinder* to route netlists signals [14].

4. Experimental Evaluation

To evaluate the proposed architecture performance, we place and route the largest MCNC benchmark circuits available and consider as a reference the optimized clustered Mesh (VPR-style) architecture. We use t-vpack [7] to construct clusters and the channel minimizing router VPR 4.3 [7] to route signals. VPR determines the optimal size as well as the optimal channel width W to place and route each benchmark circuit. In [15], author showed that the wiring structure in the fat-tree containing N logic blocks is sufficiently regular to permit a layout in $O(N)$ area (the area dictated by the nodes and switches) using $O(\log(N))$ wiring layers. Thus, in our area model we do not consider area dictated by wires. We estimate the layout area as the sum of areas required for all logic cells in an FPGA. As presented in Figure 3, switching cells depend on the interconnect structure and especially on wires directions (unidirectional/bidirectional). We use symbolic standard cells library [16] to estimate the FPGA required area. Different cells areas are presented in Table 1. We use bidirectional wires in the case of Mesh and unidirectional wires for MFPGA.

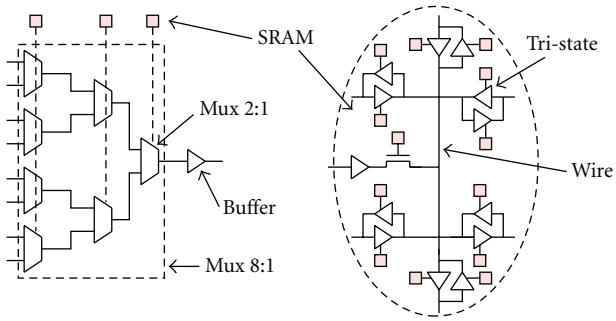


Figure 3: Unidirectional versus bidirectional wires.

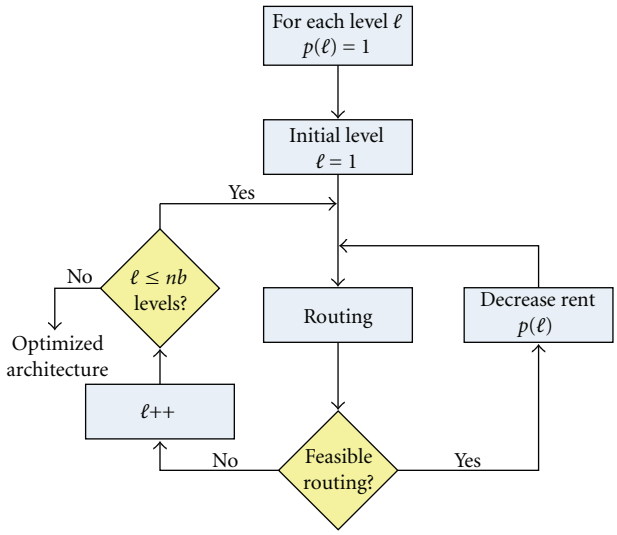


Figure 4: MFPGA architecture optimization flow (bottom-up approach).

Table 1: Standard cells characteristics.

Cell	Area λ^2
Sram	30×50
Tri-state	35×50
Buffer	20×50
Flip-flop	90×50
Mux 2 : 1	35×50

Table 2: Levels Rent's rule parameters (21 benchmark average).

Level	Circuits partitioning	Archi. top-down	Archi. bottom-up	Archi. random
1	0.64	0.98	0.79	0.88
2	0.55	0.88	0.74	0.79
3	0.50	0.80	0.77	0.76
4	0.49	0.75	0.86	0.73
5	0.45	0.59	0.87	0.7

4.1. Architecture Optimization. As explained in Section 2, MFPGA routability and switches number depend on 2 parameters: p (architecture Rent's parameter) and N (number of LBs in the architecture which defines occupancy ratio). To find the best tradeoff between device routability and switches requirement (area) we study MFPGA architectures with various N and p parameters. The purpose is to find for each netlist, the architecture with the smallest area that can implement it. N depends on Tree levels number and clusters arity. For a specific clusters arity, we determine the smallest levels number to implement the circuit. With our tools we can consider, in the same architecture, levels with different p values. Clusters located at the same level have the same Rent's parameter. In the case of Mesh, VPR adjusts the channel width W, and for Tree-based interconnect, we adjust Rent's parameters at every level in order to obtain the smallest architecture.

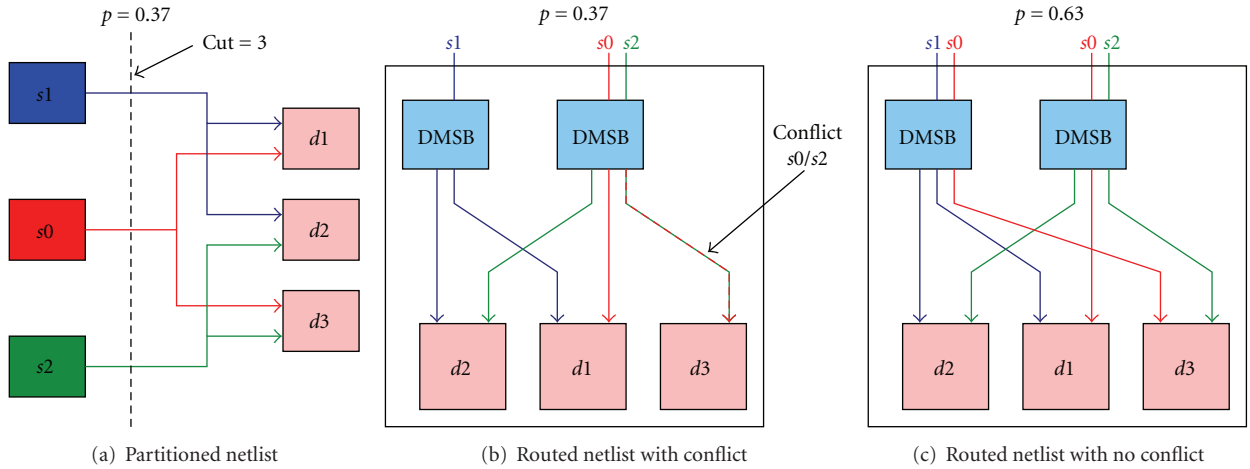
Just like VPR which applies a binary search to find the smallest value of channel width for Mesh architecture, we apply a binary search to determine the smallest value of Rent's parameters for each level of Tree-based architecture. Depending on levels order processing, we tested 3 different approaches.

(i) *Bottom-Up Approach.* As shown in Figure 4, we start by optimizing the lowest level up to the highest one. At each level we apply a binary search to determine the smallest number of input/output signals allowing to route the benchmark circuit.

(ii) *Top-Down Approach.* In this approach we start by optimizing the highest level down to the lowest one. In each level we apply a binary search to determine the smallest number of input/output signals allowing to route the benchmark circuit.

(iii) *Random Approach.* In this approach all levels are optimized in a random fashion. We choose a level randomly and we modify its input/output signals number depending on the previous result obtained at this level, then we move to another one. In this way we move randomly from one level to another until all levels are optimized.

The 3 approaches have the same objective and aim at reducing clusters signals bandwidth at each level. The difference is the order in which levels are processed. In Table 2, we show architecture Rent's parameter (at each level) obtained with each technique. The first column of the table shows Rent's parameters obtained after circuits partitioning. Results correspond to averages of all the 21 circuits. We notice that in all cases, architecture Rent's parameters are larger than partitioned circuits Rent's parameters. This is due to the depopulated switch boxes topology. In fact, to solve routing conflicts, a signal may enter from 2 different DMSBs to reach 2 different destinations located at the same cluster. In Figure 5 we show an example of a partitioned netlist to place and route onto an architecture with LBs inputs number equal to 2 (2 DMSBs in each cluster located at level 1) and clusters arity equal to 3. As shown in the figure if every signal comes from only one DMSB, we cannot solve conflicts. To deal with such a problem we propose to enter the signal driven by $S0$ from 2 different DMSBs. Thus, the



(a) Partitioned netlist (b) Routed netlist with conflict (c) Routed netlist with no conflict

FIGURE 5: A netlist routing example showing architecture Rent's parameter increase.

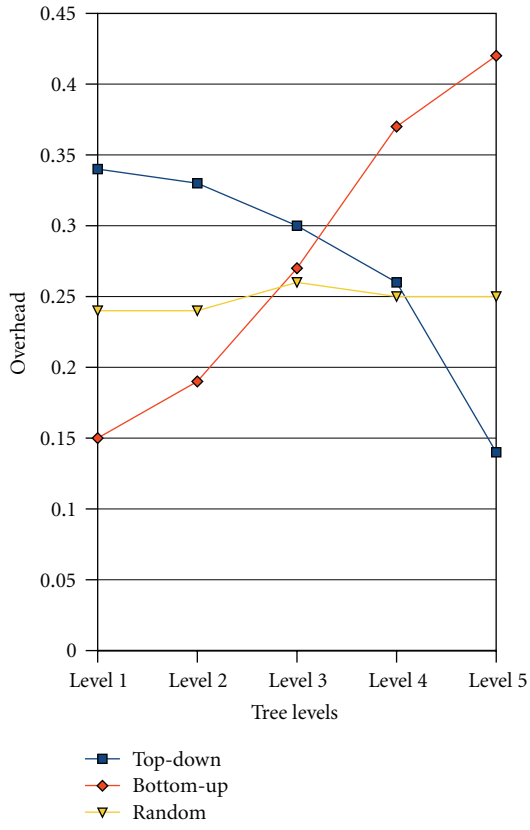


FIGURE 6: Overhead between Architecture and partitioned netlist Rent's parameters (21 benchmark average).

resulting architecture cluster degree is equal to 4, whereas the corresponding partition degree is equal to 3 (number of crossing signals).

In Figure 6, we show the average discrepancy between partitioning and architecture Rent's parameters with each optimizing approach. We notice that in the case of the top-down (bottom-up) approach, overhead increases when we go down (up) in the Tree. This was expected since the top-down (bottom-up) approach first optimizes high (low) levels. With the random approach, we notice that levels overheads are balanced.

TABLE 3: Area and performance comparison between various optimizing approaches (21 benchmark average).

Optimizing approach	Area (λ^2) $\times 10^6$	Critical path switches
Top-down	1498	98
Bottom-up	1326	106
Random	1221	101

A comparison of the average results obtained from the 3 optimizing approaches is shown in Table 3. We notice that with the random approach we obtain the smallest area. This means that optimizing levels randomly allows to avoid local minima and helps to obtain a balanced congestion distribution over levels. The bottom-up approach provides a smaller area than the top-down one, but it is penalizing in terms of critical path switches number. In fact, starting by optimizing low levels means that local routing resources are intensively reduced and signals are routed with resources located at higher levels. Consequently, signals routing uses more switches in series.

To reduce the gap between circuit and architecture Rent's parameters, we must improve the partitioning tool (especially the objective function) to reduce congestion and resources (clusters inputs) required to route signals.

4.2. Clusters Arity and LUT Size Effect. In this section we evaluate the effect of LUTs size k (number of LUT inputs) and cluster arity on area, performance, and static power of MFPGA architecture. In order to evaluate this effect, we performed a series of experiments where LUT size ranges from 3 to 7 and cluster arity ranges from 4 to 8 for each benchmark. Thus we have results for a set of 25 different architectures for each benchmark. First, as shown in Figure 7,

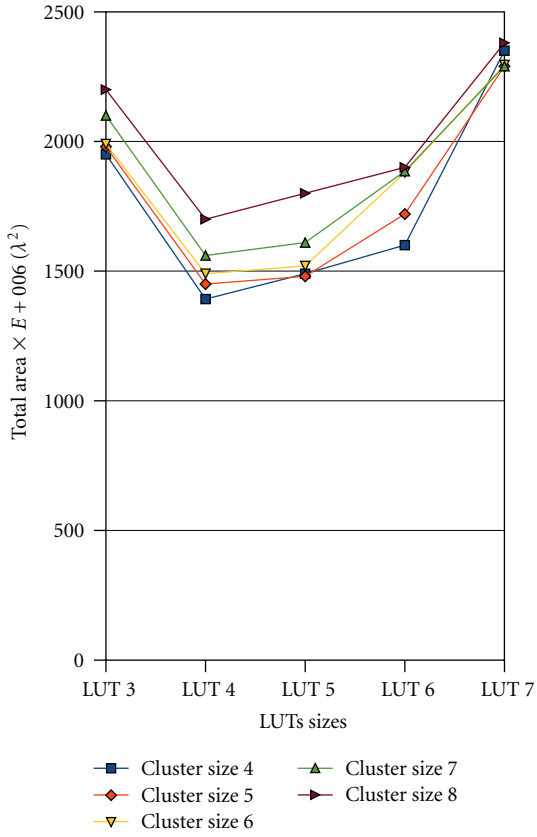


Figure 7: Total area for clusters sizes 4–8 (21 benchmark average).

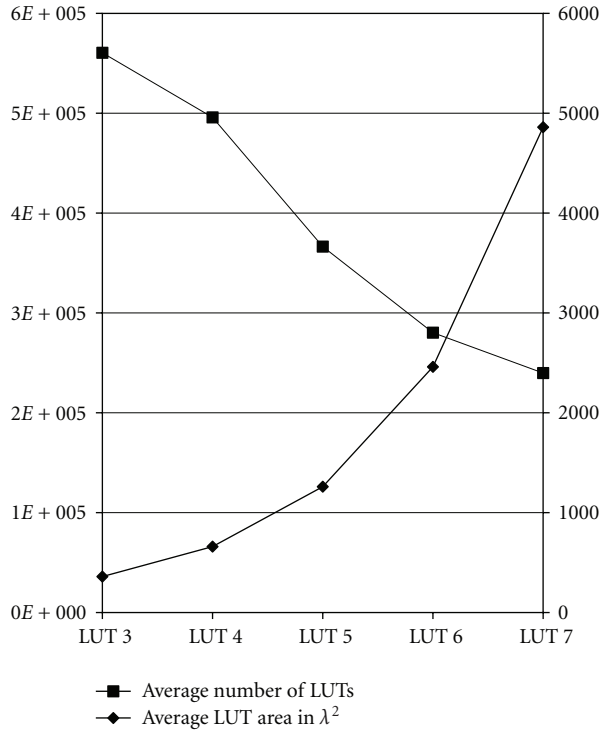


Figure 8: LUTs number and LUT area versus LUT size (for cluster arity = 4).

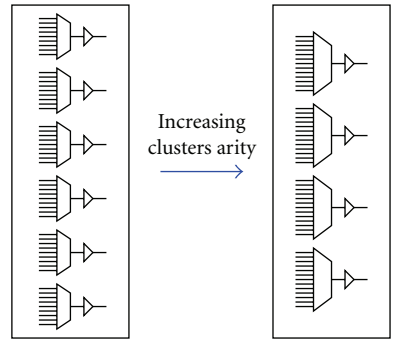


Figure 9: Varying clusters arity ⇒ varying multiplexers sizes and number.

we evaluate the effect of LUT size and cluster arity on MFPGA area. Results correspond to the average area of the 21 largest circuits. We notice that initially there is a reduction in area between $k = 3$ and $k = 4$, but afterwards there is an increase in area with the rise in LUT and cluster sizes. It can be noted from the figure that for the same LUT size, higher cluster arities give higher area values. This is due to the fact that with an increase in the cluster size, there is a decrease in the number of clusters required to implement the circuit but at the same time there is an increase in the area per cluster due to increased value of inputs/outputs bandwidth. In addition, when clusters arity increases, the required multiplexers number decreases but their size grows larger (see Figure 9) and consequently the bound on area efficiency goes down. Hence, these effects combine together and result in higher area values for same LUT size but with higher arity size.

In order to analyze further LUTs size effect on area we divided it into two parts, logic blocks area and interconnect area. From our experimentation we notice that logic area increases with LUT size. This area is the product of the total number of LUTs times the area per LUT. A plot of these two components for clusters arity equal to 4 is shown in Figure 8 (the left vertical axis presents area per LUT in (λ^2) and the right vertical axis presents LUTs number). The logic block area grows exponentially with LUT size as there are 2^k bits in a k-input LUT. Though there is a decline in the number of LUTs with an increase in k (because each LUT can implement more logic functions), the rate of increase in area is steeper than the rate of decrease in LUTs number. Concerning the interconnect area, we notice that it decreases when LUT size increases. Since the rise in logic area is steeper than the decline in interconnect area, we obtain the upward trend in Figure 7.

The second key metric is the critical path delay. Since we have no accurate wire length estimation (we do not have a complete layout generator yet), we only evaluate the number of switches crossed by the critical path. Figure 10 shows the average critical path switches number across the 21 circuits as a function of clusters arities and LUTs sizes. Observing the figure, it is clear that increasing clusters arity and LUTs size decreases the number of switches crossed by the critical path. This behavior is due to the decrease of the number of

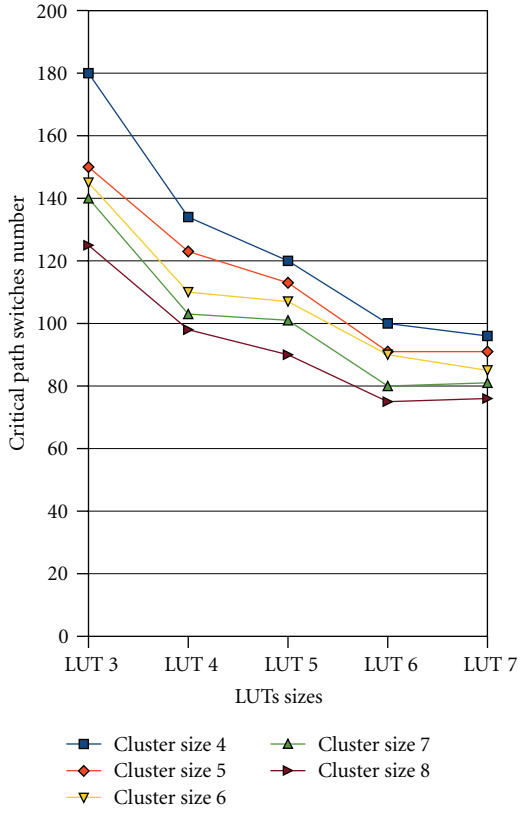


Figure 10: Critical path switches number clusters sizes 4–8 (21 benchmark average).

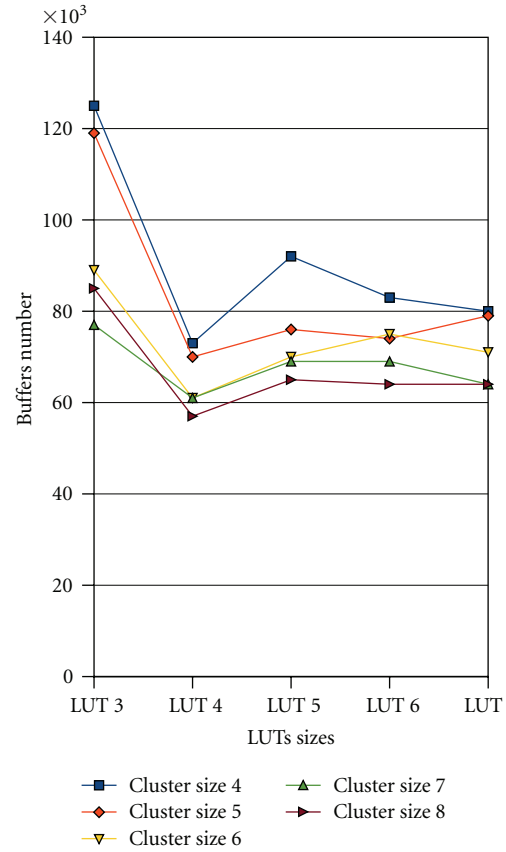


Figure 11: Buffers number clusters sizes 4–8 (21 benchmark average).

Table 4: Levels Rent's parameters for 2 circuits.

Circuits	Level 1	Level 2	Level 3	Level 4	Level 5
apex2	1	0.89	0.86	0.84	0.77
tseng	0.79	0.79	0.79	0.72	0.67

LUTs and clusters in series on the critical path. Nevertheless, to get an idea about the accurate delay we have to consider the increase of intrinsic LUT delay when its size increases.

According to [17], buffers and SRAM are the major factors behind static power dissipation. Therefore, in order to get an idea about LUT and cluster size effect on static power dissipation, we evaluate buffers and SRAM cells numbers as a function of LUT and cluster size. We assume that we insert a buffer at the output of every multiplexer. We notice, as shown in Figure 11, that initially there is a reduction in buffers number when 4-LUTs are used instead of 3-LUTs, and afterwards buffers number increases with LUT size. Also it can be noticed from the Figure 11 that for the same LUT size, higher cluster sizes have a smaller number of buffers. This is due to the fact that when we increase cluster arity, multiplexers sizes increase and their number decreases and consequently buffer number also decreases (see Figure 9). As shown in Figure 12, we notice that SRAM points number decreases when LUT size increases from 3 to 4 and increases afterwards. Clearly, results for all clusters sizes show consistently that LUT size 4 gives minimum leakage energy compared to other LUT sizes. This result is expected since LUT size 4 achieves the highest total-area efficiency.

To get a good tradeoff between area and path delays reduction, using different LUTs sizes is necessary. This was confirmed by the Stratix II architecture [18] where authors showed that the use of ALMs (Adaptive Logic Blocks) gives a good tradeoff between area and critical path delay reductions.

4.3. Area Efficiency. Here we compare MFPGA to the Mesh-based architecture in terms of area efficiency. In both cases we consider architectures with clusters arity 4 and LUT size 4. In each case, we determine the smallest architecture implementing every benchmark circuit. In the case of Mesh we use VPR to find the smallest channel width, and in the case of MFPGA we use the random optimizing approach described in Section 4.1 to determine the smallest Rent's parameters.

In Figure 13, we observe that the Tree-based architecture has a better density for all the 21 benchmark circuits. On average with the Tree architecture we save 56% of the total area. We achieve a 42% area gain with *alu*4 (smallest circuit

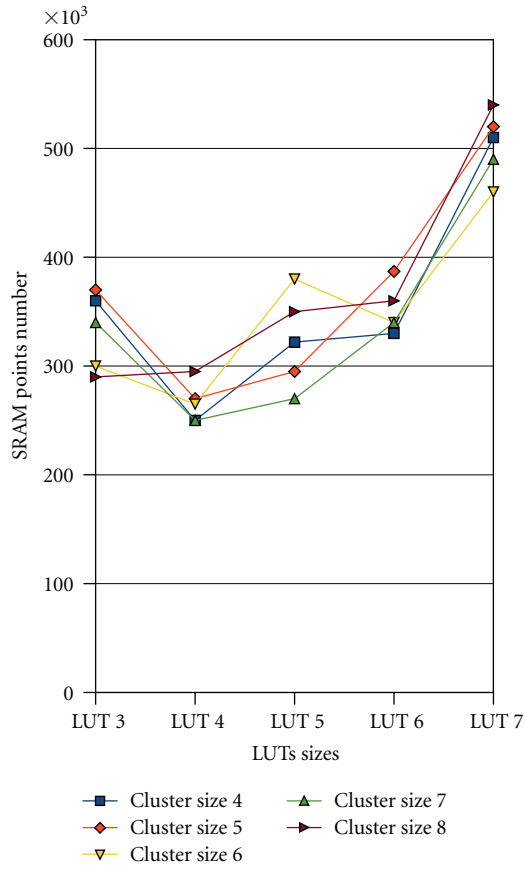


Figure 12: SRAM cells number clusters sizes 4–8 (21 benchmark average)

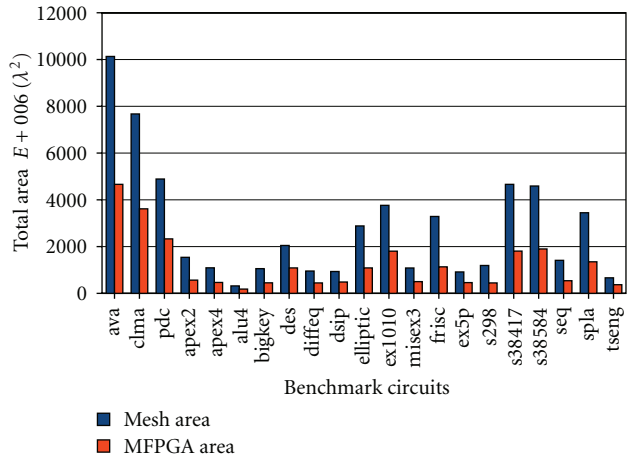


Figure 13: MFPGA area versus Mesh area (21 benchmark circuits).

584 LUTs) and 60% with *ava* (largest circuit 14964 LUTs). This confirms that Tree-based interconnect is very attractive for both small and large circuits.

We compare the areas of both architectures using a refined estimation model of effective circuit area. The Mesh area is the sum of its basic cells areas like SRAMs, tri-states, multiplexers and buffers. The same evaluation is made for the Tree, composed of SRAMs, multiplexers, and buffers. Both architectures use the same symbolic cells library.

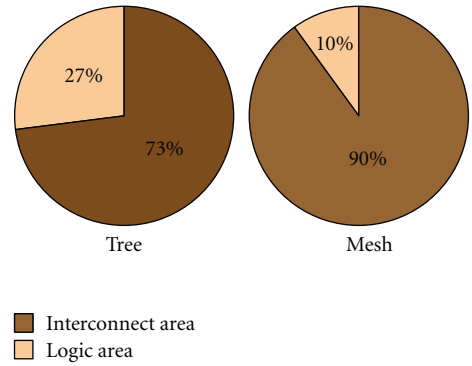


Figure 14: Area distribution between interconnect and logic blocks: Tree and Mesh cases.

The Tree architecture efficiency is due essentially to its ability to control simultaneously logic blocks occupancy and interconnect population, based on LBs number N and architecture Rent's parameter p, respectively. For example in the case of apex2 circuit, we use an architecture with high logic occupancy (91%) and high Rent's parameters as shown in Table 4. In the case of tseng circuit, we have a low occupancy (51%) and we achieve routability with a low architecture Rent's parameters as illustrated in Table 4. This confirms that we can balance interconnect and logic blocks utilization with the help of logic occupancy decreasing and congestion spreading. In fact, we use a high-interconnect/low-logic utilization approach which is in direct opposition to the high logic utilization approach that has been adopted for Mesh-based FPGA [7]. As shown in Figure 14, unlike Mesh case where interconnect occupies 90% of the overall area, in Tree-based architecture interconnect occupies 73%. Compared to Mesh architecture, we have a 20% lower occupancy; the extra logic area allows us to exploit interconnect better and to reduce its area by 69%.

5. Unifying Mesh and Tree

We showed that with a Tree-based topology, we obtain good density and we cut area by a factor of 2 compared to Mesh. Nevertheless, based on our layout experimentation we noticed that this Tree-based architecture is penalizing in terms of physical layout generation. It does not support scalability and does not fit with a planar chip structure, especially for large circuits. Conversely the Mesh and in specially the Mesh of Tree [19] has a good physical scalability since it corresponds to an array of repeated tiles. Once a tile layout is generated we can abut it to generate layout of selectable size. In addition, as shown in Figure 15, unlike

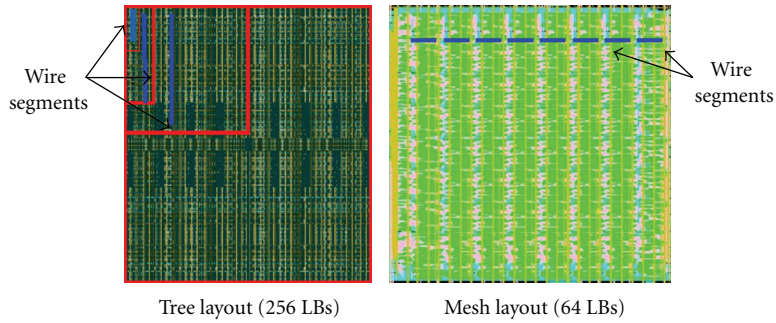


Figure 15: Layout view of Mesh and Tree interconnect structures.

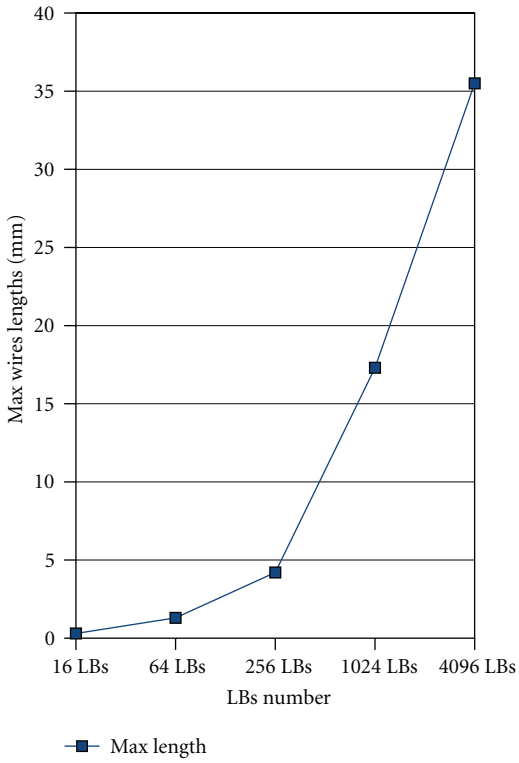


Figure 16: Maximum wire lengths depending on Tree size (arity 4).

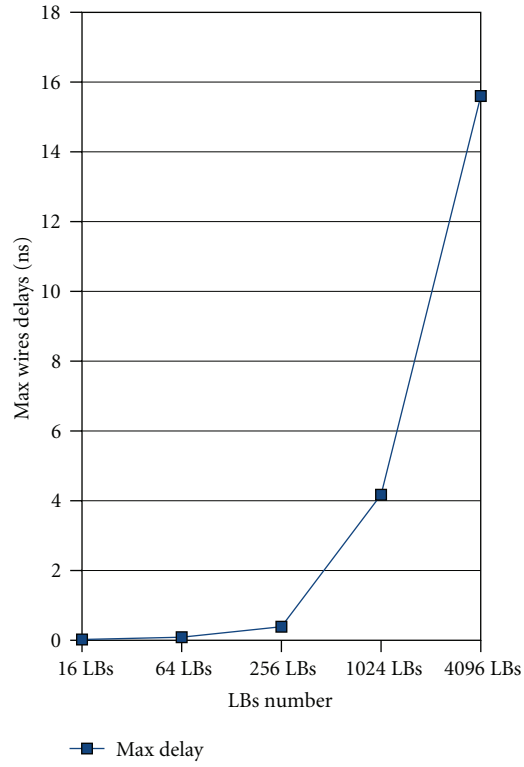


Figure 17: Maximum wire delays depending on Tree size (arity 4).

Mesh where the largest wiring distance is fixed, in the case of a Tree, wiring length increases as we move towards higher levels. An accurate estimation of wire lengths is presented in Figure 16. Maximum wires lengths are evaluated based on layout generated in 130 nm technology. We notice from Figure 17 that in the case of architectures larger than 512 LBs, wires delays become critical and dominant compared to switches delays. It is essential to cut down wire lengths to reduce the quadratic delays growth. For this purpose we propose to use a Mesh interconnect structure from this break even point (512 LBs) to reduce wires delays. In this way wires lengths depend only on Mesh clusters size and no more on total architecture size.

To take advantage of the positive points of both topologies, we propose an architecture where LBs are connected into a cluster (Mesh cluster) with a local interconnect built as a Tree. Mesh clusters are connected with an external interconnect with a Mesh topology. We use the same Tree topology presented previously. In the Mesh interconnect we use only unidirectional wires, since in [11], authors show that single driver interconnect can lead to 25% improvement in area density. Each Mesh cluster is surrounded by four channels which are connected by Switch Boxes (SBs). We do not use connection blocks in the Mesh to connect channel tracks to cluster inputs and outputs. Actually, as presented in [10], interconnect is better optimized when

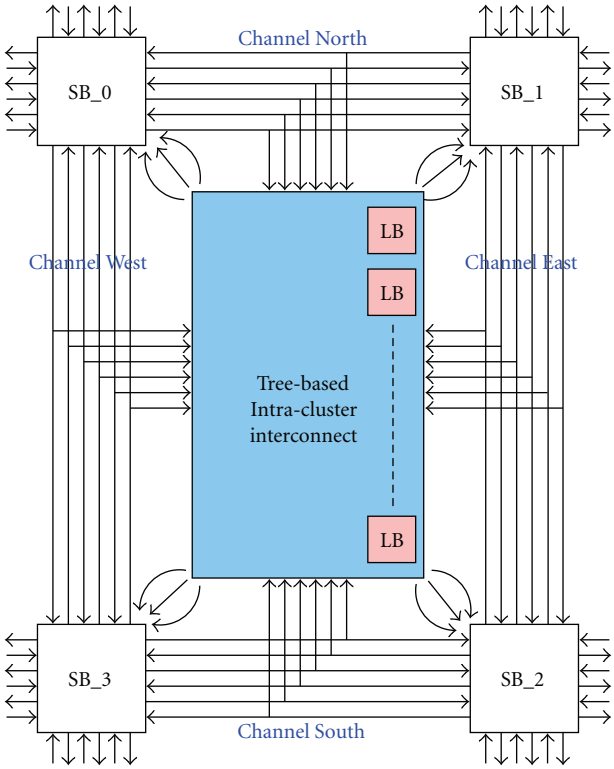


Figure 18: Node of Mesh of Tree architecture.

a connection block is combined with the cluster local interconnect.

5.1. Mesh Cluster Interface. As presented in Figure 18, each cluster is connected to the 4 adjacent channel tracks. The cluster input and output connectors are equally distributed on the 4 sides. On all sides we have the same number of inputs and outputs. The distribution of inputs and outputs is illustrated in Figure 19. Input signals (output signals) are grouped together into input Superpins (output Superpins) located at level ℓ_{in} (ℓ_{out}) of the Tree.

(i) Each input Superpin contains 4 inputs connected to the 4 adjacent channels. Each input is connected to all UMSBs located at level $\ell + 1$ of the Tree. In this way the 4 inputs are logically equivalent and can reach all Tree LBs.

(ii) Each output Superpin contains 4 outputs connected to the 4 adjacent switch boxes. Each output is connected to all DMSBs placed at level $\ell + 1$ of the Tree. In this way the 4 outputs are logically equivalent and can be reached from all Tree LBs.

This distribution has an important impact on routability and eliminates constraints in the placement of LBs inside Tree clusters. All 4 Mesh cluster sides have the same number of inputs and outputs. Side inputs and outputs numbers depend on the number of Tree leaves and on the level where they are connected:

$$Nb_{\text{in}} = \frac{N}{k^{\ell_{\text{in}}+1}},$$
$$Nb_{\text{out}} = \frac{N}{k^{\ell_{\text{out}}+1}}, \tag{2}$$

where k is Tree clusters arity. N is the number of Tree leaves. ℓ_{in} and ℓ_{out} are levels where input and output Superpins are located, respectively. As explained previously the Tree-based local interconnect of a cluster can be depopulated using Rent's parameter.

5.2. Mesh Routing Interconnect. As presented in Figure 18, cluster-based Mesh architecture is composed of logic blocks clusters, switch boxes, and in/out pads. Interconnection between clusters is done by routes through switch boxes, along horizontal and vertical routing channels.

In the Mesh interconnect structure we use only single-driver unidirectional wires; in [11], authors show that single-driver-based interconnect leads to a 25% improvement in area density. Each Mesh cluster is surrounded by 4 channels which are connected by Switch Boxes (SBs). As described in Figure 18, Mesh cluster input signals are connected to the 4 adjacent channel tracks. Thus, channel width W is given by

$$W = \frac{Nb_{\text{in}}}{4} = \frac{N}{k^{\ell_{\text{in}}+1}}. \tag{3}$$

A Mesh Switch Box (SB) allows to connect horizontal and vertical channel tracks together and also to cluster outputs. SB inputs come from the 4 channel tracks and from the 4 adjacent clusters outputs. SB outputs are connected to the 4 adjacent horizontal and vertical channels. Since we use a single-driver-based interconnect, each SB output is driven by a multiplexer. Switch boxes have a "disjoint" topology. As presented in Figure 20(b), input track j of channel i is connected to output track j of channel h with $h \neq i$. SBs allow also to connect Mesh cluster outputs to channels tracks. As illustrated in Figure 20(a), each cluster output is connected to all switch box outputs located on the 4 sides.

5.3. Configuration Flow. The configuration flow used to implement benchmark netlists on the proposed architecture is described in Figure 21. First netlist instances are partitioned among the N Mesh clusters. We obtain one external netlist describing Mesh clusters communication and N internal netlists describing instances communication inside each cluster. The external netlist is used to place Mesh clusters on the 2D grid array. All internal netlist instances are partitioned among Tree clusters. We use a multilevel mincut partitioner based on FM refinement heuristics. Once all instances are placed on the Mesh of Tree sites, we run signals routing. Routing consists in assigning netlist signals to routing resources in such a way that no routing resource

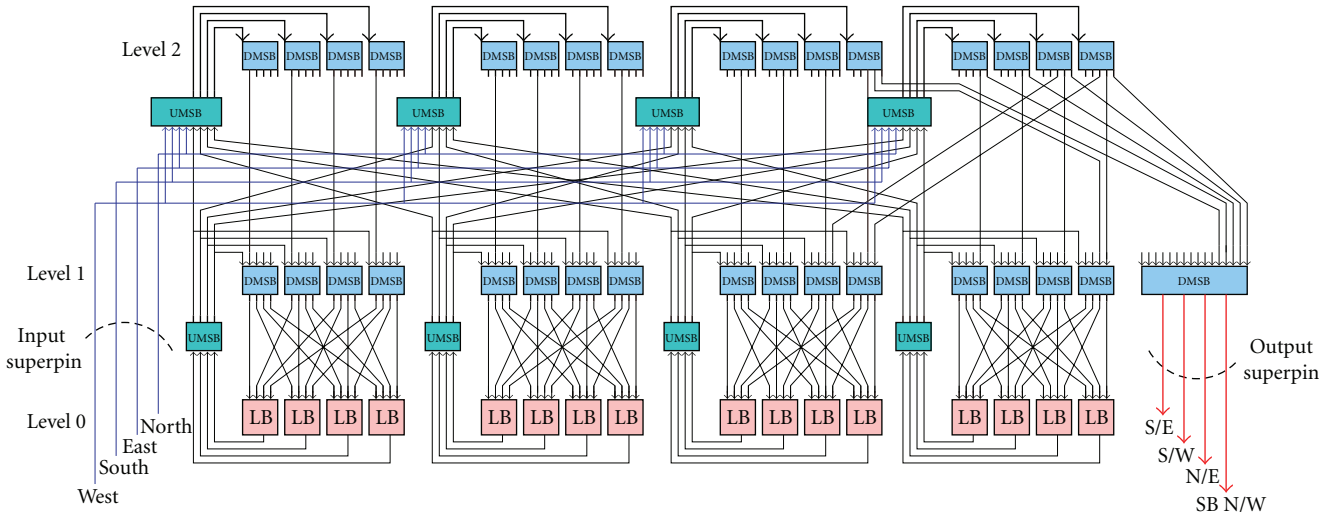
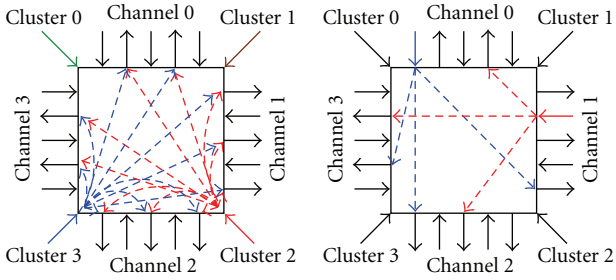


Figure 19: Cluster interface: external input and output connections.



(a) SB: Cluster outputs to channel tracks

(b) SB: channel tracks to channel tracks

Figure 20: Mesh switch box topology.

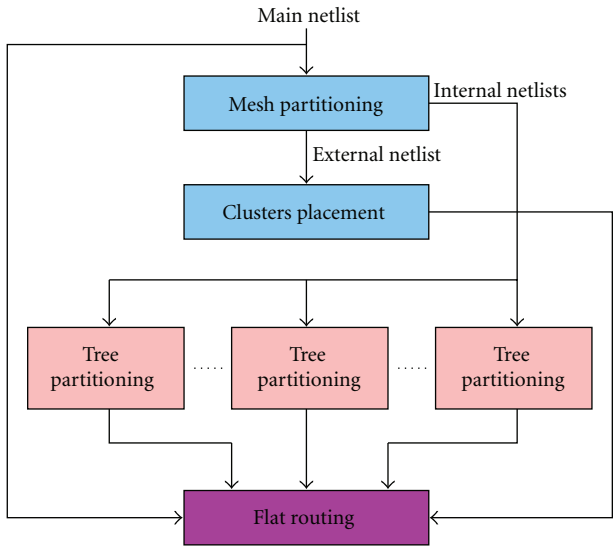


Figure 21: Mesh of Tree configuration flow.

is shared by more than one net. For this purpose we model routing resources by a flat direct graph, where nodes correspond to wires, and LBs pins and edges correspond to switches. We use the *PathFinder* algorithm to route signals on the resulting graph.

5.4. Experimentation. To evaluate the proposed architecture and tool performances, we place and route the 3 largest MCNC benchmark circuits and the *ava* circuit which is the largest design containing only LUTs. We consider as references the optimized cluster-based (VPR-style) Mesh and the MFPGA architectures. We map the 4 largest benchmark circuits on the Mesh of Tree architecture. We consider an architecture with unidirectional wires and Mesh clusters size equal to 256 LBs. Every cluster has 256 inputs and 64 outputs equally distributed on the 4 sides. This is obtained by putting input Superpin at Tree level 0 and output Superpin at level 1. For every benchmark circuit we adjust only the Mesh clusters array size. We do not tailor every Tree interconnect flexibility to every circuit. The Mesh channel width is equal to 64 and Tree signals growth rate p is equal to 0.88. The Mesh of Tree switches requirement and its distribution among Tree and Mesh levels is presented in Figure 22. As shown in Figure 23, we notice that, compared to the VPR-based Mesh architecture, total area is reduced by 42%. This is due essentially to the depopulated intracluster crossbar. In fact with $p = 0.88$ the Tree required switches number is equal to 20×10^3 switches only.

We also notice that, compared to a stand-alone Tree, the total area is increased by 28%. This increase is compensated by the Mesh of Tree layout generation simplicity and wires length reduction, compared to stand-alone Tree, especially when we target large circuits sizes. In this case, wires lengths depend only on Mesh clusters sizes and not on architecture total LBs number.

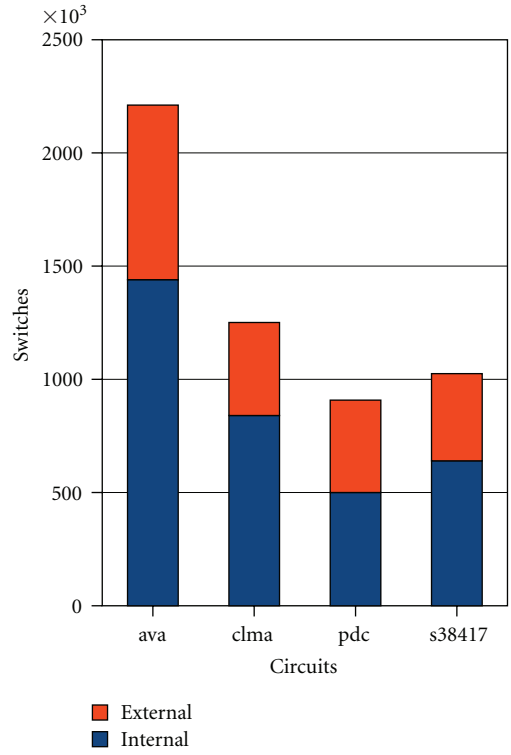


Figure 22: Interconnect distribution in Mesh of Tree architecture.

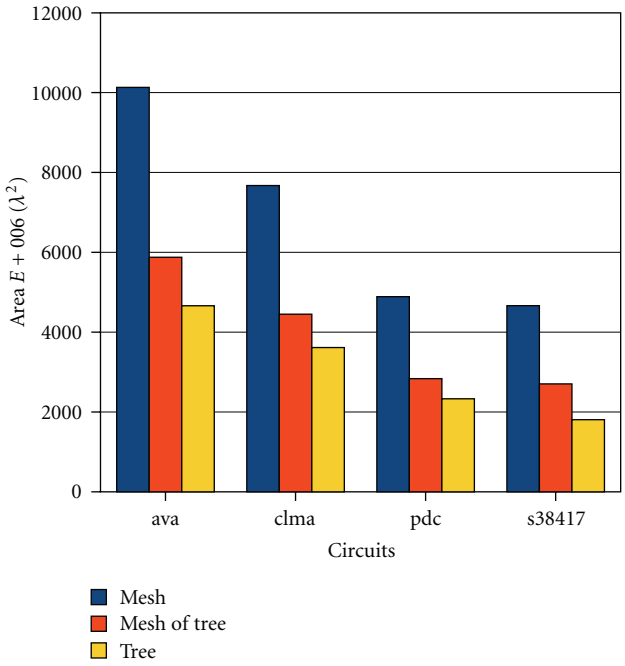


Figure 23: Comparison of various FPGA architectures areas.

6. Conclusion

We proposed a Tree-based architecture with high interconnect and low logic utilizations. Based on the largest MCNC benchmark implementation, we showed that this architecture has better area efficiency than the common VPR-Style clustered Mesh. We showed that in general LUTs with size 4 and cluster size 4 produce most efficient results in terms of area and static power dissipation for Tree-based FPGA. We also determined the evolution of the number of switches crossed by the critical path as a function of LUT and cluster size and we showed that LUTs with higher input size, and with higher cluster size can be more optimal in terms of performance though they are not very good in terms of density. Nevertheless, this Tree-based architecture is penalizing in terms of physical layout generation. To deal with such problem we proposed an architecture unifying both Mesh and Tree strong points. The Mesh of Tree has a good physical scalability: once the cluster layout is generated we can abut it to generate Mesh layouts with the desired size and shape factor. The proposed Mesh of Tree architecture is a good tradeoff between area density and layout scalability.

References

[1] A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," in *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 69–78, Monterey, Calif, USA, February 1999.

[2] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 3, pp. 288–298, 2004.

[3] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1217–1225, 1990.

[4] S. Kaptanoglu, G. Bakker, A. Kundu, I. Corneillet, and B. Ting, "A new high density and very low cost reprogrammable FPGA architecture," in *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 3–12, Monterey, Calif, USA, February 1999.

[5] D. Hill and N. Woo, "The benefits of flexibility in lookup table-based FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 2, pp. 349–353, 1993.

[6] Xilinx Corp, http://www.xilinx.com.

[7] V. Betz, A. Marquardt, and J. Rose, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[8] D. Lewis, V. Betz, D. Jefferson, et al., "The Straix™ routing and logic architecture," in *Proceedings of the 11th ACM/SIGDA ACM International Symposium on Field Programmable Gate Arrays (FPGA '03)*, pp. 12–20, Monterey, Calif, USA, February 2003.

[9] G. Lemieux and D. Lewis, *Design of Interconnection Networks for Programmable Logic*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[10] W. Feng and S. Kaptanoglu, "Designing efficient input interconnect blocks for LUT clusters using counting and entropy," in *Proceedings of the 15th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '07)*, pp. 23–32, Monterey, Calif, USA, February 2007.

[11] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in FPGA interconnect," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 41–48, Brisbane, Australia, December 2004.

[12] B. Landman and R. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Transactions on Computers*, vol. 20, no. 12, pp. 1469–1479, 1971.

[13] J. Pistorius and M. Hutton, "Placement rent exponent calculation methods, temporal behaviour and FPGA architecture evaluation," in *Proceedings of the International Workshop on System Level Interconnect Prediction*, pp. 31–38, Monterey, Calif, USA, April 2003.

[14] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for FPGAs," in *Proceedings of the 3rd ACM International Symposium on Field-Programmable Gate Arrays (FPGA '95)*, pp. 111–117, Monterey, Calif, USA, February 1995.

[15] A. DeHon, "Compact, multilayer layout for butterfly fat-tree," in *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '00)*, pp. 206–215, Bar Harbor, Me, USA, July 2000.

[16] A. Greiner and F. Pechêux, "Alliance: a complete set of CAD tools for teaching VLSI design," in *Proceedings of the 3rd EuroChip Workshop*, pp. 230–237, September 1992.

[17] S. Kaptanoglu, "Power and the future FPGA architectures," in *Proceedings of the International Conference on Field Programmable Technology (ICFPT '07)*, pp. 241–244, Kitakyushu, Japan, December 2007.

[18] D. Lewis, E. Ahmed, G. Baeckler, et al., "The Stratix II logic and routing architecture," in *Proceedings of the 13th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '05)*, pp. 14–20, Monterey, Calif, USA, February 2005.

[19] A. DeHon, "Unifing mesh and tree-based programmable interconnect," *IEEE Transactions on VLSI Systems*, vol. 12, no. 10, pp. 1051–1065, 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 453970, 14 pages
doi:10.1155/2009/453970

Research Article

A Decentralised Task Mapping Approach for Homogeneous Multiprocessor Network-On-Chips

Peter Zipf,[1] Gilles Sassatelli,[2] Nurten Utlu,[3] Nicolas Saint-Jean,[2] Pascal Benoit,[2] and Manfred Glesner[3]

[1] *Digital Technology Lab, University of Kassel, Wilhelmshöher Allee 73, 34121 Kassel, Germany*
[2] *Laboratoire d'Informatique, de Robotique et de Microélectroniqe de Montpellier (LIRMM), University of Montpellier II, UMR CNRS 5506, 161 rue ADA, 34392 Montpellier Cedex 5, France*
[3] *Institute of Microelectronic Systems, Darmstadt University of Technology, Karlstrasse 15, 64283 Darmstadt, Germany*

Correspondence should be addressed to Peter Zipf, zipf@uni-kassel.de

Received 27 December 2008; Accepted 25 May 2009

Recommended by Michael Huebner

We present a heuristic algorithm for the run-time distribution of task sets in a homogeneous Multiprocessor network-on-chip. The algorithm is itself distributed over the processors and thus can be applied to systems of arbitrary size. Also, tasks added at run-time can be handled without any difficulty, allowing for inline optimisation. Based on local information on processor workload, task size, communication requirements, and link contention, iterative decisions on task migrations to other processors are made. The mapping results for several example task sets are first compared with those of an exact (enumeration) algorithm with global information for a 3×3 processor array. The results show that the mapping quality achieved by our distributed algorithm is within 25% of that of the exact algorithm. For larger array sizes, simulated annealing is used as a reference and the behaviour of our algorithm is investigated. The mapping quality of the algorithm can be shown to be within a reasonable range (below 30% mostly) of the reference. This adaptability and the low computation and communication overhead of the distributed heuristic clearly indicate that decentralised algorithms are a favourable solution for an automatic task distribution.

1. Introduction

On-chip multiprocessing provides the computing power and parallelism required for many of today's real-world applications with high data rates. The diminishing returns of Instruction Level Parallelism (ILP) point the interest to higher levels of applications, where explicit Thread Level Parallelism (TLP) can be exploited [1]. A logical consequence of increasing performance demands is to use both ILP and TLP simultaneously by integrating a large number of processors in one Multiprocessor System-on-Chip (MPSoC). At the same time, reduced clock frequencies for the individual processor cores enable a large reduction of the overall power consumption while keeping the system performance up.

Multiprocessor systems can only be utilised sufficiently, if the software running on them can be separated into sets of communicating tasks working in parallel. These tasks are then distributed over a set of processors sharing the workload. For a well-known set of tasks and workloads, the distribution can be precalculated for an optimal mapping. For applications with unpredictable workload like, for example, user-induced multimedia processing, and subsequently unpredictable changes in active tasks and communication requirements, a run-time task mapping depending on the actual resource utilisation must be applied to balance the processor loads.

In this paper we present a decentralised task mapping heuristic for task sets on an MPSoC. The heuristic running on each processor is capable of reconfiguring the system by migrating individual tasks to neighbouring processors based on the local workload, task sizes, and communication requirements of the tasks to be migrated. It is not restricted to a final set of tasks but can also handle task sets added during operation, thus supporting a reconfiguration at task level. Due to its scalability, a homogeneous Network-on-Chip (NoC) structure is used as the underlying hardware

architecture, which is essential for the developed task mapping heuristic. An experimental implementation of the multiprocessor platform based on interconnected FPGA prototyping boards is used to investigate the potential of decentralised task distribution and workload balancing algorithms.

1.1. Multiprocessor Network-on-Chips. An MPSoC is a special form of SoC; where the functional modules are all processor modules. Due to the advantages on-chip design offers, like a free choice of bus bit widths or high data transfer rates, such systems can be adapted very well to their specific requirements. Based on the envisioned application scenario of multimedia workloads, which are characterised by structured and regular computations, some additional desired properties for the system can be derived. This refers mainly to the communication model, the processor types, and the physical interconnect architecture. Multiprocessor systems can be based on shared memory or message passing communication. For large high-performance systems with up to several hundred processors, only a communication based on message passing is reasonable [2, 3], combined with distributed local memory. To enable a simple task distribution, a homogeneous MPSoC should be preferred, where each node consists of an identical processor to present a uniform (homogeneous) array.

The components of MPSoCs are usually connected by point-to-point or bus-based structures. Both interconnect concepts cannot be scaled well for larger numbers of processors, for example, exceeding 50. A Multiprocessor Network-on-Chip (MPNoC) uses a Network-on-Chip [4] structure to interconnect its processor modules. A set of interconnection segments is combined to a network by routers. Data sent from one Processor is then relayed from one router to the next until it reaches its destination [5]. Such a MPNoC called HS-Scale [6] is used in our work.

1.2. The Task Mapping Problem. For the envisioned data flow applications, a high overall system throughput is the dominant requirement, surpassing short latencies as needed, for example, in closed loop control systems. In order to improve throughput, tasks must be mapped in the right way. The main question to be answered for a task mapping is: what makes one mapping better than another? Consequently, the objective is to reduce (a) the average distance of travelling data packets and (b) the workload on the individual processors. In addition, the maximum bandwidth on the communication links should not be exceeded. These objectives are specific for on-chip scenarios where individual interconnects are not the most dominant limitation and the network topology including all its parameters is fixed and known in advance.

Two major concepts in developing task mapping strategies are the graph theoretic approach and the mathematical programming approach [7]. Although rapid advances in both the methodology and application of graph theoretic models have been realised, many models actually are special types of linear programming problems [8]. Task mapping considering traffic generation is a nonlinear problem, which limits the usability of common graph theoretic approaches. Due to the unsatisfactory support of nonlinear task mapping by graph-based methods, in this work flexible mathematical programming for developing an algorithm is used.

1.3. Section Overview. Section 2 introduces some relevant previous work on the task mapping problem for multiprocessor systems. Section 3 describes the heuristic algorithm developed and an exact algorithm used for comparison. Section 4 discusses some experimental results obtained by running the algorithms on a set of example task sets. In Section 5 the performance of the heuristic algorithm for larger network processing unit (NPU) arrays is investigated based on a large number of random task sets. Section 6 concludes with a summary and some final remarks.

2. Related Work

The aim of this work is to develop a run-time task mapping algorithm for MPNoCs to balance the system throughput. This is done by considering the two conflicting requirements *maximisation of average processor utilisation* and *minimisation of the contention on links* caused by intertask communication. A classification of some relevant related work on task mapping is given in Table 1. The main categories are the factors taken into account for the mapping (computation and/or traffic), the flexibility of the mapping process (static or dynamic), and the way it is implemented (centralised or decentralised).

The first category is based on the target factors taken into account to achieve the mapping goal. In [9], only the network bandwidth is considered but not the computing requirements of the applications. The aim of [10] is the minimisation of total communication time for sets of similar tasks. Other factors like congestion are not considered. Also, workload balancing is only done by mapping exactly one task to one processor. A more general load balancing model considering job and resource migration is used in [11]. As communication bandwidth is assumed to be sufficient, the mapping depends only on the communication distance and is independent of the network traffic. In contrast, a mapping optimisation regarding computation and traffic is given in [12]. The goal is to minimise the total execution and communication costs. Communication costs are used as an attracting force between tasks, causing them to be assigned to the same processor. The costs of incompatibilities between tasks are used as a repulsive force, causing a task distribution over several processors. Communication costs occur if two tasks are assigned to different processors and are independent of the congestion on the links. They are not explicitly specified, but occur as the multiplication of the communication flow between two tasks and the distance between the processors they are mapped to. The mapping problem is solved by a Max Flow/Min Cut algorithm in combination with a greedy algorithm. In [13] also the total execution time is minimised by weighting the computation of each task and each interaction between tasks. The resulting

Table 1: Classification of related work on task mapping.

Group	Description	Examples
Computation or Traffic:	Maximum utilisation of processors or minimum traffic generation	[9–11]
Computation and Traffic:	Maximum utilisation of processors and minimum traffic generation	[12–14]
Static Mapping:	Offline or predictive mapping at design time	[12, 15–17]
Dynamic Mapping:	Supports run-time task migration (and injection)	[18–20]
Central Mapping:	Global view, Master Slave	[12, 14, 19–21]
Decentrlised Mapping:	Local view	[22]

cost function is minimised by a hybrid of a genetic algorithm and mean field annealing. The turnaround time is improved in [14]. After defining execution and communication costs simulated annealing is used.

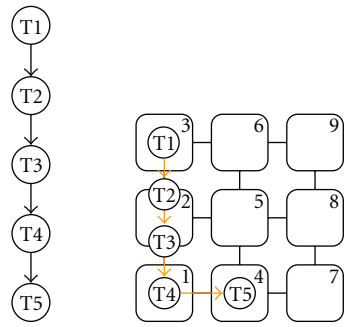
While workload balancing tries to exploit parallel execution in space by distributing all tasks regarding the computation demand evenly among the processors, intertask communication tends to exploit computation in time, by mapping the whole application to a single processor in order to save bandwidth of communication links [23]. Task Mapping differs regarding the time at which assignment decisions are made. Most authors propose the use of static mapping [12, 15–17], according to most of the current real-time operating systems of embedded systems [18]. Static mapping is less complex and easier to design than dynamic mapping. The assignment is defined prior to the application execution at design time and is not changed any more later on. To improve the performance of dynamic workloads at run-time, task migration has been used [18–20] to relocate tasks in order to distribute the workload more homogeneously among the resources. Differently from task migration, dynamic mapping can insert new tasks into a system at run time [24].

For the decision-making policy of task mapping, the two fundamental models centralised and decentralised can be considered. In a centralised model [12, 14, 19–21], one specialised master processor and an arbitrary number of slave processors are used [20]. The master has global knowledge of the application characteristics and of the distributed system [12]. It performs task mapping, aiming at an equal distribution of the load among the slave processors and communication links. The centralised task mapping allows a globally coordinated and hence efficient placement mechanism, however at the cost of scalability. An increasing number of processors in future systems or a great number of tasks will overload the master. In decentralised models, the authority for task mapping is shared among all processors. Because of the absence of a global view, knowledge of application and processor characteristics is shared by the exchange of messages. All decisions for the task mapping are made from local interaction laws.

Typical applications running in MPNoCs like multimedia and networking display a dynamic workload of tasks. This implies a varying number of tasks running simultaneously [24]. It is impossible to foresee and specify an appropriate response for every potential run-time scenario before the application execution. Therefore, unpredictable information like task arrival times, workload of processors, and contention on the links must be gathered during execution. This work considers dynamic mapping for MPNoCs, which supports varying workloads by task injection and target load distribution by task migration. Tasks are mapped on the fly, according to computation and communication requirements, following a distributed (decentralised) mapping scheme, considering both computation (workload) and traffic data.

3. The Heuristic Task Mapping Algorithm

Since scalability of the platform architecture and programming model will be a major challenge for MPSoC designs in the years to come, a platform providing a large number of processors must discard all non-scalable properties. Our hardware platform HS-Scale [6] is a homogeneous MPSoC based on programmable RISC processors, small distributed memories, and an asynchronous Network-on-Chip (NoC). The software model is a multithreaded sequential programming model with communication primitives handled at run-time by a simple multitasking operating system specifically developed for the platform; the threads are described in C language. The HS-Scale framework guarantees any application to be executed independent of the platform settings, specifically the number of processing elements (PE) and the chosen task mapping. The communication is abstracted via communication primitives, so that tasks can communicate with each other without knowing their position in the system. The communication primitives were derived from 5 of the 7 layers of the OSI model, allowing transparent data communications between tasks either locally or remotely: the routing is done following a dynamic routing table. If the task is local, the writing of data is done on a local software FIFO. If it is a remote task, the operating system must assure that there is enough space for the remote software FIFO to avoid deadlocks on the network. This is done using dedicated functions. As soon as the OS gets a positive answer, it can start encapsulating and sending the data packets to the



(a) Task graph (b) Task placement

Figure 1: A task graph and its placement on an NPU array.

Task	T1	T2	T3	T4	T5
NPU	3	2	2	1	4

Figure 2: Task placement representation.

remote task while the remote task can de-encapsulate and receive the data packets and write them to its local software FIFO. A lightweight operating system has been developed for the specific needs of the MPNoC platform. The OS provides preemptive task switching and communication support for task interactions using the communication primitives [6].

Load balancing, the overall communication bandwidth, and the local communication bandwidth have to be considered for the task mapping. This section introduce the implemented algorithms after defining the underlying model and a mathematical problem formulation.

3.1. Problem Definition and Model Formulation. To reduce the average distance of travelling data, the number of data packets and the distance between the communicating network processing units (NPUs) must be known.

The mapping alternatives are determined by using an appropriate solution representation and by modifying representations (solutions). Every possible solution can be represented by a Table with two rows (see also Figure 2). The first row is an ID list of all existing tasks without repetition. This constraint results from the fact that each task must only be mapped exactly once. The second row contains the IDs of used NPUs. Because each NPU has multitasking capabilities which enables a time-sliced execution of tasks, a repetition of NPU IDs is allowed. Not all NPUs need to be used and thus some need not appear in the second row. As an example, the task graph of Figure 1(a) with five consecutive tasks is mapped on the array with 3×3 NPUs shown in Figure 1(b). The according solution representation Table is shown in Figure 2.

The target hardware architecture is a homogenous array. Therefore it is possible to assign any task to any NPU. New solutions can easily be generated by exchanging NPUs in the second line of the solution representation by other existing NPUs. This is equivalent to the combinatoric variation with repetition, where order matters and an object can be chosen more than once. The number of possible variations with repetition is given by

$$\mathrm{NPU}^{\mathrm{Task}}, \tag{1}$$

where NPU is the number of available NPUs to be chosen from and Task is the number of tasks to be placed.

The problem can now be formulated as follows. Given the computation time for every task and the data flow between communicating tasks, find a task placement that reduces the distance through which data travels and balances computation load. Each task has to be assigned to a single NPU and each NPU can execute multiple tasks.

The communication costs $f_{i,k}$ between task i and task k depend on the distance $d_{j,l}$, determined by the position of NPU j to which task i is assigned ($x_{i,j} = 1$) and NPU l on which task k is assigned ($x_{k,l} = 1$). The problem is a quadratic assignment problem (QAP) [8]. The formulation of the overall bandwidth minimisation can be given as.

$$\text{minimise } z = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{(k=1,k\neq i)}^{n}\sum_{(l=1)}^{m} f_{i,k}\cdot d_{j,l}\cdot x_{i,j}\cdot x_{k,l} \tag{2}$$

The load balancing between NPUs can be considered as the linear assignment problem (LAP), where each task i in the task graph has been assigned a constant computational complexity t_i, where $t_{i,j}$ is this cost when task i is assigned to NPU j:

$$\text{minimise } z = \sum_{i=1}^{n}\sum_{j=1}^{m} t_{i,j}\cdot x_{i,j}. \tag{3}$$

subject to

$$\sum_{j=1}^{m} x_{i,j} = 1, \quad i = 1,\ldots,n, \tag{4}$$

where n is the number of tasks, and m is the number of NPUs. This constraint guarantees that task i is assigned to exactly one NPU.

To consider local bandwidth, the congestion $c_{j,l}$ on the links between NPU j and NPU l must also be included. A complete formulation of the objective function can be to minimise z, where

$$\begin{aligned} z = {} & \alpha\cdot\sum_{i=1}^{n}\sum_{j=1}^{m} t_{i,j}\cdot x_{i,j} \\ & + \beta\cdot\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{(k=1,k\neq i)}^{n}\sum_{(l=1)}^{m} f_{i,k}\cdot d_{j,l}\cdot x_{i,j}\cdot x_{k,l} \\ & + \gamma\cdot\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{(k=1,k\neq i)}^{n}\sum_{l=1}^{m} c_{j,l}\cdot x_{i,j}\cdot x_{k,l} \end{aligned} \tag{5}$$

subject to (4).

Equation (5) considers load balancing, overall bandwidth and local bandwidth, weighted by the scaling factors $0 \leq \alpha, \beta, \gamma \leq 1$.

Table 2: Area scalability and power consumption results for 90 nm technology.

No. of NPUs	1	2	2×2	3×3
Area (mm^2)	18.22	36.63	73.61	165.30
Power (mW/MHz)	2.56	5.14	10.34	23.26

Table 3: FPGA synthesis results.

Module	NPU	Router	Processor	Other
No. of Slices	2496	683	1462	351
% of Slices	14.4	3.9	8.5	2.0

3.2. The Task Mapping Algorithms. Three task mapping algorithms have been implemented. The first one is an exact algorithm based on complete enumeration. It delivers one solution which is guaranteed to be as good as any other objective function value. This algorithm is only used for small examples (up to 9 NPUs and 11 tasks) and as a reference, because (5) contains a modified QAP formulation (QAP problems have been shown to be NP-hard [25]) and for a complete enumeration NPU^{Task} solutions have to be generated. The program flow is shown in Figure 3. All solutions are generated, evaluated, and the best value encountered is returned as the result.

The second algorithm is a constructive algorithm. Its results are used as the starting point for the main improvement heuristic. to produce a feasible initial mapping solution, the constructive algorithm is run on one task injection (boundary) NPU. Initially only, global information is available, because no task is running on any NPU. Also, the 2D mesh structure of the hardware is used based on a reachability measure.

All NPUs are evaluated regarding their reachability. For illustration the array given in Figure 4 is used. The distance between two NPUs is given by the number of required hops, as shown in Figure 4(a). The sum of hops from NPU 1 to all other NPUs is 18. Applying this procedure to all NPUs gives their reachability. It can be seen in Figure 4(b) that the NPU with the best reachability is in the centre. To avoid overloading NPUs with good reachability, the reachability of NPUs which run tasks is penalised proportional to their computation time.

The program flow of the constructive algorithm is shown in Figure 5. Output tasks are sinks in the task graph. All tasks on the injection NPU are mapped to the remaining NPUs. This is done starting with the input task of each application's task set and continued by moving on to its successor tasks.

The constructive algorithm is activated once. Later the improvement algorithm is started on all NPUs until a steady state is reached. The closer the initial solution to the optimum, the fewer the number of required operations during the following improvement procedure. However, a good solution usually requires a complex algorithm and high computational effort. The proposed constructive heuristics balances the desire for a high quality initial solution and a simple algorithm, which is easy to implement and does not require extensive computations.

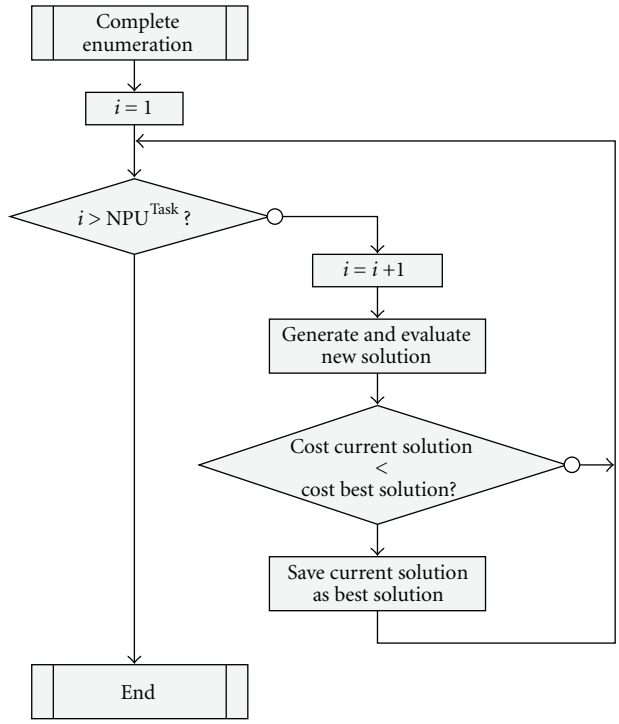
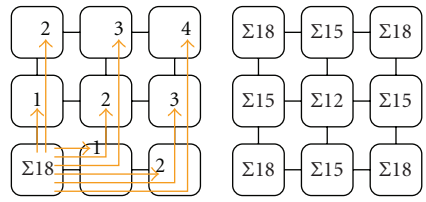


Figure 3: Program flow: exact algorithm.



(a) Reachability of NPU 1 (b) Reachability of all NPUs

Figure 4: Reachability calculation.

The third algorithm is a hybrid tabu search and force directed improvement algorithm. It is a distributed algorithm meant to run on each NPU if required. A model of spring-connected weights is used as its basis. Weights correspond to tasks and springs to communication between the tasks. A spring will try to pull its tasks closer together or push them apart, depending on its stiffness which is proportional to the quality (objective function rating) of the considered neighbourhood. The algorithm starts with a stiffness of distance 0 and 1. The NPU of the sending task and its neighbours which have a distance of one hop are considered first. If the objective function value worsens, the stiffness value is incremented to consider a growing neighbourhood. Assignment of tasks with high communication demands are prioritised. In order to achieve proper tradeoffs between time spent looking for solutions and the quality of the solutions found, a feature of tabu search, the candidate list strategy feature of the tabu search is applied [26]. The candidate list is used as a penalty Table

Table 4: Evaluation results for the task graph (TG) examples for $\alpha = \beta = \gamma = 1$ in (5) for constructive algorithm (CA), improvement algorithm (IA), and exact algorithm (EA).

Examples	Value	CA	IA	EA
TG 1	OF	1,161,862	941,508	941,380
	%	123.4	100.0	100.0
	LB	256	384	320
	OB	256	384	320
	CL	1,161,350	940,740	940,740
TG 2	OF	1,657,440	942,148	941,764
	%	176,0	100,0	100.0
	LB	832	704	512
	OB	448	704	512
	CL	1,656,160	940,740	940,740
TG 6	OF	1,359,998	762,921	762,157
	%	178.4	100.1	100.0
	LB	6,720	7,932	6,848
	OB	2,240	2,688	3,008
	CL	1,351,038	752,301	752,301
TG 7	OF	1,378,832	909,235	732,787
	%	188.2	124.1	100.0
	LB	8,568	7,680	5,376
	OB	2,432	2,624	2,176
	CL	1,367,832	898,931	725,235
TG 8	OF	2,565,628	2,108,464	2,009,720
	%	127.7	104.9	100.0
	LB	704	832	640
	OB	704	832	640
	CL	2,564,220	2,106,800	2,008,440
TG 9	OF	1,591,952	869,200	868,816
	%	183.2	100.0	100.0
	LB	384	640	448
	OB	128	640	448
	CL	1,591,440	867,920	867,920
TG 15	OF	2,958,737	2,334,168	2,033,084
	%	145.5	114.8	100.0
	LB	768	1,024	832
	OB	704	1,024	832
	CL	2,957,265	2,332,120	2,031,420
TG 17	OF	2,819,944	2,406,374	2,008,914
	%	140.4	119.8	100.0
	LB	896	1,280	512
	OB	768	704	512
	CL	2,818,280	2,404,390	2,007,890
TG 24	OF	359,000	232,869	230,629
	%	155.7	101.0	100.0
	LB	960	2,880	1,216
	OB	896	1,472	896
	CL	357,144	228,517	228,517
TG 26	OF	1,820,626	942,789	942,597
	%	193.1	100.0	100.0
	LB	1,952	1,088	928
	OB	672	960	928
	CL	1,818,002	940,741	940,741

Table 5: Comparison of the constructive algorithm (CA), improvement algorithm (IA) and exact algorithm (EA).

Algorithm		CA	IA	EA
HW throughput (bytes/s)		133,118	240,365	266,237
% of Exact		49.99	90.28	100
Contention on links	256 byte link	1	0	0
	128 byte link	1	2	2
	64 byte link	1	7	4
Overall Communication OB		448	704	512
Computational Load CL		1,656,160	940,740	940,740

Table 6: Relative OF-minima (values in % of random mapping).

	Minima [NPUs]		Relative value [%]	
Task count	Annealing	Heuristic	Annealing	Heuristic
10	9	16	55.4	64.5
20	16	16	49.9	63.8
30	25	25	52.5	65.3
40	36	36	56.2	64.9
50	36	49	61.9	66.6
60	49	49	65.5	67.4
70	49	64	68.1	68.8
80	49	81	71.8	69.8
90	64	64	71.9	69.7

which includes one element for each NPU. For example, a penalty is applied if the algorithm could not attain a better objective function value. After a certain value in the candidate list is reached, for example, a certain number of unsuccessful repetitions of the algorithm, the corresponding NPU is marked tabu and is no longer allowed to run the task mapping algorithm. This procedure provides three mechanisms.

(i) Avoid cycling by setting NPUs tabu if the improvement algorithm repeatedly cannot reach a better objective function value.

(ii) Intensification of the search by remaining NPUs, excluding nonpromising regions.

(iii) Termination criterion for the algorithm: eventually, all NPUs will be marked tabu.

The program flow of the improvement algorithm is shown in Figure 6. First, the algorithm checks whether a task is assigned to the NPU on which it runs. If no task is available, the tabu candidate originally set to zero will be incremented by one. Otherwise, all successor tasks are determined, except those with output flow which are not allowed to migrate (sinks). If no valid successor exists, the tabu candidate value is increased by one and the task is excluded from consideration. If successors exist, the successor with maximum receiving flow will be selected. According to the objective function, the NPU costs consist of the computation workload on the sending NPU, the computation workload on the receiving NPU, the distance between task and considered successor multiplied by the flow, and finally of the congestion between sending and receiving NPU. The flow to the successor is then checked to determine if it is so high that it is worth assigning both tasks to execute on the same NPU, despite the increasing computational demand. If this is not the case, the successor will be assigned to the neighbour NPU of distance 0 and 1, with minimum sum of congestion and NPU workload. If the NPU costs worsen, the neighbourhood is expanded to a distance of 2 and the value of the tabu candidate is incremented. This procedure of neighbourhood expansion will be continued until the NPU costs are at least equal to the old NPU costs. The improvement algorithm is repeated on the considered NPU as long as its repetition is not forbidden by the tabu list.

4. Experimental Results

A complete synthesisable RTL model of the HS-Scale hardware has been designed. The VHDL model was synthesised with a 90 nm ST Microelectronics design kit. The NPU clock has been constrained to 3 nanoseconds allowing a 300 MHz clock frequency. Table 2 summarizes the results. The model has been placed and routed with a 64 KB local memory, which occupies 87% of the total NPU area. Of the remaining 13%, the processor occupies 54%, the router 38%. Other elements (UART, interrupt controller, network interface, etc.) occupy about 7%. Table 2 clearly shows the areascalability of the MPNoC hardware platform and gives the estimated power consumption.

The very first validations of the system were performed using RTL simulations. Since this method is too slow for

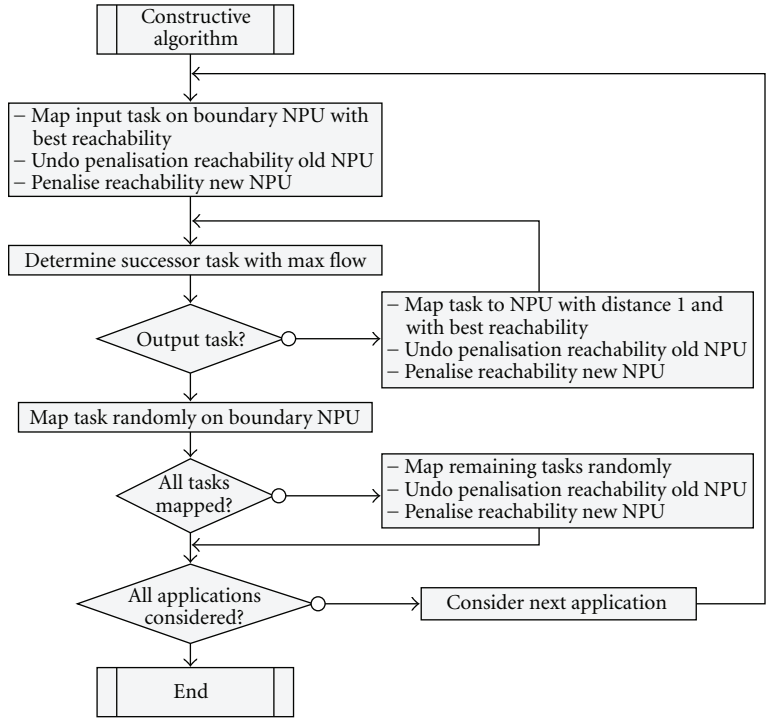


Figure 5: Program flow: constructive algorithm.

running realistic application scenarios, a prototype system using Xilinx Spartan-3 XC3S1000 FPGAs on Xilinx Starter Kit FPGA boards was realised. Table 3 gives the device utilisation for a single NPU on a single XC3S1000 FPGA providing 17,280 logic cells. Each NPU is placed on one board. The complete prototype is then composed of several prototyping boards connected by ribbon cables. This allows for easy extension of the system by adding further FPGA boards.

A set of 27 task graphs was used as examples to evaluate the quality of the constructive and improvement algorithms. The properties of the graphs, that is, computational and communication requirements, are taken from real applications, for example, Motion JPEG video-codec. Variations were generated by duplicating tasks to enable load sharing or by iterative execution of tasks. The example task sets range between 5–11 tasks, distributed to 1–4 independent applications, that is, independent data flows. Figures 7(a), 7(b), and 8(a) show examples of the task graphs used, including computational and communication requirements (given in clock cycles and bytes resp.). In task graph 6 (Figure 7(a)) the tasks 2, 3, and 4 have been replicated twice for load sharing, while at the same time also increasing communication (arrows). Due to the problem complexity for the exact mapping solution, the target array was limited to 3×3 NPUs. Table 4 shows a representative selection of the data obtained from the evaluation. The rows for local bandwidth or link contention (LB), overall bandwidth (OB), and computational load (CL: z from (3)) show the respective algorithm representation of these values. The individual values for the objective function (OF) of the calculated mappings and their relation to the exact results are given.

The average deviation between the results of the improvement algorithm and the exact solution is 6.47% for the given examples, and the maximum difference is below 25%. TG 1 contains task 2 with a computational requirement of 494,810. TG 2 is a parallelised version of TG 1, where task 2 has been replicated once (task 20), resulting in a computational requirement of 247,405 for each of task 2 and 20. It can also be seen that a load balancing can easily be done at the cost of increased communication (OB of the exact algorithm increases from 320 to 512, corresponding to the two additional communication links with costs of 128 and 64 bytes per block calculation). The CL of the exact algorithm for TG 1 and TG 2 are identical because no changes in the computational complexity arises by duplicating tasks. From the viewpoint of the computational loadbalancing, it can be seen by comparing the CL values of TG 2, that the construction algorithm provides an inferior solution, whereas the improvement algorithm and the exact algorithm provide solutions with equal quality (visualised in Figure 9).

Figure 8 shows the task graph model of application example 2 with 6 tasks and a 3×3 NPUs graph. The task graph of Figure 8(a) was mapped by all three algorithms on an FPGA-based NPU array implementing the 3×3 array of Figure 8(b). Figure 9 shows the mapping results for the three algorithms. Table 5 gives the corresponding throughput numbers measured on a VHDL simulation of the hardware platform running at 7 MHz. It can be seen that the result of the improvement algorithm for the example

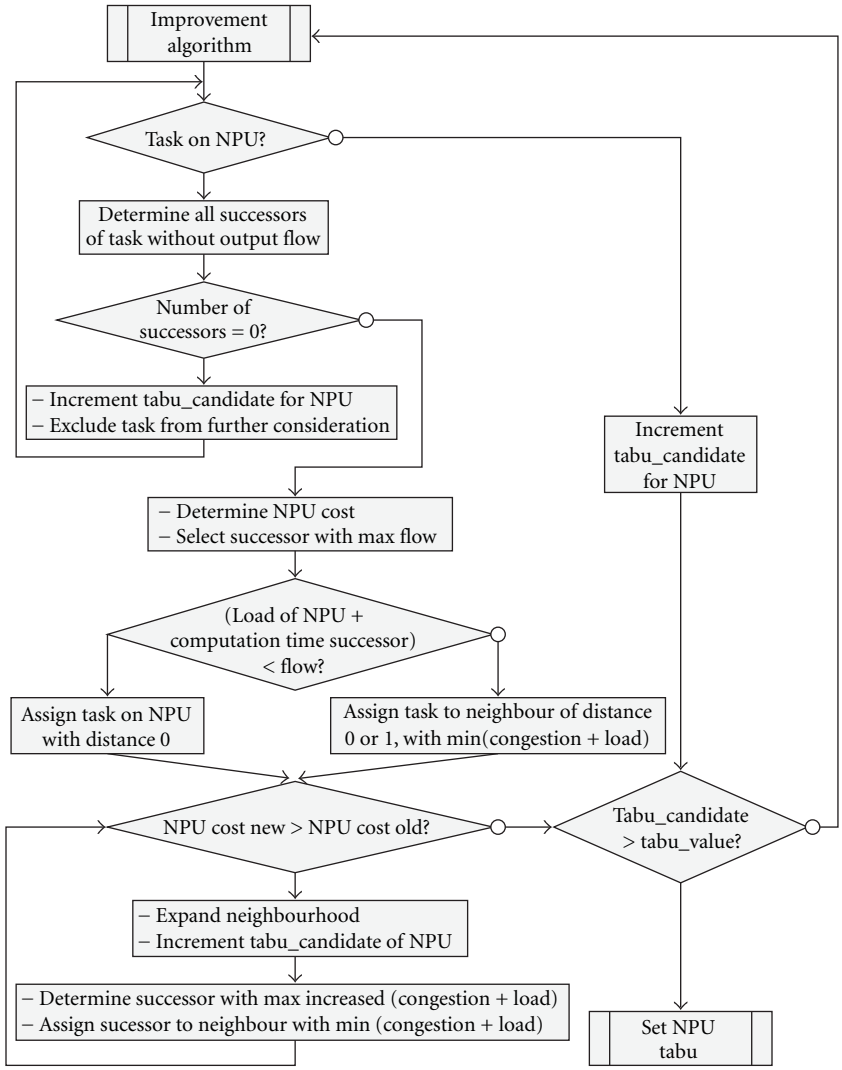


Figure 6: Program flow: improvement algorithm.

is within 10% (90.28%) of the best solution. The local and overall communication requirements (abstracted values for the objective function) and the computational load of the NPUs as computed by the three algorithms are also given.

5. Results for Larger Arrays and Task Sets

The previous results indicate the feasibility of the proposed decentralised placement heuristic. The general performance of the heuristic can only be evaluated by considering a larger range of array sizes and task counts. The exact enumeration algorithm cannot be used as a reference for array sizes above 3×3 and more than about 12 tasks because of the high complexity of $O(\text{NPU}^{\text{Task}})$. Instead, we use a simulated annealing algorithm to optimise the task mapping problem with global knowledge for larger arrays and higher numbers of tasks. These results can be compared to the results of our heuristic.

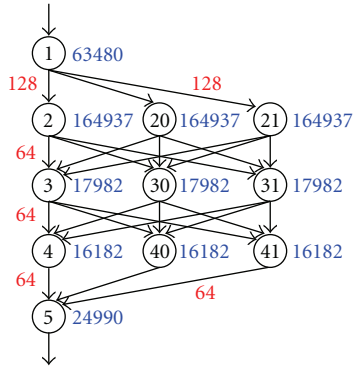
5.1. Experimental Settings and Data. To gain significant information on the behaviour of the algorithms, a large number of experiments must be made for different array sizes and task counts. A task graph generator was implemented to produce random task graphs. Each task graph is characterised by the number of nodes (tasks) it contains, the number of unconnected subgraphs (task groups or processes), and the specific values for the computational load of each task and the communication bandwidth of each edge (data communication between tasks). For our experiments the following parameters are varied.

(i) Array size: array sizes from 1×1 to 9×9, that is, from 1 to 81 NPUs.

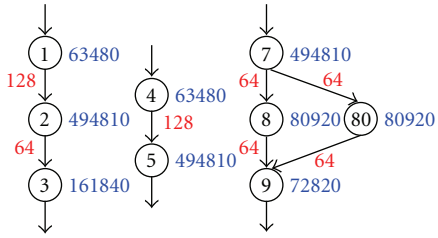
(ii) Task count: task sets with between 10 and 90 tasks.

(iii) Process count: values between 2 and 8 have been used.

The graph generator software produces a number of samples for each parameter combination, for example, 100 graphs

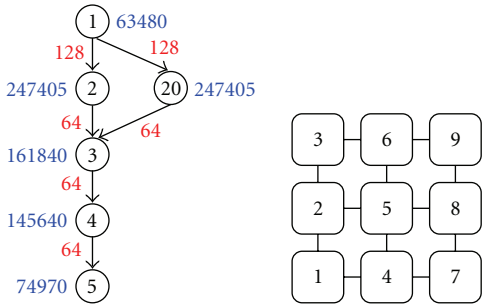


(a) Task graph 6



(b) Task graph 17

Figure 7: Examples of the task graphs used.

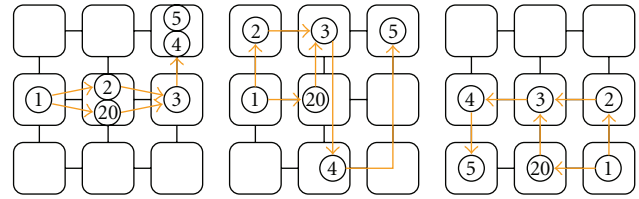


(a) Task graph 2 (b) Processor graph

Figure 8: Modelling example for an application and NoC processor array.

with 25 tasks, and 4 independent processes, with a randomly distributed number of tasks per process. Figure 10 shows one of the task graphs generated during the experiments. It contains 14 tasks arranged in 3 independent groups (processes) which are meant to run in parallel on the NPU array. Each task graph is then handed to the heuristic and the simulated annealing algorithm for placement. Additionally, a random placement is also generated. The resulting objective function values for all three obtained placements are saved as average, minimum, and maximum values over all samples for each parameter combination.

Simulated annealing is known as a good heuristic approach for problems with a largely unknown solution space structure and should produce reasonable reference results.



(a) Constructive (b) Improvement (c) Exact

Figure 9: Comparison of constructive, improvement and exact algorithm.

To get some general information about the design space, two considerations can be made. Firstly, we will assume that input and output tasks must be placed on a boundary NPU. For array sizes above 2×2, the number b of boundary NPUs grows linearly with the square root of the NPU count ($b = 4(\sqrt{n} - 1) = 4\sqrt{n} - 4$, $n \geq 4$, with n being the (square) number of NPUs), while the number i of internal NPUs grows linearly with the NPU count ($i = (\sqrt{n} - 2)^2 = n - 4\sqrt{n} + 4 = n - b$, $n \geq 4$), that is, the fraction of boundary NPUs $b/i = b/(n - b)$ shrinks. While this does not reduce the complexity class of the problem, it still reduces the number of valid mappings due to the fact that input and output tasks must be mapped to boundary NPUs. The number of valid mappings m_v is given by

$$m_v = \left(4(\sqrt{n} - 1)\right)^{2g} \cdot (\sqrt{n} - 2)^{2(t-2g)}, \tag{6}$$

where n is the number of NPUs, and t is the number of tasks, and assuming that each task group g must have at most one input and one output task (or one-task processes, this can be the same task, so $2g$ is an upper limit). The first term gives the number of boundary NPUs, while the second term gives the number of "inner" NPUs of the array. The break even point of b and i is between array sizes of 6×6 and 7×7 (36 and 49 NPUs). For 9×9 array, there are 32 boundary NPUs and 49 inner NPUs, so a large predominance of inner NPUs needs not to be considered.

Secondly, some information about the objective function values to be expected can be obtained by examining random mappings or by using the simulated annealing algorithm to search for worst case solutions. Figure 11 shows the values of random task mappings for 50 tasks and different array sizes, averaged over 1000 samples each (please note the logarithmic scale for the y-axis). The error bars give the range between the best and the worst mapping value found within the samples. It can be seen that larger arrays allow for more efficient mappings according to the objective function. Also, a saturation effect can be observed towards larger arrays.

5.2. Mapping Evaluation. The obtained data can be analysed and the quality of the heuristic mapping results can be rated in relation to the simulated annealing results. Figure 12

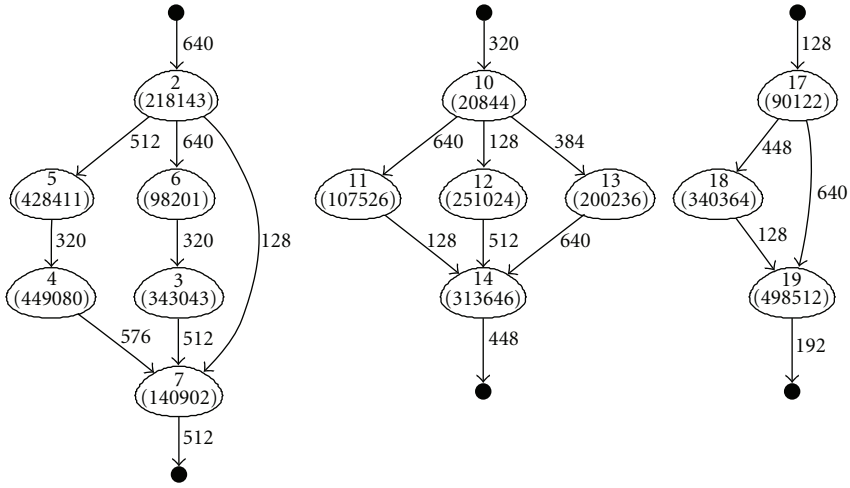


Figure 10: Example: generated task graph.

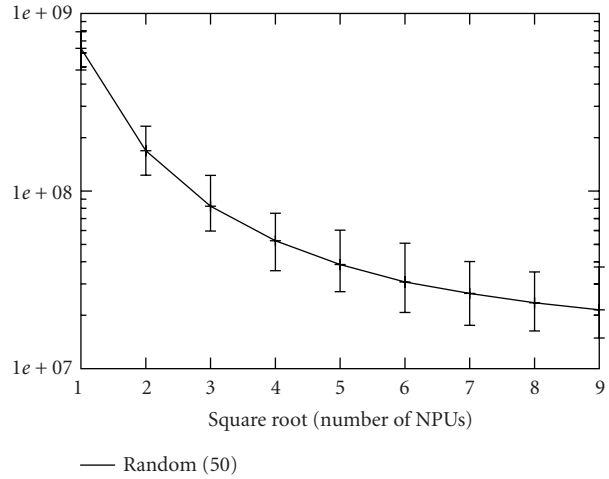


Figure 11: Objective function value space for 50 tasks and random mapping.

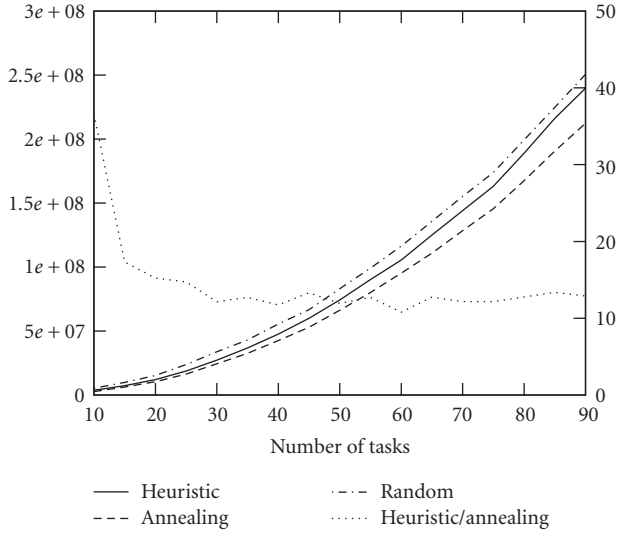


Figure 12: Objective function values (left y-axis) on a 3×3 NPU array and ratio in percent (right y-axis).

shows the objective function results for the heuristic and those for the simulated annealing algorithm for the same task graphs for a 3×3 array (4 processes); Figure 13 shows the same data for a 6×6 array (8 processes). The dotted line (values in %, right y-axis) gives the ratio between simulated annealing and the heuristic. It can be seen that the heuristic performs better for higher numbers of processes. For 8 processes, the heuristic delivers never more than 30% less well results than simulated annealing for arrays between size 3×3 to 5×5, while even keeping below 20% for arrays larger than that. For 4 processes, the heuristic results are never more than 45% worse than simulated annealing, but less than 35% in the great majority of examples. This is true over all data sets, that is, for all array sizes.

Figures 14 and 15 show the mapping development for a fixed task size of 30 and 60 tasks, respectively, and different array sizes (data shown for 8 processes). For the figures, the y-axis range is fixed. It can be seen that there is a diminishing tendency for saturation towards larger NPU arrays, which was already visible in the data for random placement (see Figure 11).

Looking at the relative difference between the heuristic and simulated annealing on the one hand and the random placement on the other hand, it can be seen that there is an distinct minimum in both results at array sizes specific to the task count considered. Figures 16 and 17 show this for 20 and 70 tasks respectively (data shown for 8 processes). The relative minimum for 20 tasks is at 4×4 arrays while it is at 49 and 64 NPUs for 70 tasks. It also becomes clear that the specificity of the minimum diminishes for higher task counts. More specific, while simulated annealing can get down to about 50% of the objective function values of random placement for 20 tasks, it can only accomplish little below 70% of it for 70 tasks. At the same time, the results for

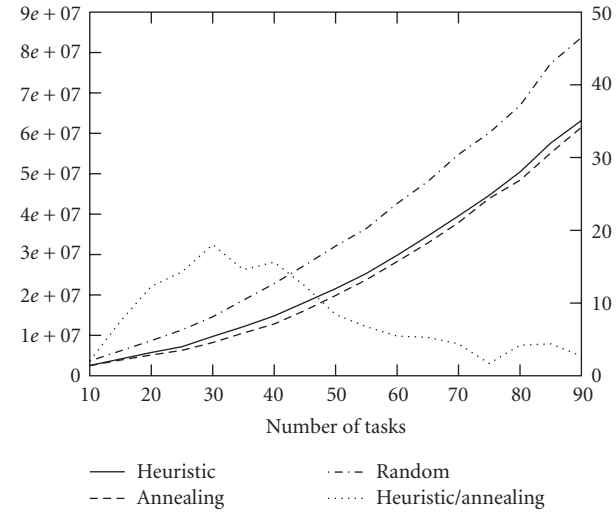


Figure 13: Objective function values (left y-axis)on a 6×6 NPU array and ratio in percent (right y-axis).

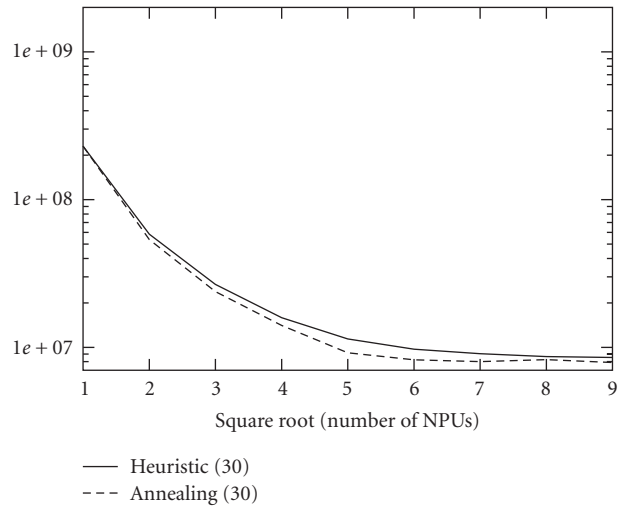


Figure 14: Objective function values for a fixed task count (30) and different array sizes.

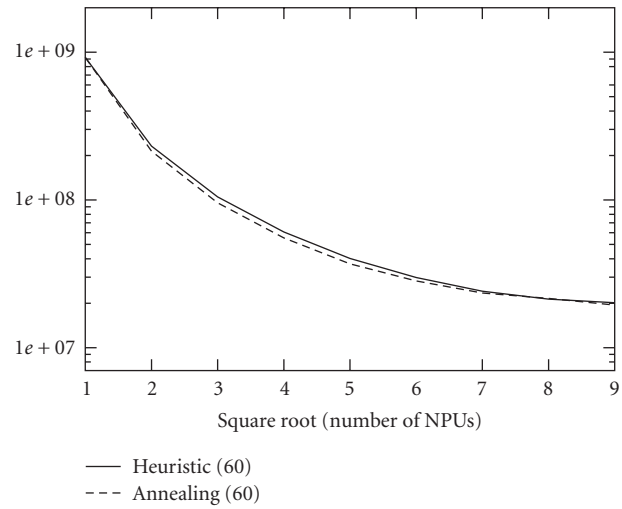


Figure 15: Objective function values for a fixed task count (60) and different array sizes.

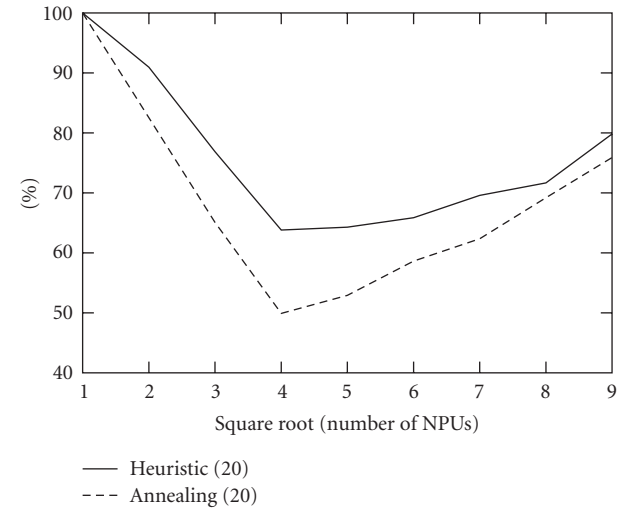


Figure 16: Relative quality of a fixed task count (20) and different array sizes.

the heuristic get closer to that of simulated annealing, already apparent in Figures 12 and 13. Table 6 gives an overview on the minima found for selected task counts. It can be seen that for higher task counts, the heuristic tends to require larger processor arrays than simulated annealing to accomplish its best results.

Finally, it is interesting to look at a specific placement of tasks produced by the heuristic and the simulated annealing algorithm, to see the basic differences. Figure 18 shows the task mapping of the task graph from Figure 10 on a 5×5 array as produced by the heuristic. The three processes are composed of tasks 2–7, 10–14, and 17–19, respectively, with the first and last tasks of each process being input and output tasks. It becomes obvious that the placement produced by the heuristic is limited by the initial distribution of the input and output tasks. In two cases, NPUs 5 and 24, two tasks share the same processor. Apart from this, all other tasks could be placed on their own NPU, thus distributing their workload evenly. The same holds for the simulated annealing result where no processor sharing occurs. It can be seen that the processes are better clustered by the simulated annealing algorithm, while the far-apart input and output tasks as initially placed by the heuristic disrupt a close clustering. Nevertheless, the overall result of the heuristic (OF = 4,616,314) is only 31% worse than that of simulated annealing (OF = 3,516,371) which is quite a good result in the light of the missing global information for the heuristic.

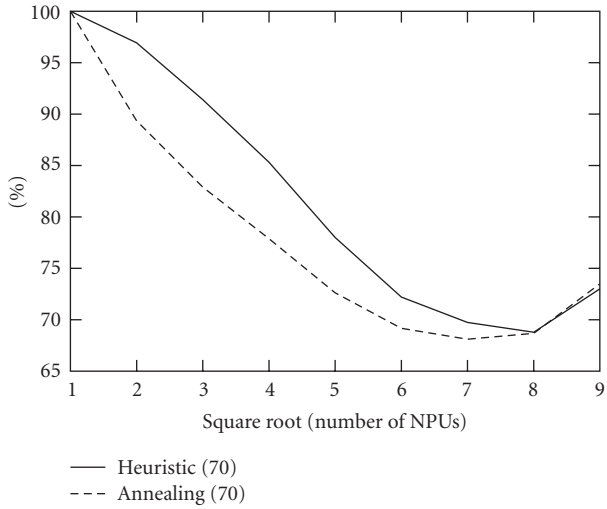


Figure 17: Relative quality of a fixed task count (70) and different array sizes.

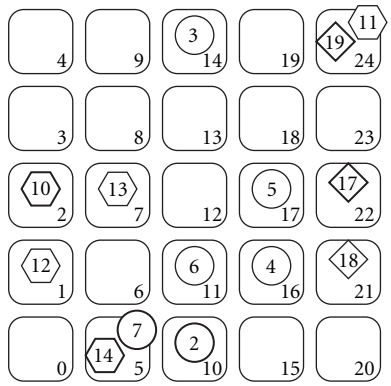


Figure 18: Heuristic placement of task graph from Figure 10 on a 5×5 array.

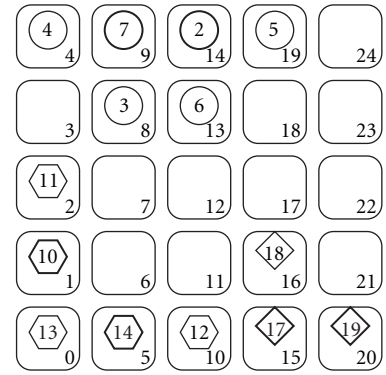


Figure 19: Annealing placement of task graph from Figure 10 on a 5×5 array.

6. Conclusion

This paper describes a distributed task mapping heuristic for homogeneous MPNoCs, derived from a mathematical model. It is based on an initial placement of tasks and a distributed improvement strategy locally implemented on the processing elements. Task sets belonging to different initial applications can be handled as well as tasks added during system operation. For the mapping improvement, only local information available at the affected NPUs and its close vicinity is used, thus avoiding additional communication overhead. Also, the low computational load of the algorithm itself makes its application very attractive.

Running the heuristic for a selected set of example applications shows the good results of the heuristic compared to the exact solution. The accuracy of the results is supported by a system simulation of the VHDL hardware model. For larger array sizes, the heuristic was compared to a simulated annealing algorithm and random placement. It can be seen from the obtained data, that for larger process counts not only do the achieved results of the heuristic come closer to those of simulated annealing—in fact, for large array sizes and high task counts they are even better in some cases—but also the difference of both towards the random placement results get much better. This means, the heuristic delivers increasingly better results for increasing process and task counts. Thus, the heuristic appears to be well suited for future challenges. In summary, the combination of the constructive and the distributed improvement algorithms in the final system appears as a promising decision eliminating many potential scaling problems.

The presented algorithm implementation is a first approach to the problem of efficiently using homogeneous multiprocessor NoC platforms with a large number of processors. Dynamic workloads pose a heavy problem on such systems, for example, because task migration costs will not be negligible any more and must be included into the optimisation algorithms. We believe that the answer to this challenge can only be a scalable solution (like that presented in the paper) which is mainly based on distributed algorithms, using only local information, like the one presented. There is a large design space waiting to be discovered, for example, looking at biologically -inspired algorithms that have proved to be very successful already in nature.

References

[1] L. Benini and D. Bertozzi, "Network-on-chip architectures and design methods," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 261–272, 2005.

[2] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 1, pp. 1–6, Munich, Germany, March 2006.

[3] T. D. Braun, H. J. Siegel, N. Beck, et al., "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," in *Proceedings of the 8th Heterogeneous Computing Workshop (HCW '99)*, pp. 15–29, San Juan, Puerto Rico, April 1999.

[4] E. Carvalho, N. Calazans, and F. Moraes, "Congestion-aware task mapping in NoC-based MPSoCs with dynamic workload," in *Proceedings of IEEE Computer Society Annual*

Symposium on VLSI (ISVLSI '07), pp. 459–460, Porto Alegre, Brazil, March 2007.

[5] J. Chakrapani and J. Skorin-Kapov, "Mapping tasks to processors to minimize communication time in a multiprocessor system," in *The Impact of Emerging Technologies of Computer Science and Operations Research*, pp. 45–64, Kluwer Academic Publishers, Boston, Mass, USA, 1995.

[6] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task allocation in distributed data processing," *Computer*, vol. 13, pp. 57–69, 1980.

[7] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, no. 6, pp. 50–56, 1982.

[8] F. Glover, M. Laguna, and R. Martí, "Fundamentals of scatter search and path relinking," *Control and Cybernetics*, vol. 29, no. 3, pp. 653–684, 2000.

[9] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: a scalable, communication-centric embedded system design paradigm," in *Proceedings of the 17th IEEE International Conference on VLSI Design*, pp. 845–851, Mumbai, India, January 2004.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 2003.

[11] F. S. Hillier and G. J. Lieberman, *Introduction to Operations Research*, McGraw-Hill, Boston, Mass, USA, 7th edition, 2001.

[12] B. Hong and V. K. Prasanna, "Performance optimization of a de-centralized task allocation protocol via bandwidth and buffer management," in *Proceedings of the 2nd International Workshop on Challenges of Large Applications in Distributed Environments (CLADE '04)*, pp. 108–117, Honolulu, Hawaii, USA, June 2004.

[13] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular NoC architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 4, pp. 551–562, 2005.

[14] J. A. Keane, A. J. Grant, and M. Q. Xu, "Comparing distributed memory and virtual shared memory parallel programming models," *Future Generation Computer Systems*, vol. 11, no. 2, pp. 233–243, 1995.

[15] F.-T. Lin and C.-C. Hsu, "Task assignment scheduling by simulated annealing," in *Proceedings of the 10th Conference on Computer and Communication Systems*, pp. 279–283, Hong Kong, September 1990.

[16] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384–1397, 1988.

[17] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner, "Time and energy efficient mapping of embedded applications onto NoCs," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '05)*, vol. 1, pp. 33–38, 2005.

[18] A. Ngouanga, G. Sassatelli, L. Torres, T. Gil, A. Soares, and A. Susin, "A contextual resources use: a proof of concept through the APACHES' platform," in *Proceedings of IEEE Design and Diagnostics of Electronic Circuits and Systems*, pp. 42–47, Prague, Czech Republic, April 2006.

[19] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 234–239, Munich, Germany, March 2005.

[20] J. M. Orduna, F. Silla, and J. Duato, "A new task mapping technique for communication-aware scheduling strategies," in *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW '01)*, pp. 349–354, 2001.

[21] K. Park, "A heuristic approach to task assignment optimization in distributed systems," in *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, pp. 1838–1842, Orlando, Fla, USA, October 1997.

[22] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, Morgan Kaufmann, San Francisco, Calif, USA, 2004.

[23] S. Sahni and T. Gonzalez, "P-complete approximation problems," *Journal of the Association for Computing Machinery*, vol. 23, no. 3, pp. 555–565, 1976.

[24] N. Saint-Jean, G. Sassatelli, P. Benoit, L. Torres, and M. Robert, "HS-Scale: a hardware-software scalable mpsoc architecture for embedded systems," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pp. 21–28, Porto Alegre, Brazil, March 2007.

[25] R. Varadarajan, "An efficient approximation algorithm for load balancing with resource migration in distributed systems," Tech. Rep., 1992, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6072.

[26] P. Yang and F. Catthoor, "Dynamic mapping and ordering tasks of embedded real-time systems on multiprocessor platforms," in *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES '04)*, pp. 167–181, 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 912301, 10 pages
doi:10.1155/2009/912301

Research Article

A System on a Programmable Chip Architecture for Data-Dependent Superimposed Training Channel Estimation

Fernando Martín del Campo,[1] René Cumplido,[1] Roberto Perez-Andrade,[1] and A. G. Orozco-Lugo[2]

[1] *Computer Science Department, National Institute of Astrophysics, Optics and Electronics, CP 72840, Puebla, Mexico*
[2] *Section of Communications, CINVESTAV-IPN, CP 07360, Mexico City, Mexico*

Correspondence should be addressed to Fernando Martín del Campo, fmartin@ccc.inaoep.mx

Received 25 December 2008; Accepted 25 May 2009

Recommended by Peter Zipf

Channel estimation in wireless communication systems is usually accomplished by inserting, along with the information, a series of known symbols, whose analysis is used to define the parameters of the filters that remove the distortion of the data. Nevertheless, a part of the available bandwidth has to be destined to these symbols. Until now, no alternative solution has demonstrated to be fully satisfying for commercial use, but one technique that looks promising is superimposed training (ST). This work describes a hybrid software-hardware FPGA implementation of a recent algorithm that belongs to the ST family, known as Data-dependent Superimposed Training (DDST), which does not need extra bandwidth for its training sequences (TS) as it adds them arithmetically to the data. DDST also adds a third sequence known as data-dependent sequence, that destroys the interference caused by the data over the TS. As DDST's computational burden is too high for the commercial processors used in mobile systems, a System on a Programmable Chip (SOPC) approach is used in order to solve the problem.

1. Introduction

The air is inherently noisy and its nature can contribute to the presence of different kinds of interference, as the one known as Intersymbol Interference or ISI, in which the energy of the message symbols is spread in such way that a part of each symbol overlaps with that of the neighboring symbols. The ISI can, in fact, make almost impossible to the detector inside the receiver to differentiate between a symbol and the spread energy of consecutive ones. Nevertheless, this channel can be modeled as a linear system, whose effects can be reverted in the receiver, if one knows its parameters with enough precision.

To obtain these parameters, the majority of the digital wireless communication systems use sequences of known symbols that are also called training sequences. These groups of symbols, after a certain analysis, allow the estimation of the communication channel. Once this has been performed, the original information can be extracted, using well-known mathematical formulas and DSP techniques for data recovery.

The most extended technique to integrate the training sequences to the information is known as *time-division multiplexed channel estimation or time-division multiplexed training (TDMT)*, where some of the transmission slots are used for the pilots or training symbols [1]. The performance of this approach is very high, but it has the disadvantage of needing part of the available bandwidth to accommodate the extra data. Even though several options have been proposed, none of them have demonstrated to be more feasible than the usual training.

One of the most promising techniques that has not yet been implemented physically is *Superimposed Training (ST)*, where the training sequence is arithmeticly added to the information, saving the necessity of more bandwidth at the expense of a little power loss on the information signal [2, 3].

The method named *Data-dependent Superimposed Training* goes beyond ST, adding another sequence (the data-dependent training sequence) to the information. When estimating the channel, what is analyzed is the training sequence so, from this point of view, the original data can

be considered additive noise that distorts the object of study. The data-dependent sequence cancels the contribution of the input signal at the frequency bins at which the training sequence has energy, improving the channel estimates over ST [4].

The problem with DDST, talking about its possible implementation, is its high computational complexity, that renders the commercial mobile and low power demanding processors useless for this purpose, at least taking into account the performance demanded by the systems in which it could be used. Even though simulations with the DDST method show a performance that can compete with the TDMT [4–6], the inherent computational burden of the method has made, at the moment, impossible to implement it in commercial digital communication systems, due to the time, power, and space constraints imposed by the devices used in this field. Until now, the only alternative solution has been the use of a DSP architecture, with both fixed point and simple precision floating point (32 bits) arithmetic. Resulting speeds from these architectures (specially the floating point one) are in the order of kHz, so, even when they are useful for error comparison, it is impossible to use them in commercial systems.

This work will describe a combined software/hardware implementation of the DDST algorithm using an SOPC approach, highlighting the most challenging issues that have arisen, and the way in which they have been tackled. It is true that obtaining a fast and functional DDST solution is a highly complex task, but the promise of a larger available bandwidth for the information is very attractive. Moreover, the method, seen as a set of individual steps, presents challenges for which an optimal solution has not yet been found, so the fact of, at least getting closer to them, can be of use for other open problems.

From an academic point of view, the DDST implementation has a special interest, as neither a full software nor a hardware approach seems to be a satisfying solution. On the one hand, a software alternative is, at this moment, unfeasible, due to the time it would require for obtaining a channel estimate and then using such estimate for the equalization. On the other hand, a hardware architecture, for example, using an FPGA thinking toward the construction of an ASIC, presents several problems, caused by the enormous amount of data that has to be operated constantly, the high degree of data dependencies between stages of the process (which make very difficult to use techniques as parallel processing and pipelining) and the complex control required by some of the mathematical operations that have to be performed. It is true that some of the top branch FPGAs available form Altera and Xilinx can accommodate a full hardware architecture, and in fact a solution like this one is the final goal of a DDST receptor, but actually, two problems arise from this option: first, the transformation of the FPGA prototype into an ASIC solution can be far from optimal, as it would be neither cheap nor low power consuming, overall because the architecture presents several DSP like structures, that usually do not map very well to ASIC implementations [7]. The second problem with this solution is the highly complex control, that is, necessary to process the amount of data that a digital receptor usually must handle. As this work presents the first implementation of the DDST channel estimation algorithm on an FPGA, a hybrid software/hardware implementation presents the advantage of a simple control through a series of software coded instruction and the power of dedicated hardware coprocessors to perform the most time and resources demanding tasks of the DDST receiver stages for the obtaining of the estimate.

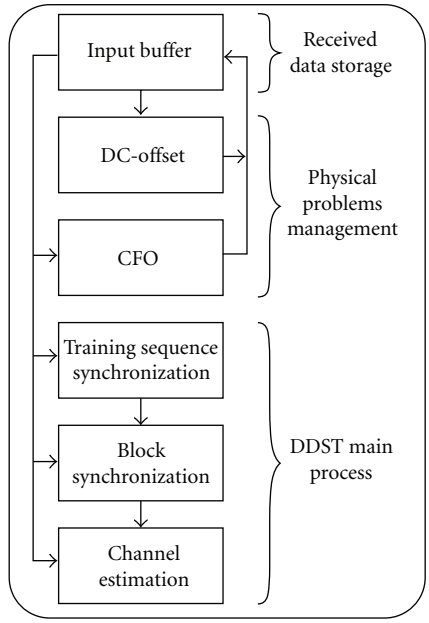


Figure 1: DDST general block diagram.

2. DDST Algorithm Review

Figure 1 shows a general block diagram of a digital communication receiver based on DDST.

Before describing each block, it is important to note that they cannot be executed in a parallel fashion, that is, to start the process of each block, it is absolutely necessary that all the previous stages have already finished their own function. The lines with the arrow markers indicate from where each block receives its input and to where it feeds its output.

It is also important to mention that the DC-offset, along with the two synchronization steps, and the channel estimation itself, exploit the *cyclostationarity,* that is, induced in the transmitted signal, when superimposed training is employed [2, 3].

This work does not focus on the mathematical meaning of the formulas used to solve the different steps of the DDST approach, but in the computational burden that they present and in the procedure followed to implement them in the system.

2.1. Input Buffer. To begin with the steps of the algorithm, it is necessary to store the input data samples as they are being received, because it is not possible to correct them "on the fly." As shown in Figure 1, the input data samples are

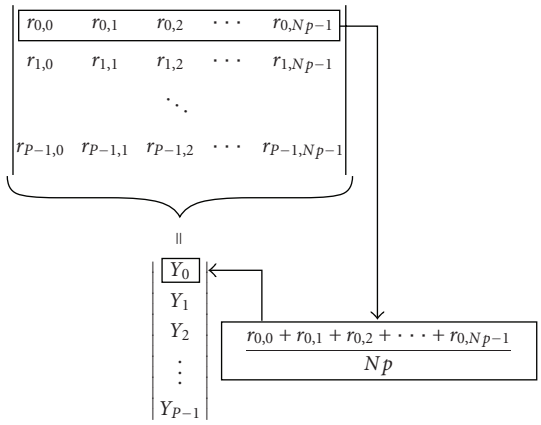


Figure 2: Arithmetic mean from the input buffer.

corrected several times as the different stages of the DDST channel estimation are fulfilled.

Before going further into the process, it is fair to mention that all data samples are complex valued quantities, so all operations performed in the following steps involve the computation of both a real and an imaginary part of several numbers.

2.2. DC-Offset Estimation and Correction. Practical systems commonly face a physical problem resulting from the building techniques used; their output, seen as voltage levels, presents an unwanted constant value, that is, added to the expected signal. Even though this value is almost always very small, it must be considered as the method works with first order statistics [2].

This block involves the reshaping of a vector formed by an N size subset of the original input data into a matrix of size $\|P \times (N/P)\|$, whose rows are then summed to form a vector of length P. Each element of the vector is then multiplied by $1/(N/P)$ so, in fact, each element of the output vector corresponds to the arithmetic mean of each of the rows of the matrix mentioned above. This process is shown in Figure 2.

Once this vector has been obtained, the process reaches an iterative phase that involves matrix multiplications, norm of a vector (the square root of the sum of the squares of the real and imaginary parts of each element of the vector), several other multiplications, and a division. All of these operations have a high computational complexity, where the square root and the matrix multiplication are the more challenging. Once the DC-offset has been obtained, it is removed from the input data by simply subtracting it from each input element.

2.3. Carry Frequency Offset (CFO) Estimation and Correction. Due to the lack of perfect oscillators and because of Doppler shifts, receivers in practical pass-band systems always experiment a carry frequency offset [8]. Among the mathematical operations found in a DDST-based digital communications receiver, CFO estimation has the more complex of all, both in time and resource usage. This block requires several summations, Fast Fourier Transforms and vector norms, but even these operations result insignificant when compared with the real problem of this stage: to obtain a CFO estimate, a simulation run, for example, needed to calculate 36864 complex exponentials. If they are solved using the Euler's Formula, 73728 trigonometric operations have to be performed. As the CFO estimation is an iterative process, the complexity is not the only problem, because low data resolutions lead to fast growing errors, that in the end can result in a very inaccurate estimate. Once this process is complete, the CFO is removed from the input data by multiplying each input sample by one complex exponential term.

2.4. Training Sequence Synchronization Estimation. In practical applications, it is usually impossible to suppose a perfect synchronization between the transmitter and the receiver at the training sequence level, so channel estimation must consider this issue.

When there is no perfect synchronization, the estimate is just a circular shifted version of the real channel estimation that would have been obtained under ideal conditions. Using the cyclostationarity of the signal, one of the possible permutations in the circular array is equal to the estimate supposing perfect synchronization, so the problem is reduced to obtain the knowledge of the correct permutation. Mathematical operations in this stage are almost identical to those performed for the dc-offset estimation [5].

2.5. Block Synchronization Estimation. As with the training sequence estimation, block synchronization is also based on the particular structure of the channel output's cyclic mean vector, and can be achieved even in the presence of a DC-offset. Due to its special characteristics, in DDST it is not enough to locate the start of a training sequence period, because it is also necessary to find the start of each received block. Only the vector encompassing a full DDST block will provide a cyclostationary mean vector independent from the data sequence, with a reduced "data" noise compared to the rest of the estimates. This procedure is achieved through a specific cost function, that will give a minimum value only with the right version of the cyclostationary mean vector [5]. Even though this block is, in concept, very different from the Training Sequence Synchronization estimation, the necessary operations for the Block Synchronization Estimation also include matrix multiplications, norm of a vector, and other multiplications.

2.6. Channel Estimation. Once the two synchronization estimations have been obtained, the channel estimation stage can be tackled with very similar operations to those that have already been used. In fact, this easy step only needs a vector reshape, the mentioned vector obtained from the arithmetic mean of that reshaped matrix, and a matrix multiplication. At the end, what is obtained is a vector whose complex elements correspond to the values of each tap of the estimated channel.

Table 1: Computational complexity of the stages of the DDST channel estimation algorithm.

Stage	Complexity
Input buffer	$O(1)$
DC offset	$O(n^2)$
Carry frequency offset	$O(n)$
Training sequence synchronization	$O(n^2)$
Block synchronization	$O(n^2)$
Channel estimation	$O(n^2)$

2.7. Computational Complexity Review. To show the problems generated by each of the different stages of the DDST channel estimation algorithm, from a different point of view, Table 1 enlist their computational complexity. Nevertheless, this approach is deceptive, as it supposes that the atomic operations in the process consume the same amount of time. For example, a matrix multiplication requires two nested cycles, but the basic operation of the multiplication presents an $O(n)$ complexity, so, in fact, the full operation presents an $O(n^3)$ complexity. Another example is the square root operation, which presents an $M(n)$ complexity, or the FFT calculation, with $O(n \log_2 n)$. Moreover, parameters as P and N from the algorithm are variable, so it is very difficult to give an idea of the real magnitude of the problem in terms of computational complexity. Section 6 tackles this issue by giving the results from the implementations in terms of consumed time and necessary clock cycles to fulfill the operations of each stage.

3. DDST and Its Hardware Implementation

As DDST is an experimental method for channel estimation, there is not any commercial implementation nor a full prototype receiver for it. Until now, the great majority of the performed tests correspond to software simulations, in which the main purpose is to compare the performance of the superimposed methods with the one of techniques like the TDMT, with respect to errors and noise tolerance.

At this point, it can be inferred that the difficulty for implementing the algorithm in hardware is caused by two main reasons: the complexity of the operations (square root and trigonometric functions are far from an optimal hardware solution), and the amount of data that is used, transformed, and updated constantly. The first of these issues leads to the generation of huge hardware components, that usually need several multipliers, units that are scarce in mid-range FPGAs. The second one requires a very complex control unit and the need of a high amount of memory accesses.

Moreover, the huge amount of data dependencies (operations that require several previous results from past stages) makes it very difficult to use techniques such as parallel processing and pipeline implementation. Even in those few cases when it is possible to identify those stages of the process in which either the parallel operations or the pipelining is feasible, their performance gain versus that of a software only implementation can be small, and they usually require a considerable FPGA area, due to the amount of information that has to be operated in a concurrent fashion.

This implementation tackles the problem by using a hybrid software/hardware approach. A set of C language programs running over an *NIOS II soft processor* spare the need of a complex control, while dedicated hardware coprocessors perform the most time and resource demanding operations (like the FFTs), also allowing operations with nonstandard data lengths (e.g., 48 bits) that are hidden to the C programs of the system, making it easier to control, modify, and extend the software section of the SOPC. In spite of the FPGA advantages, it is undeniable that they cannot compete in many fields with the general purpose microprocessors computer systems (like the PCs), due to the huge amount of available memory and high speed processing of these last ones. Nevertheless, the PCs are also outperformed in applications where parallel or pipelined processing of large amounts of data is required, or in those where price, size, and power consumption constraints are very strict. These are the reasons why the proposed solution tries to take advantage from both approaches generating a specific purpose SOPC as a proof-of-concept solution for the DDST problem.

4. Hardware Architecture

Figure 3 depicts the DDST hardware architecture. It has been designed to run in an Altera Stratix II FPGA as a system controlled by an NIOS II. As it can be seen, the architecture resembles that of a common computer system, with the difference that it presents several dedicated memories for fast data fetching and processing, and a set of special hardware accelerators that interact directly with the rest of the system. The processor only passes them certain parameters and activates them (through the use of their slave ports), but they execute its processing in a stand alone fashion. This means that they can read and write, using the master ports, all the memories in the system (although they almost always interact with the dedicated ones) and, while one of them is working, neither the processor nor do the other accelerators need to be interrupted. The control necessary to avoid the resource competition is performed by the software section of the system.

4.1. NIOS II Soft-Core Embedded Processor. The NIOS II from Altera is a soft-core microprocessor that, for the DDST implementation, presents several advantages, like its low power consumption and its small required FPGA area, along with the ease to interact with custom hardware accelerator modules.

Its main advantage over similar processors is its flexibility and configurability capacity, that allows not only to use a great variety of included peripherals, but also to create custom peripherals that can then be easily interfaced to the processor. NIOS II flexibility allows to even add new

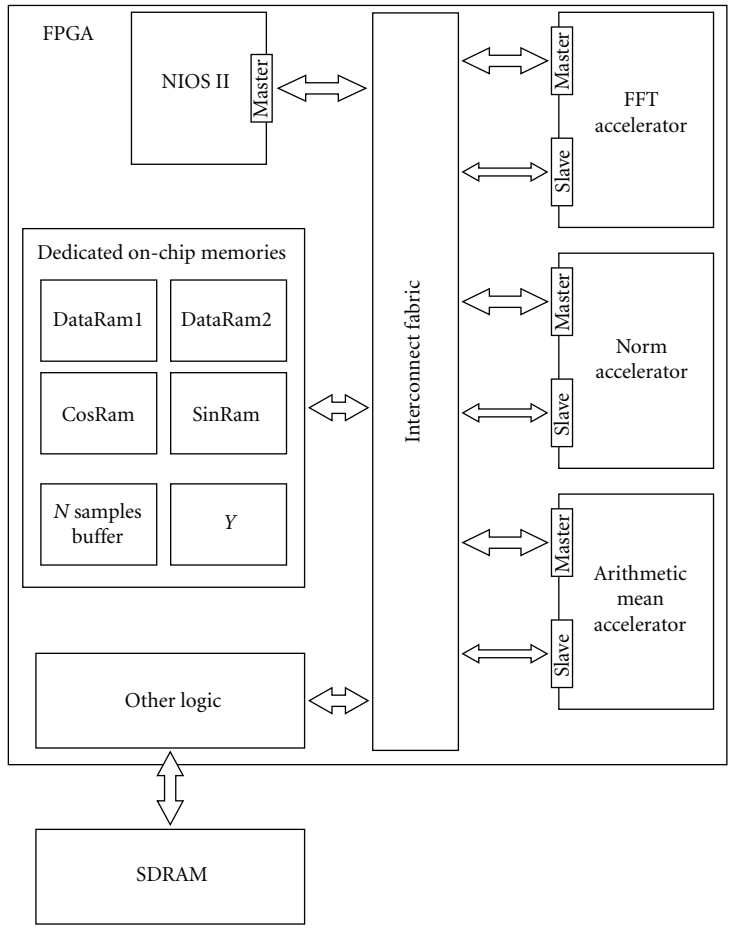


Figure 3: Hardware architecture of the system.

instructions to the instruction set. This is performed by hardware modules that are connected directly to the NIOS II ALU. Both the custom peripherals and the custom instructions can be used through a relatively transparent interface written in C language, or even in assembler.

4.2. Dedicated On-Chip Memories. The on-chip memories are structures that allow the transparent management and use of the memory blocks contained inside the FPGAs. They have the smallest latency (1 cycle) of all the available memories in the Altera boards. Low latency reduces the number of cycles needed to obtain, operate, and store a datum or a group of them. Fixed latency means that the system does not need to access the memory sequentially to achieve the highest throughput.

In the Altera NIOS II IDE, on-chip memories can be used as if they were part of the general data memory by just declaring a pointer to its assigned address, inside a typical C language program, that then will be compiled for the soft processor architecture. Dedicated memories DataRAM1, DataRAM2, CosRAM, and SinRAM are used by the FFT accelerator, while the N Samples Buffer and Y are accessed by the other two coprocessors. The NIOS II can access all of them.

5. Paradigm of the Architecture

A pure VHDL or Verilog implementation of the DDST architecture results in a very complex control, and in a logic that cannot fit on the majority of FPGAs without sacrificing speed for resources usage. The alternative proposed in this work is an SOPC that runs a series of C programs, but leaves the most *computer intensive* or *memory demanding* operations to special hardware accelerators.

Contrary to the majority of *Systems on a Chip* and common computer systems, Altera does not use a conventional bus scheme. The systems built on the manufacturer's programmable hardware feature a switch interconnect fabric (Avalon) which bypasses bus contention in most applications and gives a higher-performance pipe between processors and peripherals. This improves the DDST implementation execution time and prevents the necessity of extra control in both the software and hardware parts of the SOPC. The Avalon is a nonblocking interface, created by the *SOPC Builder* tool, that interconnects all the components in the system and permits multiple simultaneous master-slave transactions, while still requiring minimal FPGA resources. It replaces the traditional *shared bus* of usual electronic systems.

There are two kinds of ports that can access or be accessed by the Avalon: the slave and the master. Slaves are used to

receive signals from other components of the system so they can be controlled. Meanwhile, masters can manage other components and perform actions like doing a memory read or write. In *SOPC builder*, a master can read or write up to 1024 bits on each memory access and not only can they communicate with on-chip memories, but also with any other memory device in the system. All that is needed is the base address of such memory and the existence of a controller for this last one. Those controllers are usually provided by Altera, like in the case of the SDRAM.

The three accelerators in Figure 3 (that will be explained in the following sections) are activated by the processor through their slave ports. They can perform memory accesses using their master ports and, after their work is finished, they indicate it with a signal that can be read from the processor using the slave port again. Their operation is hidden to the user by a series of C functions that read from and write to the slave port. The result of each of the accelerator processes is stored in the on-chip memories *DataRAM2*, *Y*, and *RK*.

At the moment, the only device outside the FPGA that is used is the SDRAM, but to add ADCs to directly digitize the received data is being studied.

5.1. FFT Accelerator. The FFT coprocessor is based on an original design of Altera that uses a tool named C2H to convert C language instructions to hardware elements directly. There are other options (to use an IP core or a custom hardware design) that can achieve a better performance than this solution, but the C2H solution shows an interesting capability of the system being built: if the DDST algorithm is modified to have a better performance or to reduce the computational complexity of a certain step, it is possible to implement such change in the architecture by just modifying some lines of code in a C program, instead of redesigning a full hardware accelerator. As it is, the FFT coprocessor executes its function with acceptable speed and area consumption. This implementation uses a decimation in time FFT algorithm known as decimation in time Cooley-Turkey. Results are obtained through the use of a technique of ping-pong buffering, in which two memories are used to store the data. At the beginning, the source data are stored in the first memory, and the processed result in the other one. For the next iteration, the source data will be read from the first memory, that is, the results of the past stage are now the input of the next one, and the output will be stored in the first memory. The process is repeated until the FFT is completed.

The Nios II C2H Compiler maps ANSI C constructs directly to RTL. For example, an if construct will be mapped directly to a multiplexer, and a multiplication will be mapped to one of the available embedded multipliers.

The FFT module is different from the other hardware accelerators programmed directly with VHDL, and has three main advantages: first, modifications over the code are very easy to perform, and even a programmer that does not know anything about the system can perform them. Second, the complex control that the algorithm requires is automatically generated by the tool. This is important overall taking into account that the optimizations of the method require several accesses to four different memories (two for the real part of the data and another two for the imaginary one) in the ping-pong buffering stage. Finally, changes in the original DDST algorithm can be transformed into C code easily, reducing drastically the time required to update this block of the system.

The implementation of this kind of FFT algorithm consumes significant FPGA memory resources, because of the amount of data that have to be temporarily stored for the ping-pong method. Moreover, two extra memories are necessary to store the twiddle factors, that are basically two sets of prestored values from sine and cosine calculations, that are used to combine successive FFT results. Consequently, to implement two FFT modules would consume resources that could be used for other operations. Nonetheless, the structure of the FFT of $2N$ points contains, implicitly, the result of an N points one. The even elements of the $2N$ FFT contain all the results of the N points version. In this way, it can be said that the odd elements of the big FFT only provide information, that is, useless for the following calculations. Obviously, this approach takes twice the time that would be used by a simple N point FFT module, but as it is, even in the $2N$ case, this result is preferred to the extra FPGA area that the FFT modules would consume for a faster implementation.

5.2. Arithmetic Mean Accelerator. In the low complexity blocks of the DDST SOPC, the most critical operations to be performed are the matrices multiplications (basically $C^{-1}Y$). Nevertheless, the hardware acceleration of such multiplications is well studied, and the basic strategy is always the same: to execute as many operations in parallel as possible. This approach gives good performance, but it consumes many resources, in logic elements, embedded multipliers, and routing. As many other operations in the receiver require the embedded multipliers, and because of the space restrictions, it was decided not to accelerate this operation, but to concentrate the efforts on the other steps necessary for this kind of block (dc-offset, TSS, block synchronization, and channel estimation). An operation, that is, repeated several times in these blocks is the redimension and average of the sample vector (which lead to the obtaining of Y). This requires many memory accesses, several additions, and P multiplications. As mentioned in Section 2, the reshape operation is performed constantly along the DDST execution, and then followed by the arithmetic mean of the rows of the resulting matrix. At the end, the result is a vector of P elements, where P is the length of the training sequence.

This coprocessor, whose block diagram is shown in Figure 4, performs the summations over the vector to reshape in parallel, sending the result to a dedicated memory that corresponds to the vector of P elements. In this way the step of the reshaping can be bypassed and the execution time is reduced considerably.

As Np is a parameter that has to be known before computing, then its inverse can also be previously calculated, and the P necessary divisions can be performed as multiplications by $1/Np$, accelerating the process. In fact, many of

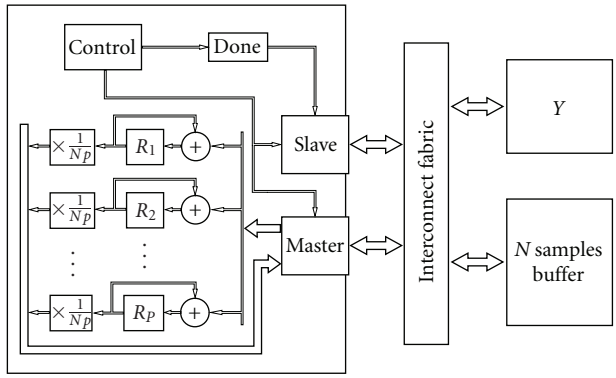


Figure 4: Arithmetic mean hardware accelerator.

the parameters used along the algorithm are also previously known, so it is possible to reduce the time required by several operations.

Summarizing, this coprocessor reads directly from the *N samples buffer* dedicated on-chip memory, **P** data, each one of 32 bits, accumulating their respective values to **P** registers of 32 bits width. At the end of the process, they are multiplied by $1/(N/P)$, so now they contain the arithmetic means of the rows from the reshaped matrix. Finally, the results are stored in another on-chip memory.

As it will be seen in the results section, this accelerator outperforms a software only version by more than 30 times, giving also a smaller error as it works with higher resolutions, during the multiplication stage.

5.3. Norm or Magnitude Accelerator. There are several steps of the algorithm in which it is necessary to work only with the magnitude of the complex elements of a vector. The high complexity of these operations comes not from the two multiplications to obtain the squares of the real and imaginary parts of a complex number, but from the necessity to perform a square root. Sometimes, as in the DC-offset estimation, it is possible to work with the square norm of the complex samples, but this is not the case in steps like the CFO estimation.

The norm of a complex number $a + bi$ is calculated as in (1) and it represents the magnitude of that number,

$$\sqrt{a^2 + b^2}. \tag{1}$$

The computational burden of this operation is high, as it does not only need to multiply the real and imaginary part by themselves, but also to obtain a square root. This last operation is difficult to implement with the required speed both in software and hardware, and the algorithms used for this computation are iterative, involving the use of multiplications, substractions, and comparisons.

This problem is solved by using an accelerator that has several advantages over the software-only version: it fetches both the real and the imaginary part of the FFT elements each time it performs a read operation; it works with 64 bits arithmetic, so there is no loss in the accuracy of the result. In addition, it accumulates the calculated magnitudes as it works, so at the end of the process this section of the system will have the summation of all of the results, a parameter needed for the iterative part of the CFO block. Finally, it is possible to assign an *"offset"* so, for example, the module can obtain only the norm of the complex samples in positions $0, 4, 8, \ldots$, and so forth, and not from every sample in the input vector. Figure 5 shows the block diagram of the accelerator. The operation of the square root block will be explained in the following section. The magnitude accelerator calculates all the norms of the complex samples in a vector V of n elements (as shown in (2)),

$$\sqrt{{v_r}^2(k) + {v_i}^2(k)}, \tag{2}$$

with $V = \{v_r(k) + v_i(k) * i \in \mathbb{C} \mid \forall k \text{ from } 0 \text{ to } n-1\}$.

As it can be seen, it is possible to send to the N Samples Buffer the result of each of the magnitudes as they are obtained, or their total summation. These decisions are fed to the module by the software program, along with the address of the memory and the amount of complex numbers to process (e.g., 1024 numbers resulting from the FFT).

5.3.1. Square Root Hardware Submodule. The square root is solved by a submodule with an operation based on the nonrestoring algorithm implementation, like the one of [9], but with two main differences: first, the 8 more significant bits of the root are obtained from a look-up table. This could be considered an approximated root, that then can be *adjusted*. For example, the square root of 5 is ≈2.236. A value of 2 would be obtained from the look-up table, so only the decimal part of the result would have to be calculated. This increases the speed of the coprocessor drasticly while still using very little FPGA area. Second, each iteration calculates, in parallel, 4 bits of the root, and not only one. This system is depicted in Figure 6.

The approximated root is obtained from the look-up table using as index the most significant bits of the radicand. Then, this value is appended to a set of possible roots that are squared and compared to the original radicand. A comparator tree evaluates all the results and decides which of the possible roots gave the smallest error. This value is then updated as the new approximated root and the next four bits are calculated. With each iteration, the approximated root of the possible set of roots grows four bits, until it reaches the least significant bit.

Another advantage of the coprocessor is that it stops its operation as soon as it finds an exact root of an introduced number, so not all the entries take the same amount of cycles to be calculated. For example, if we try to find the root of 14.0625, the process will stop as soon as it realizes that 3.75 is its exact square root (on the first iteration), even if the original number is represented as a 64 bits array, that usually requires 6 iterations for full resolution or 5 iterations for a maximum error of $\approx 7.15 \times 10^{-7}$.

It is important to consider the bit length of the look-up table memory. As more bits are added to this structure, the number of iterations necessary to have the best square root calculation will decrease. Nonetheless, as this length grows, the necessary size of the memory to store it also increases

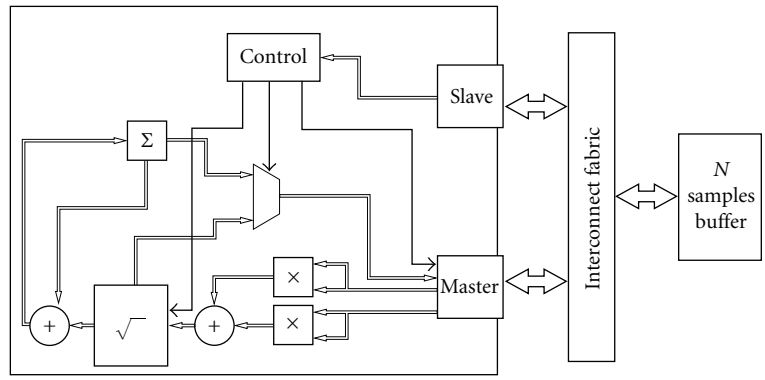


Figure 5: Magnitude hardware accelerator module.

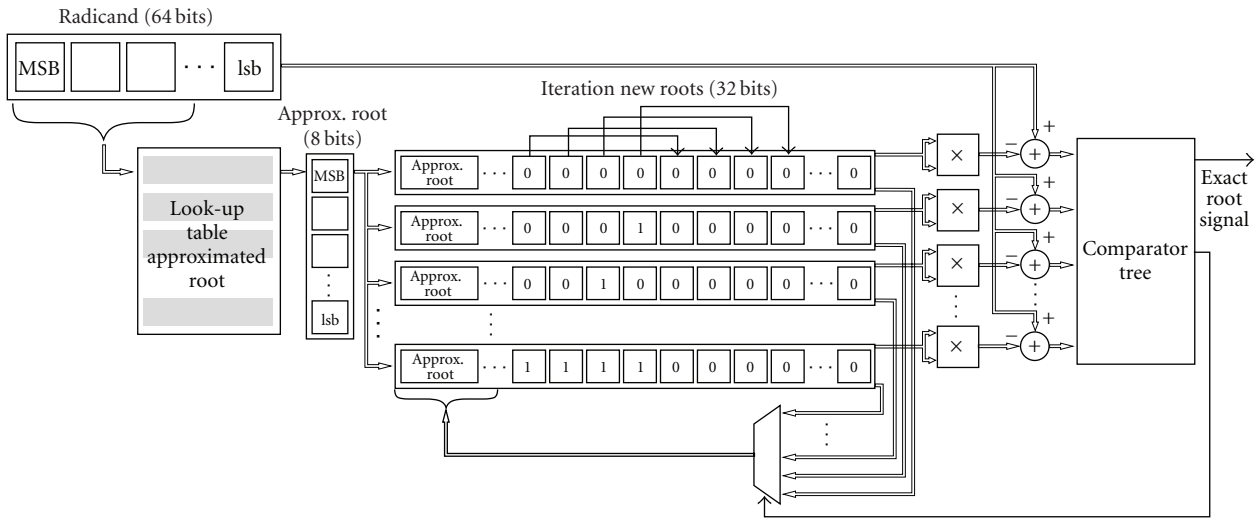


Figure 6: Square root submodule.

significantly, to the point that it is impossible to implement it using the FPGA memory blocks and even the external memories available in the board, like the SDRAM. The size of the look-up table involves a tradeoff, that is, particularly important in the case of architectures based on FPGAs.

6. Results

The hybrid architecture was compared against an optimized software-only version of the DDST algorithm implementation. The system for this nonhardware implementation is similar to the accelerated version, but it does not have the coprocessors or the dedicated on-chip memories. The reason for this decision is that a commercial implementation of the algorithm would run in a microprocessor built for mobile devices, with characteristics very similar to those of the NIOS II.

Both systems were tested with a set of 3765 complex data, with 32 bits for the real and imaginary part, respectively. In fact, lower resolutions (e.g., 16 bits) degrade the performance so drasticly that the obtained results are far too different from the expected ones (obtained from a Matlab simulation using double precision floating data). Execution time was measured using the Altera module *performance counter*, that physically times a group of code lines and outputs both seconds consumed and cycles taken for the operation to finish.

Table 2: Synthesis summary.

Implementation	Software	Hardware
Max. frequency	238.72 MHz	238.72 MHz
Space (total)	29%	68%
ALUTs	15%	60%
Registers	13%	24%
DSP blocks	10%	100%

Table 2 shows an abstract of the synthesis report for the software-only and hybrid systems. Space percentages refer to an estimate that the synthesizer makes according to the total available resources. The equal frequencies are expected as all the coprocessors would run at higher speeds than the NIOS II if working in a standalone way. When functioning together, the slowest device (the NIOS II microprocessor) determines

Table 3: Comparisons between software-only and hardware optimized operations.

Operation	Implementation				Performance increment
	Software only		Hardware accelerated		
	Cycles	Time [ms]	Cycles	Time [ms]	
Vector reshape and arithmetic mean	74824	0.75	2238	0.02	37.5x
1024 points FFT	1591929	15.92	56743	0.57	27.92x
Norm of all the output data from the FFT	3144061	31.44	138511	1.39	22.61x
CFO iterative process	354248859	3.54249	319750003	3.1975	1.1x

the maximum performance of both designs. Because of the large amount of performed multiplications along the algorithm, it can be seen from the table that the total of available DSP blocks were used by the hardware accelerated implementation.

On the other hand, Table 3 reports the time and cycles taken to complete some specific operations in the software program and its hybrid architecture counterpart.

All the arithmetic and manipulation operations of the accelerated SOPC outperform their software-only version, not only in speed, but also in precision, thanks to the non standard data lengths that are used in the intermediate results. This cannot be done, with such efficiency, in a common program, due to the fixed data lengths that have to be used. For example, a series of 14 bits wide data have to be stored in 16 bits wide variables, and their multiplication in variables of 32 bits, unless a data resolution loss can be tolerated. On the other hand, the hardware solution can take registers of 14 bits, and store the products in 28 bits wide structures, both if they are send to general purpose memories or to registers inside the FPGA. If the fact that the input data of the system are 32 bits wide is considered, this characteristic gets more importance, as a 64 bit resolution in the software-only program is more difficult to use, and it would be a necessity from the first performed multiplication.

As it can be seen in Table 3, the only operation in the system that still needs to be optimized is the CFO iterative section. This is expected as both the hybrid implementation and the software-only one use the $\sin f$ and $\cos f$ functions from the Altera version of the *math.h* library, that calculates the sine and cosine functions of simple precision floating point inputs. To find a way to accelerate these sine and cosine calculations, used for the complex exponential operations, the use of look-up tables and a CORDIC generator [10], are being tested. These experiments consider the tradeoffs between precision (so the final estimate has enough accuracy), speed (for a solution that can equal the performance of the other coprocessors), and occupied FPGA area.

7. Conclusions

An alternative solution that uses both a hardware and a software approach was developed to allow the implementation of a digital communications receiver based on the Data-dependent Superimposed Training algorithm for channel estimation. It was shown that it is possible to analyze a software-only code to detect the critical sections that can be translated into faster and more accurate hardware coprocessors, which can both be managed by the central microprocessor and operate independently, accessing the memories in the system without the necessity of interrupting the other components. The problem of the complex exponentials has yet to be solved. As a solution using common mathematical series is not suitable for FPGAs, a cordic generator and a hybrid approach using small look-up tables is being analyzed. In addition, to improve the performance of the whole system, the use of a DMA module is also under study, in order to reduce the time consumed in memory accesses.

It can be also concluded that, in communications algorithms in which operations like FFTs, matrices multiplications, averages, and norms, among others, are needed, DSP blocks and, overall, embedded multipliers are a very valuable resource, as it is not possible to expect the same speed from a multiplier that has been implemented with logic blocks available in the FPGA; as the dedicated multipliers are assigned to some of the built accelerators, the synthesis tool has to use the logic blocks to implement the rest of such multipliers, an issue that impacts the maximum frequency of the system directly. This situation is very different from that of the memory, due to the multiple options that can be found nowadays in FPGA boards. It is true that the memory blocks inside the reconfigurable chip present the smallest latency, but a reduction in the access speed of off-chip memories (SRAM, SDRAM) can be tolerated if it allows the storage of a significantly bigger amount of data. Furthermore, experimental results show that the difference of maximum frequencies between on-chip and off-chip memories are not very significant, if techniques as the fetching of several data at the same time (like in the case of the arithmetic mean accelerator) are used.

With this work, it was possible to detect the most problematic stage of a receiver based on DDST: the Carry Frequency Offset Estimation. Nevertheless, the use of FFTs and a *look-up table/parallel/iterative* norm accelerators make the calculation of trigonometric functions (sines and cosines) the only obstacle left in order to obtain a practical system for DDST.

The hybrid software-hardware approach demonstrated to be very versatile and flexible, allowing fast implementation of several kinds of algorithms and their fast modification, from a small change in the input parameters values to the alteration of a full stage of the process. In fact, the

built prototype fits perfectly into the study of the DDST algorithm, as this algorithm is still under study and constant modifications and improvements have been made over it. For example, the first versions of superimposed training worked under the time domain, so operations like the FFT were not necessary. If future changes in the algorithm require a significant modification of any of the accelerators, it will be very easy to adapt the whole system.

References

[1] M. Dong, L. Tong, and B. M. Sadler, "Optimal insertion of pilot symbols for transmissions over time-varying flat fading channels," *IEEE Transactions on Signal Processing*, vol. 52, no. 5, pp. 1403–1418, 2004.

[2] A. G. Orozco-Lugo, M. Mauricio Lara, and D. C. McLernon, "Channel stimation using implicit training," *IEEE Transactions on Signal Processing*, vol. 52, no. 1, pp. 240–254, 2004.

[3] E. Alameda-Hernandez, D. C. McLernon, M. Ghogho, A. G. Orozco-Lugo, and M. Mauricio Lara, "Improved synchronisation for superimposed training based channel estimation," in *Proceedings of the 13th IEEE/SP Workshop on Statistical Signal Processing Proceedings*, pp. 1324–1329, Bordeaux, France, July 2005.

[4] M. Ghogho, D. C. McLernon, E. Alameda-Hernandez, and A. Swami, "Channel estimation and symbol detection for block transmission using data-dependent superimposed training," *IEEE Signal Processing Letters*, vol. 12, no. 3, pp. 226–229, 2005.

[5] E. Alameda-Hernandez, D. C. McLernon, A. G. Orozco-Lugo, M. Mauricio Lara, and M. Ghogho, "Synchronisation and DC-offset estimation for channel estimation using data-dependent superimposed training," in *Proceedings of the European Signal Processing Conference (EUSIPCO '05)*, Istanbul, Turkey, September 2005.

[6] A. G. Orozco-Lugo, G. M. Galvan-Tejada, M. Mauricio Lara, and D. C. McLernon, "A low complexity iterative channel estimation and equalisation scheme for (data dependent) superimposed training," in *Proceedings of the European Signal Processing Conference (EUSIPCO '06)*, Florence, Italy, September 2006.

[7] M. Helfenstein and G. S. Moschytz, *Circuits and Systems for Wireless Communications*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[8] A. G. Orozco-Lugo, M. Mauricio Lara, E. Alameda-Hernandez, S. Moosvi, and D. C. McLernon, "Frequency offset estimation and compensation using superimposed training," in *Proceedings of the 4th International Conference on Electrical and Electronics Engineering (ICEEE '07)*, vol. 5, pp. 118–121, September 2007.

[9] K. Piromsopa, C. Aportewan, and P. Chongstitvatana, "An FPGA implementation of a fixed-point square root operation," in *Proceedings of the International Symposium on Communications and Information Technology*, vol. 14–16, pp. 587–589, Thailand, 2001.

[10] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 191–200, Monterey, Calif, USA, February 1998.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 242981, 20 pages
doi:10.1155/2009/242981

Research Article

An Adaptive Message Passing MPSoC Framework

Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, and Michel Robert

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), Centre National de la Recherche Scientifique (CNRS), University of Montpellier 2, 161 rue Ada, 34392 Montpellier, France

Correspondence should be addressed to Gabriel Marchesan Almeida, gabriel.marchesan@lirmm.fr

Received 19 December 2008; Accepted 14 April 2009

Recommended by J. Manuel Moreno

Multiprocessor Systems-on-Chips (MPSoCs) offer superior performance while maintaining flexibility and reusability thanks to software oriented personalization. While most MPSoCs are today heterogeneous for better meeting the targeted application requirements, homogeneous MPSoCs may become in a near future a viable alternative bringing other benefits such as run-time load balancing and task migration. The work presented in this paper relies on a homogeneous NoC-based MPSoC framework we developed for exploring scalable and adaptive on-line continuous mapping techniques. Each processor of this system is compact and runs a tiny preemptive operating system that monitors various metrics and is entitled to take remapping decisions through code migration techniques. This approach that endows the architecture with decisional capabilities permits refining application implementation at run-time according to various criteria. Experiments based on simple policies are presented on various applications that demonstrate the benefits of such an approach.

1. Introduction

The exponentially increasing number of transistors that can be placed on an integrated circuit is permitted by the dropping of technology feature sizes. This trend plays an important role at the economic level, although the price per transistor is rapidly dropping the NRE (Nonrecurring Engineering) costs, and fixed manufacturing costs increase significantly. This pushes the profitability threshold to higher production volumes opening a new market for flexible circuits which can be reused for several product lines or generations and scalable systems which can be designed more rapidly in order to decrease the Time-to-Market. Moreover, at a technological point of view, current variability issues could be compensated by more flexible and scalable designs. In this context, Multiprocessor Systems-on-Chips (MPSoCs) are becoming an increasingly popular solution that combines flexibility of software along with potentially significant speedups.

These complex systems usually integrate a few mid-range microprocessors for which an application is usually statically mapped at design-time. Those applications however tend to increase in complexity and often exhibit time-changing workload which makes mapping decisions suboptimal in a number of scenarios. Additionally, such systems are designed in very deep-submicron technologies that bring a number of hardly predictable physical effects that, associated also to the increasing process variability, demonstrate the intrinsic and unavoidable unreliability of future nanoscale integrated systems.

These facts challenge the design techniques and methods that have been used for decades and push the community to research new approaches for achieving system adaptability and reliability (out of unreliable technology components).

This paper presents a hardware/software framework (HS-Scale platform) that is based on a set of adaptive principles which endows the architecture with some decisional capabilities. This approach helps continuously refining application mapping for optimizing various criteria such as performance or power consumption and should eventually enable fault tolerance.

This hardware/software framework is intended to permit the exploration of scalable solutions for future MPSoCs

in the context of massive on-chip parallelism with several hundreds of processing elements (PEs). Therefore, the proposed architecture relies on principles that do not imply resource sharing among processors in the broad sense of the term. The system is made of a regular arrangement of PEs that runs applications in a distributed way, exchanging messages that relate to both platform management and application data. The used programming model is derived from a popular message passing interface (MPI) system that has been augmented for supporting adaptive mechanisms.

This paper is organized as follows.

Section 2 presents the related works in the field of multiprocessor systems, programming models, and task migration techniques. Section 3 introduces the HS-Scale framework which covers the hardware, software, and the programming model used in this approach. Section 4 shows the validations in terms of both developed hardware and area utilization figures in the context of an SoC realization. Section 5 presents the results of various applications mapped on the framework emphasizing on the cost induced by the used migration techniques and the corresponding observed benefits. Section 6 draws some conclusions on the presented work and puts this in perspective with other upcoming challenges of the area.

2. Related Works

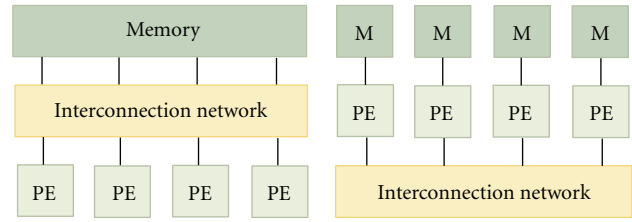
This section shortly introduces the different existing families of multiprocessor systems and puts focus on the relevant approaches that are found in the fields of scalable message passing architectures and task migration techniques.

2.1. Preliminary Considerations. Multiprocessor systems are increasely considered as an attractive solution for accelerating computation. Parallel architectures have been studied intensively during the past 40 years; there is consequently a huge amount of books [1, 2] related to this topic, and we will therefore only focus on general concepts. The most common type of multiprocessor systems falls into the Multiple Instruction Multiple Data (MIMD) family as that defined by the Flynns taxonomy [3]. There are two types of MIMD machine classified in accordance with their memory architecture:

(i) shared memory architectures in which all processors share the same memory resources; therefore all changes done by a processor to a given memory location become visible to all other processors of the system;

(ii) distributed memory architectures in which every processor has its own private memory; therefore one processor cannot read directly in the memory of another processor. Data transfers are implemented using message passing protocols.

From an architecture design point of view shared memory machines are poorly scalable because of the limited bandwidth of the memory. Existing realizations marginally use more than tenth of processors because of this reason.



(a) Shared memory (b) Distributed memory

Figure 1: Shared *x* distributed memory.

Distributed memory machines are more scalable since only the communication medium may be shared among processors. There exist two families of programming models, each of which exhibits a better adequacy to one of the previously presented architecture families.

(i) Shared memory systems require synchronization mechanisms such as semaphores, barriers, and locks since no explicit communication mechanism exists. POSIX threads [4] and OpenMP [5] are two popular implementations of the thread model on shared memory architectures.

(ii) Distributed memory systems require mechanisms for supporting explicit communications between processes (that may be hosted on the same or different processors). Usually a library of primitives that allow writing in communication channels is used. The Message Passing Interface (MPI) [6] is the most popular standard that is used in High-Performance Computing (HPC) computer clusters for instance.

In a shared memory architecture (Figure 1(a)), processes (executed by different processors) can easily exchange information through shared variables; however it requires handling carefully synchronization and memory protection. In a distributed memory architecture (Figure 1(b)), a communication infrastructure is required in order to connect processing elements and their memories and allow exchanging information.

Since this work targets massively parallel on-chip Multiprocessor systems, scalability is a major concern in the approach. For this reason, we put focus on distributed memory machines and therefore choose a message passing programming model for it provides a natural mapping to such machines.

2.2. Message Passing for Embedded Systems. The Message passing model is based on explicit communication between tasks. This model is often used for architectures that do not provide global address space; here communications among tasks take place through messages and are implemented with functions allowing reading and writing to communication channels. CORBA, DCOM, SOAP, and MPI are examples of message passing models.

Message Passing Interface (MPI) is the most popular implementation of the message passing model, and only for this model some embedded implementations exist. The

Message Passing Interface is a specification for an API that allows many PEs to communicate through a communication network.

MPI provides a comprehensive number of primitives that relate to general-purpose distributed computing; a number of work have devised lightweight implementations supporting only a subset of the mechanisms of MPI for embedded processors and systems. This makes sense since the nature of applications for these systems is well defined, often limited to data flow applications for which Kahn Process Networks formalism offer a sufficient support that requires only blocking read operations [7]. Some MPI implementations are layered, and advanced communication synchronization primitives (such as collective, etc.) found in the upper layers make use of the simple point-to-point primitives such as MPI_Send() and MPI_Receive(). This enables using these collective mechanisms in an application-specific basis in case they prove necessary.

The specific requirements of embedded systems have led programmers to develop lightweight MPI implementations, basically built upon a subset of the original MPI mechanisms. In [8] the authors present TMD-MPI which is a lightweight MPI implementation for multiple processors across multiple FPGAs. It relies on a layered implementation which provides only 11 primitives. As no operating system is used, task mapping is static and done at design-time. Similarly, authors in [9] have also selected 11 primitives among which only 2 relate to point-to-point communications. Finally, in [10] authors present eMPI which also uses the simplest low-level point-to-point communication primitives in a layered style.

2.3. Task Migration Support. Task migration techniques have been mainly studied in contexts that fall in one of the following areas.

(i) General purpose computing, involving usually a single computer made of several processors or processor cores. Such systems are usually built around shared memory architectures.

(ii) High-Performance Computing (HPC) computer clusters. Such systems are usually of distributed memory type and therefore generally use message passing programming style.

(iii) Multiprocessor embedded systems, which may make use of either shared or distributed memory architecture.

For shared memory systems such as today's multicore computers, task migration is facilitated by the fact that no data or code has to be moved across physical memories: since all processors are entitled to access any location in the shared memory, migrating a task comes down to electing a different processor for execution. There exist several efficient implementations on general purposes OS such as Windows or Linux [11].

In the case of multiprocessor-distributed memory/message passing architectures, both process code and state have to be migrated from a processor private memory to another, and synchronizations must be performed using exchanged messages such as in [12] which targets Linux computer clusters. Some other approaches aimed at augmenting MPI for providing a support for process migration, such as [13, 14]. In [15] users present similar features based on a JAVA MPI framework that provides hardware independence; they show that despite migrating tasks imply overheads, which are in the order of seconds, significant speedups can be achieved. All these approaches target computer clusters with the typical resources of general-purpose computers and are therefore hardly applicable to MPSoCs.

Task migration has also been explored for MPSoCs, notably based on locality considerations [12] for decreasing communication overhead or power consumption [16]. In [17], authors present a migration case study for MPSoCs that relies on the Clinux operating system and a check pointing mechanism. The system uses the MPARM framework [18], and although several memories are used, the whole system supports data coherency through a shared memory view of the system.

In [19] authors present an architecture aiming at supporting task migration for a distributed memory multiprocessor-embedded system. The developed system is based on a number of 32-bit RISC processors without memory management unit (MMU). The used solution relies on the so-called "task replicas" technique; tasks that may undergo a migration are present on every processor of the system. Whenever a migration is triggered, the corresponding task is respectively inhibited from the initial processor and activated in the target processor. This solution induces a significant memory overhead for every additional task and therefore falls beyond the scope of this paper. Finally, to the best of our knowledge, no other work combines the use of a message passing programming model, on-chip multiprocessor system, and transparent decentralized automated task migration.

3. HS-Scale

The key motivations of our approach are scalability and self-adaptability; the system presented in the rest of this paper is built around a distributed memory/message passing system that provides efficient support for task migration. The decision-making policy that controls migration processes is also fully distributed for scalability reasons. This system therefore aims at achieving continuous, transparent, and decentralized run-time task placement on an array of processors for optimizing application mapping according to various potentially time-changing criteria.

3.1. System Overview. The system is based on an array of compact general-purpose PEs interconnected through a packet switching Network-on-Chip. The HS-scale system is a purely distributed memory system which is programmed using a simple message passing protocol. Contrary to MPI, processes must not be mapped to a given processor but shall freely move in the system according to user-definable policies that may aim at optimizing a given property in the system, such as performance or power consumption. Both hardware and software resources are intended to be minimalist for

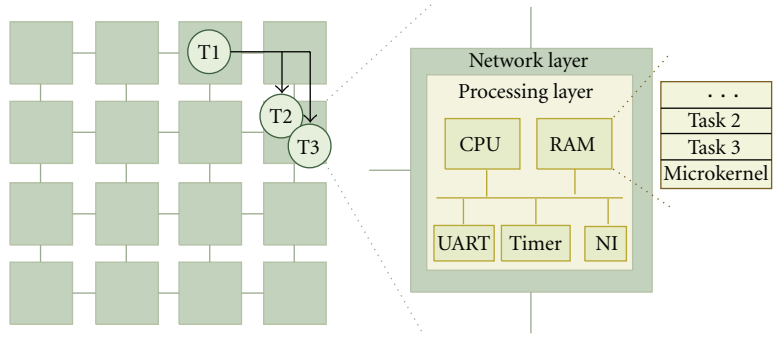


Figure 2: NPU structural description.

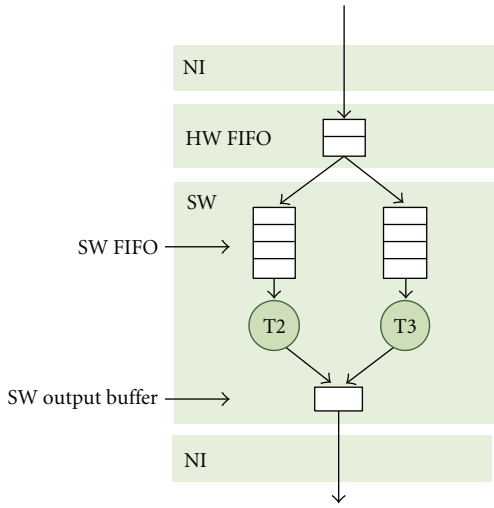


Figure 3: NPU functional description.

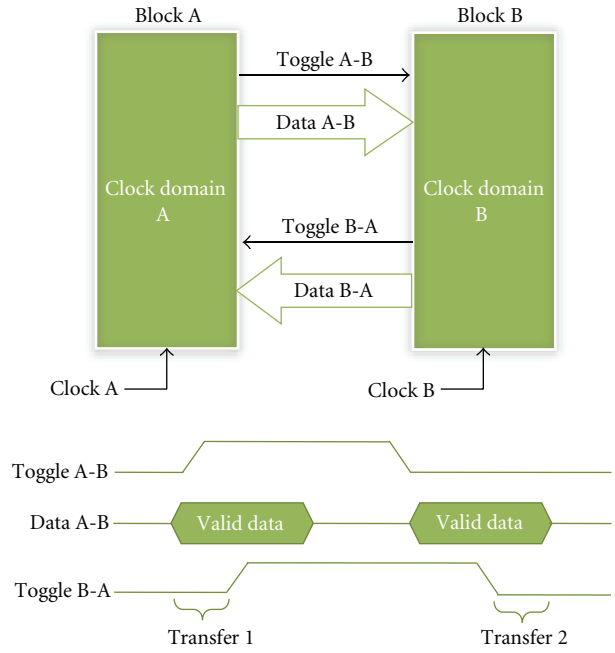


Figure 4: The asynchronous toggle protocol.

favoring compactness of processors and therefore encouraging massive parallelism. Also, for scalability reasons, there exists no master in the system unless a given application requires it.

3.2. Hardware Structure

3.2.1. Network Processing Unit. The architecture is made of a homogeneous array of Processing Elements (PEs) communicating through a packet-switching network. For this reason, the PE is called NPU, for Network Processing Unit. Each NPU, as detailed later, has multitasking capabilities which enable time-sliced execution of multiple tasks. This is implemented thanks to a tiny preemptive multitasking Operating System which runs on each NPU. The structural and functional views of the NPU are depicted in Figures 2 and 3, respectively.

The NPU is built around two main layers, the network layer and the processing layer. The Network layer is essentially a compact routing engine (XY routing). Packets are read from incoming physical ports, then forwarded to either outgoing ports or the processing layer. Whenever a packet header specifies the current NPU address, the packet is forwarded to the network interface (NI in Figure 3). The network interface buffers incoming data in a small hardware FIFO and simultaneously triggers an interrupt to the processing layer. The interrupt then activates data demultiplexing from the single hardware FIFO to the appropriate software FIFO as illustrated in Figure 4.

The processing layer is based on a simple and compact RISC microprocessor, its static memory, and a few peripherals (one timer, one interrupt controller, one UART) as shown in Figure 2. A multitasking microkernel implements the support for time-multiplexed execution of multiple tasks.

The processor used has a compact instruction set comparable to an MIPS-1 [20]. It has 3 pipelines stages, no cache, no Memory Management Unit (MMU), and no memory protection support in order to keep it as small as possible.

3.2.2. Communication Infrastructure. For technology-related concerns, a regular arrangement of processing elements (PEs) with only neighboring connections is favored. This helps in (a) preventing using any long lines and their associated undesirable physical effects in deep submicron CMOS

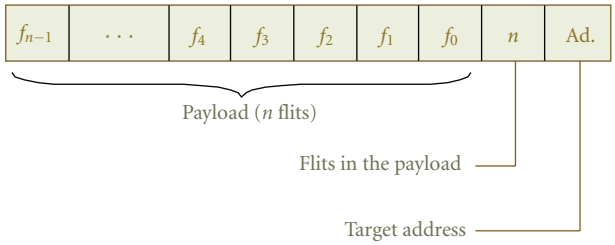


Figure 5: Packet format.

technologies and (b) synthesizing the clock distribution network since an asynchronous communication protocol between the PEs might be used. Also, from a communication point of view, the total aggregated bandwidth of the architecture should increase proportionally with the numbers of PEs it possesses, which is granted by the principle of abstracting the communications through routing data in space. The Network-on-Chip (NoC) paradigm enables that easily thanks to packet switching and adaptive routing.

The communication framework of HS-scale is derived from the Hermes Network-on-chip; refer to [21] for more details. The routing is of wormhole type, which means that a packet is made of an arbitrary number of flits which all follow the route taken by the first one which specifies the destination address. Figure 5 depicts the simple packet format used by the network framework constituted by the array of processing elements. Incoming flits are buffered in input buffers (one per port). Arbitration follows a round-robin policy giving alternatively priority to input ports. Once access to an output port is granted, the input buffer sends the buffered flits until the entire packet is transmitted (wormhole routing).

Inter-NPU communications are fully asynchronous and are based on the toggle-protocol. As depicted in Figure 4 this protocol uses two toggle signals for the synchronization, a given data being considered valid when a toggle is detected. When the data is latched, another toggle is sent back to the sender to notify the acceptance. This solution allows using completely unrelated clocks on each PE in the architecture.

3.3. Operating System. The lightweight operating system we use was designed for our specific needs. Despite being small (28 KB), this kernel does preemptive switching between tasks and also provides them with a set of communication primitives that are presented later. Figure 6 gives an overview of the operating system infrastructure and the services it provides.

Figure 7 presents the process state diagram each task follows depending on the events that may occur. This scheme answers to the general principles of operating systems in general although transition events have been specialized for this specific case.

The interrupts manager may receive interrupts from 3 hardware sources: UART, Timer, and network interface (NI). Whenever an interrupt occurs, other interrupts are disabled, and the processor context is saved in the system stack. Following the type of interrupt, it reads from the UART, schedules another task (timer), receives data from other NPUs, or use a communication primitive (interrupt from the network interface FIFO_in). Afterwards processor context is restored, and interrupts are re-enabled. The scheduler is the core of the microkernel. Each time a timer interrupt occurs, it checks if there is a new task to run. In the positive case, it executes this new task. Otherwise, it has two possibilities: either there is no task to schedule then it just runs an idle task or there is at least one task to schedule. Tasks are scheduled periodically following a round robin policy (there is no priority management between tasks) as depicted in Figure 8.

Figure 9 presents the memory layout of an NPU running this operating system along with two tasks. Each task is located in a memory region that embeds code, data, and stack segments.

3.4. Dynamic Task Loading and Migration (PIC). One of the objectives of this work is to enable dynamic load balancing which implies the capability to migrate running tasks from processor to processor. Migrating tasks usually implies the following.

(i) To dynamically load in memory and schedule a new process.

(ii) To restore the context of the task that has been migrated.

3.4.1. Dynamic Process Loading. Both points are challenging for such microprocessor targets since, for density reasons, no Memory Management Unit (MMU) is available. An MMU, among other tasks, usually performs the translation between virtual and physical addresses and therefore permits to load and run a code in an arbitrary region of the physical memory. The code then performs read, write, and jump operations to virtual memory locations that are being translated into physical locations matching the memory layout decided by the operating system upon loading.

A possible alternative to overcome this problem relies in resolving all references of a given code at load-time; such a feature is partly supported in the ELF format [22] which lists the dynamic symbols of the code and enables the operating system loader to link the code for the newly decided memory location. Such mechanisms are memory consuming and imply a significant memory overhead which clearly puts this solution out of the scope of the approach.

Another solution for enabling the loading of processes without such mechanisms relies on a feature that is partly supported by the GCC compiler that enables to emit relocatable code (PIC: Position Independent Code). This feature generally used for shared libraries generates only relative jumps and accesses data locations and functions using a Global Offset Table (GOT) that is embedded into the generated ELF file. A specific postprocessing tool which operates on this format was used for reconstructing a completely relocatable executable. Experiments show that both memory and performance overheads remain under 5% for this solution which is clearly acceptable.

Figure 10(a) shows an example of relative jump with PIC compilation with code compiled for an execution at the

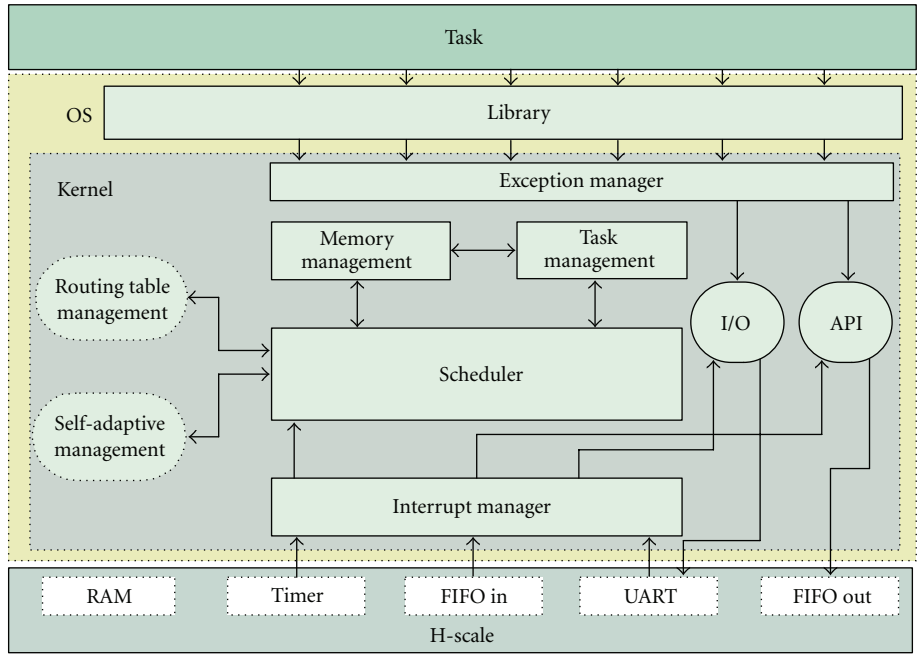


Figure 6: Operating system overview.

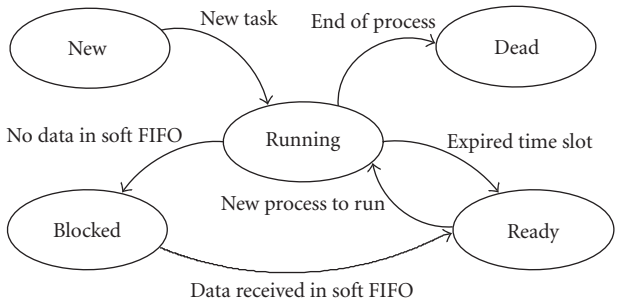


Figure 7: Task state diagram.

address 0x0000; the address for the jump is referenced by the sum of current address and GOT entry reference. So if the code needs to be executed to another location, for example, 0×0100 (Figure 10(b)), the code is copied to the new location, and the only modification to perform is to add the same offset of the code (0×0100) to the GOT section entries.

3.4.2. Task Context Migration. Migrating a process implies not only instantiating a new executable into the memory but also restoring its context. Again, the lack of MMU makes this task difficult since the context of the process includes the stack which not only embeds data (such as return values of functions) but also returns addresses that are memory-location dependent. The solution we developed is based on defining migration points that are at specific locations in the code, namely, whenever a communication primitive is called. This method is restrictive since it assumes that the computation relies on a strict consumer/producer model where no internal state is kept from iteration to iteration. This translates in the fact that there cannot be any dependencies between two adjacent computed data chunks.

When a task migration order is issued by the operating system, the following sequence of action is initiated between NPU1 which is current host for the task and NPU2 which is the future.

(1) Task code is sent from NPU1 to NPU2. NPU2 then loads the task into memory, creates the necessary software FIFOs, and runs the task which is frozen when it reaches the first communication primitive call (MPI_Send() or MPI_Receive()).

(2) NPU1 modifies routing tables (that embeds task and FIFO placements) and broadcasts this information to the other NPUs. Future messages for the task will be buffered in the newly created FIFO on NPU2.

(3) Task execution on NPU1 continues until the next communication primitive call is reached which freezes application execution. Following this, the remaining task FIFO content on NPU1 is sent and reordered on NPU2.

(4) Task execution is resumed on NPU2, and concurrently the task is removed from memory on NPU1.

Table 1 presents an example of migration time of one task (20 KB), between two NPUs with a distance of one. These results show that time migration is mainly due to the time to send executable through the network (with a frequency $f = 25$ MHz). *TShutdown* refers to the time taken by the operating system for unregistering the task. *TSend* represents the time taken for the operating system to perform the necessary actions for formatting and sending the task to

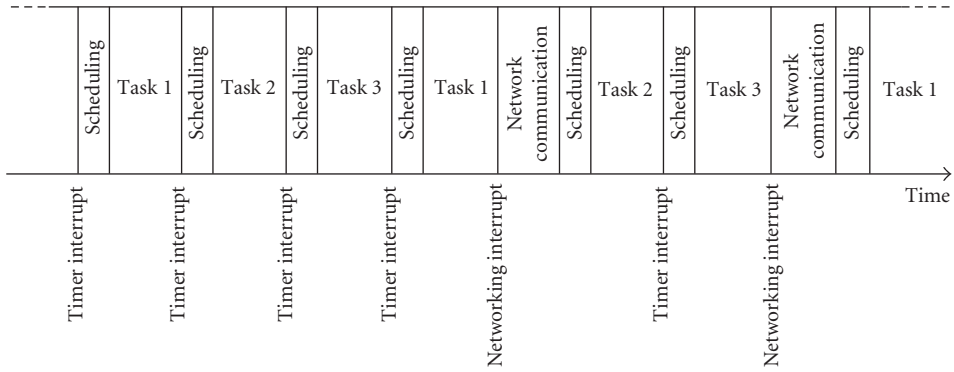


Figure 8: Scheduling diagram.

Table 1: Timeline of the migration mechanism.

	TShutdown	*TSend*	*TReceive*	*TRelight*	*TReboot*	*TMigration*
Time (ms)	3.067	13.970	13.968	3.110	0.107	34.222

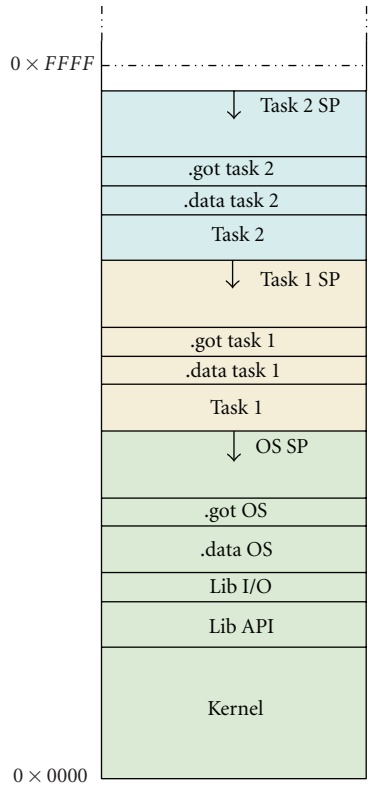


Figure 9: Memory layout.

the remote NPU, the time actually spent in sending the task through the network *TSend* being negligible. Similarly, the operating system of the remote NPU requires a significant amount of time *TReceive* for receiving and instantiating the task in memory. Finally, the last action taken before running the task is updating both local and remote routing tables; this is realized in a time *TRelight*. The task is then rapidly scheduled and executed (*TReboot*).

3.5. Programming Model. Programming takes place using a message passing interface. Hence, tasks are hosted on NPUs which provide through their operating system communication primitives that enable data exchanges between communicating tasks. The proposed model used only two communication primitives, MPI_Send() and MPI_Receive(). This communication primitive is based on the synchronous MPI communication primitive (MPI_Send and MPI_Receive). Figure 11 depicts the layered view of the communication protocol we use. MPI_Receive() blocks the task until the data is available while MPI_Send() is blocking until the buffer is available on the sender side. In our implementation each call exhibits this behavior and is translated into a sequence of low-level Send_Data()/Receive_Data() methods that set up a communication channel through a simple request/acknowledge protocol as depicted in Figure 12. This protocol ensures that the remote processor buffer has sufficient space before sending the message which helps lowering the contentions in the communication network and also prevents deadlocks.

Figure 12 depicts the communication stack that is used in our system. Although a hardware implementation could certainly help improving performance, for compactness reasons it is fully implemented in software down to packet assembling.

No explicit group synchronization primitives are provided; however this can be simply achieved in an ad hoc fashion through using MPI_Send() and MPI_Receive() for passing tokens. Broadcast, gather, and similar mechanisms can also be implemented in the same manner. Furthermore, although it could be easily implemented, no nonblocking receive method is provided since the targeted applications usually do not require it.

The prototypes of those functions are as in Algorithm 1.

The prototypes of these functions are self explanatory, a reference to the graph edge identifier, a constant void pointer, and data size expressed in bytes.

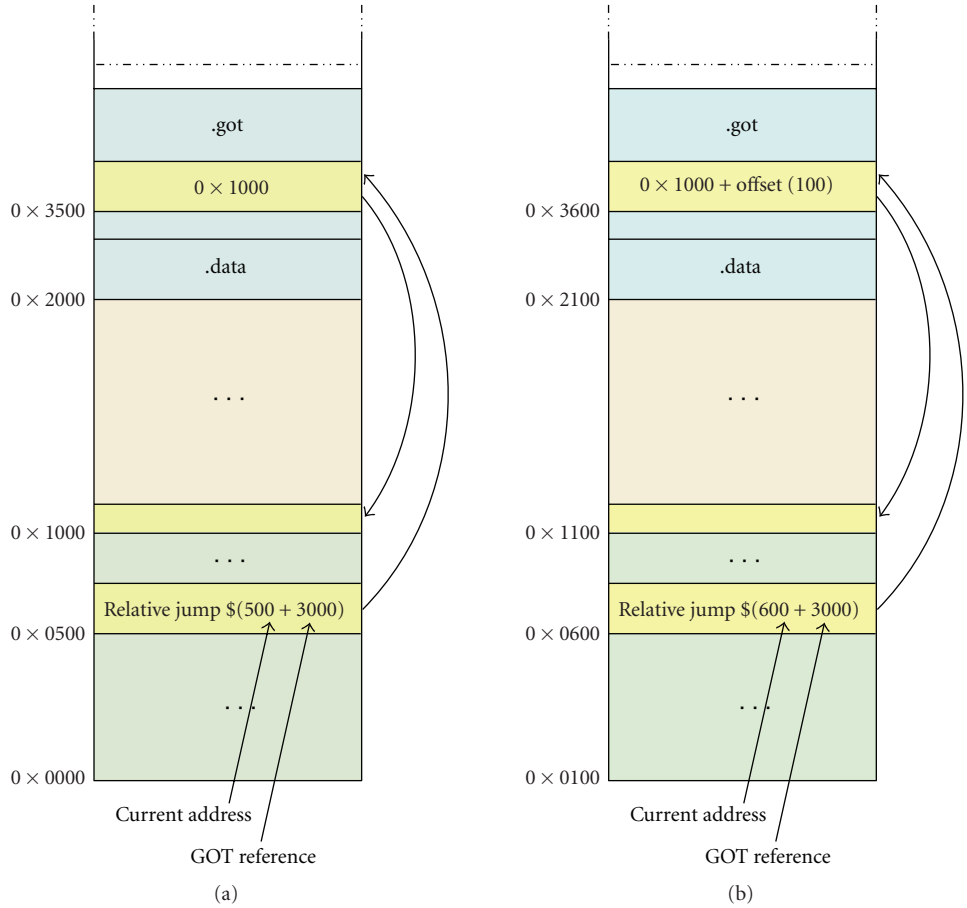


Figure 10: Relative jumps with GOT.

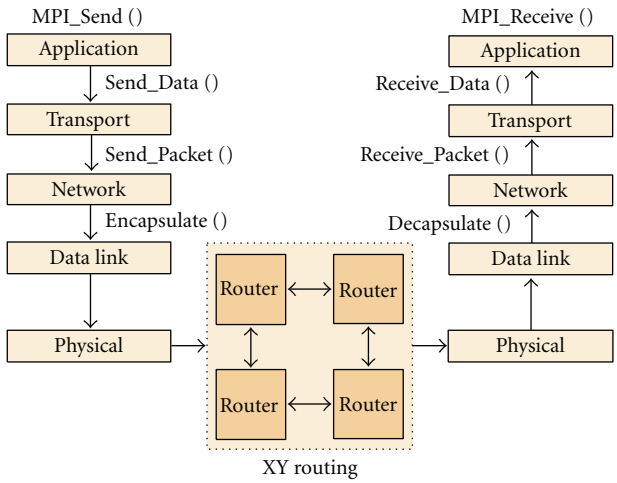


Figure 11: HS-Scale protocol stack.

Figure 13 shows an example of task graph where it can be seen that communication channels feature a (software) FIFO queue at the receiver side. Queues sizes can be parameterized, and their size can be tuned on-line as the operating system provides memory allocation and deallocation services.

```
MPI_Send(int edge, const void *data, int size)
MPI_Receive(int edge, void *data, int size)
```

Algorithm 1

3.6. Self-Adaptive Mechanisms. The platform is entitled to take decisions that relate to application implementation through task placement. These decisions are taken in a fully decentralized fashion as each NPU is endowed with equivalent decisional capabilities. Each NPU monitors a number of metrics that drive an application-specific mapping policy; based on these information an NPU may decide to push or attract tasks which results in, respectively, parallelizing or serializing the corresponding tasks execution, as several tasks running onto the same NPU are executed in a time-sliced manner.

Figure 14 shows an abstract example where it can be observed that upon application loading the entire task graph runs onto a single NPU; subsequent remapping decisions then tend to parallelize application implementation as the final step exhibits one task per NPU. Similarly, whenever a set of tasks become subcritical, the remapping could revert to situation 3 where T1, T2, and T3 are hosted on a single

NPU while the other supposedly more demanding do not share NPU processing resources with other tasks. These mechanisms help achieving continuous load-balancing in the architecture but can depending on the chosen mapping policy help refining placement for lowering contentions, latency, or power consumption.

Mapping decisions are specified on an application-specific basis in a dedicated operating system service. Although the policy may be focused on a single metric, composite policies are possible. Three metrics are available to the remapping policy for taking mapping decisions.

(i) *NPU load*. The NPU operating system has the capability of evaluating the processing workload resulting from task execution.

(ii) *FIFO queues filling level*. As depicted in Figure 13, every task has software input FIFO queues. Similarly to NPU load, the operating system can monitor the filling of each FIFO.

(iii) *Task distance*. The distance that separates tasks is also a factor that impacts performance, contentions in the network, and power consumption. Each NPU microkernel knows the placement of other tasks of the platform and can calculate the Manhattan distance with the other tasks it communicates with.

Algorithm 2 shows an implementation of the microkernel service responsible of triggering task migrations. The presented policy simply triggers task migration in case one of the FIFO queues of a task is used over 80%.

The request_task_migration() call then sequentially emits requests to NPUs in proximity order; the migration function will migrate the task to the first NPU which has accepted the request; the migration process is started according to the protocol described previously in Section 3.4.2. This function can naturally be tuned on an application/task specific basis and select the target NPU taking into account not only the distance but also other parameters such as available memory and current load.

We have implemented also a migration policy based on the CPU load. The idea is very similar to the first one, and it consists of triggering a migration of a given task when the CPU load is lower or greater than a given threshold. This approach may be subdivided in two subsets.

(1) Whenever the tasks time is greater than or equal MAX_THRESHOLD, it means that tasks are consuming more than or equal to the maximum acceptable usage of the CPU time.

(2) Whenever the tasks time is less than MIN_THRESHOLD, it means that the tasks are consuming less than the minimum acceptable usage of the CPU time.

For both subsets, the number of tasks inside one NPU must be verified. For the first subset, it is necessary to have, at least, two tasks running in the same NPU. For the second subset, whenever there are one or more tasks in the same NPU, the migration process may occur.

In the same way the migration process occurs whenever the CPU load is less than MIN_ THRESHOLD (20%). When

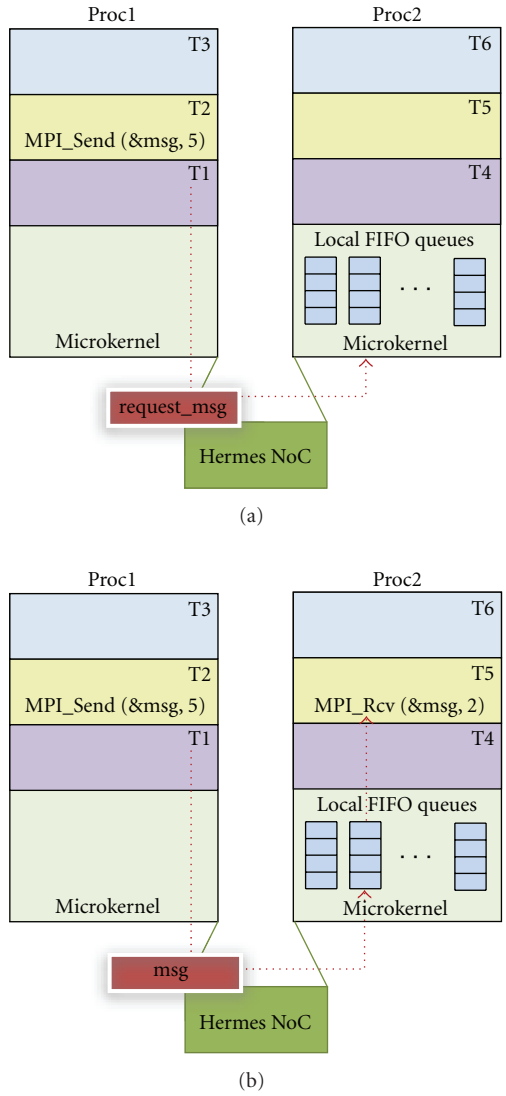


Figure 12: Proactive communication principle.

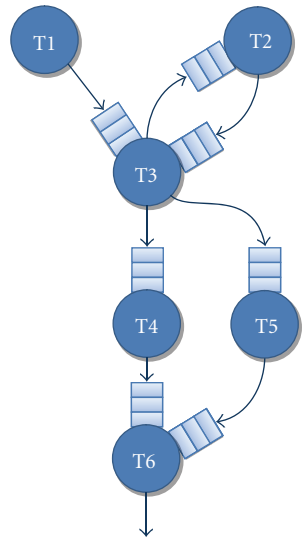


Figure 13: Example of task graph.

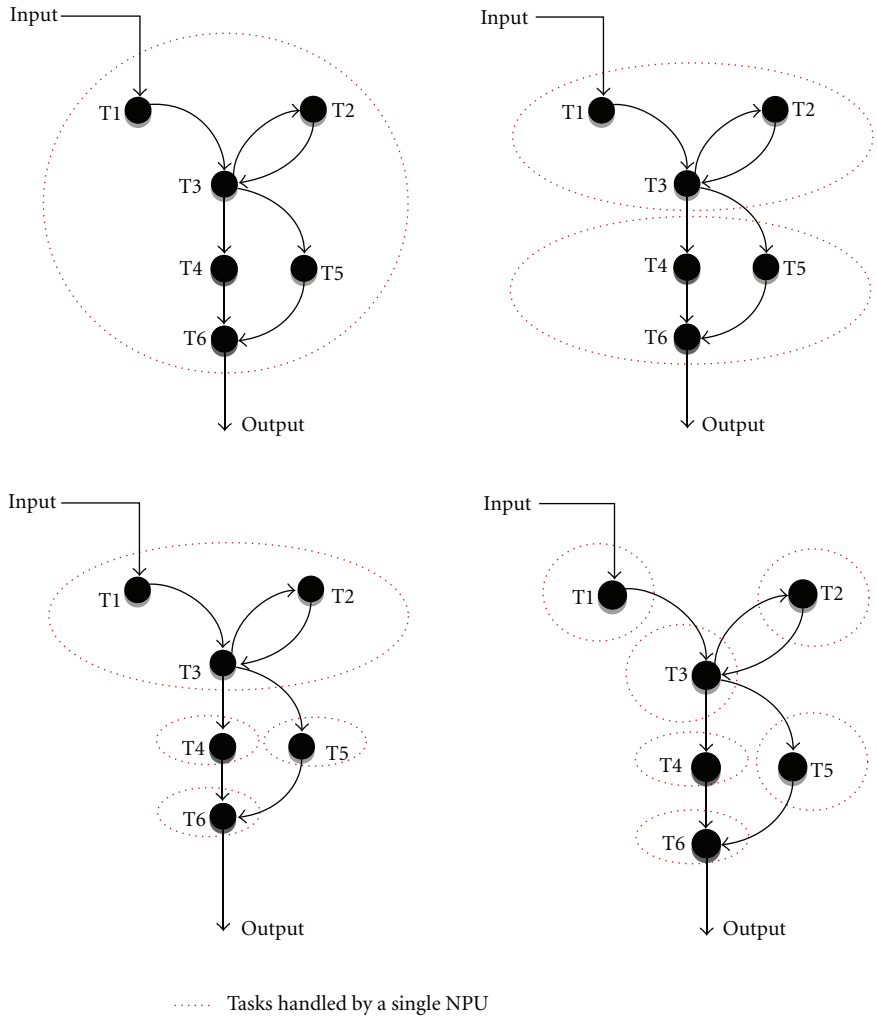


Figure 14: Task graph.

```
void improvement_service_routine(){
    int i, j;
    //Cycles through all NPU tasks
    for (i = 0; i < MAX_TASK; i++){
        //Deactivates policy for dead/newly instantiated tasks
        if (tcb[i].status != NEW && tcb[i].status != DEAD){
            //Cycles through all FIFOs
            for (j = 0; j < tcb[i].nb_socket; j++){
                //Verifies if FIFO usage > MAX_THRESHOLD
                if (tcb[i].fifo_in[j].average > MAX_THRESHOLD){
                    //Triggers migration procedure if task
                    //is not already alone on the NPU
                    if (num_task > 1)
                        request_task_migration(tcb[i].task_ID);
                }
            }
        }
    }
}
```

Algorithm 2

Table 2: Area scalability results.

Number of NPU	1	2	4 (2 × 2)	9 (3 × 3)	16 (4 × 4)
Area (mm^2)	1.14	2.29	4.60	10.33	18.40

this occurs, the migration function must look for a NPU that is using at given threshold of CPU usage, in this case, 60% of usage. To avoid the task with less than MIN_THRESHOLD keep migrating every time, we have inserted a delay to reduce the number of migrations.

4. Validations

4.1. Estimations of the Silicon Hardware Prototype. A complete synthesizable RTL level description (about 6000 lines of VHDL) of the H-Scale system has been designed. It has allowed us to validate our approach, to estimate areas (post place and route, with ST Microelectronics 90 nm design kit), and to improve the design. Any instance of the H-Scale MP-SOC system may be easily generated with generic parameters and then evaluated with any standard CAD tool flow (Encounter Cadence was used).

Table 2 summarizes these evaluations. The clock of the NPU has been constrained to 3 nanoseconds allowing a 300 MHz frequency. Table 2 clearly shows the area scalability of H-Scale hardware system (the very low overhead is due to the wires needed to interconnect the NPUs). These results let us easily extrapolate that we could design an HS-Scale system with 32 processors and 2 MB of embedded memory with less than 50 mm^2 of silicon area.

4.2. Multi-FPGA Prototype. The first validations of the systems were performed thanks to VHDL simulation. Obviously, this was far too slow for realistic application scenarios (about 4 minutes for a 10 milliseconds simulation with a 1.6 GHz processor). Although a SystemC prototype is also available, we chose to develop a scalable multi-FPGA prototype.

4.2.1. Platform Description. It is essentially based on a Spartan3 S1000 FPGA which 1920 configurable logic blocks (CLB). The board features several general purpose I/Os (8 slide switches, 4 pushbuttons, 8 LEDs, and 4-digit seven-segment display), 1 MB of fast asynchronous SRAM, several ports for debugging/monitoring purposes (one serial port, a VGA port, and PS2 mouse/keyboard port), and three 40-pin expansion connectors for the interconnections of boards.

As mentioned, one NPU is synthesized on a single FPGA board. The maximum frequency of the synthesized design on Spartan3 S1000 FPGA is 25 MHz. Figure 15 depicts the board with two of the 40-pin expansion connectors used for North, South, East, and West connections. Communications are taking place in an asynchronous fashion as described previously (toggle protocol).

Table 3 gives the device utilization figures for a single NPU hosted on a single board. The complete prototype is then composed of several instances of the prototyping boards connected through the 40-pin expansion connectors.

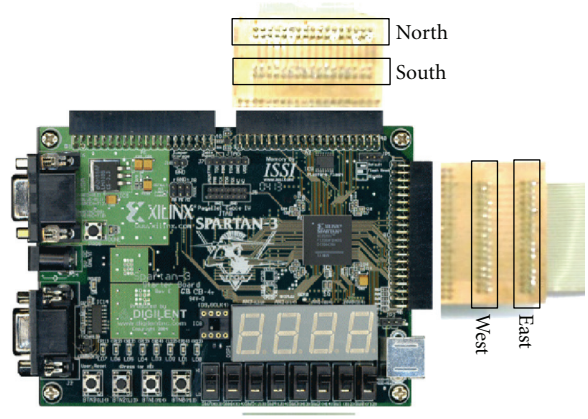


Figure 15: The prototyping board.

One board, that is, one UART of a single NPU, is directly connected to a PC as depicted in Figure 16. This PC is used as a human-machine interface for sending program data (i.e., task codes and microkernel code), the data to compute and to display debugging messages in the monitoring terminal.

Each NPU has originally a bootloader which performs upon power up the following operations in sequence.

(1) It checks whether a PC is connected to the UART port. If so, the NPU initializes its XY coordinates to address $(0 : 0)$. It then acts as a Dynamic Host Configuration Protocol (DHCP) server and proactively sends packets to the East and South ports informing that it has taken address $(0 : 0)$.

(2) If no UART connection is detected, an incoming network request is expected. Once the corresponding packet is received (that the interface NPU has initiated as described above), an address is calculated: East neighbor will take address $(1 : 0)$ and South neighbor address $(0 : 1)$.

(3) This process is reiterated until the boundaries of the network are found. The X-axis and Y-axis boundaries are then broadcasted in the network in order to inform each NPU of the current network topology.

(4) After the topology information update, the interface NPU bootloader downloads the microkernel code from the PC through the UART interface and broadcasts it to the other NPUs in the network. Each NPU starts up the operating system as soon as received. The microkernel is common for each NPU. Depending on the address of the router, the microkernel knows its location. For application code, again NPU0 receives the code since it is the only one connected to the UART. This code is forwarded only to the processor, where it is supposed to execute. Only a single copy of application code exists inside the system.

This method allows to easily scale up the prototype to any size and shape (form factor may be different than 1). It has

Table 3: FPGA synthesis result.

Device	#Slices	#FPGA resources used
NPU	2496	32,50%
Router	683	8,89%
MIPS R3000	1462	19,04%
Other	351	4,57%
Total used	4992	65%

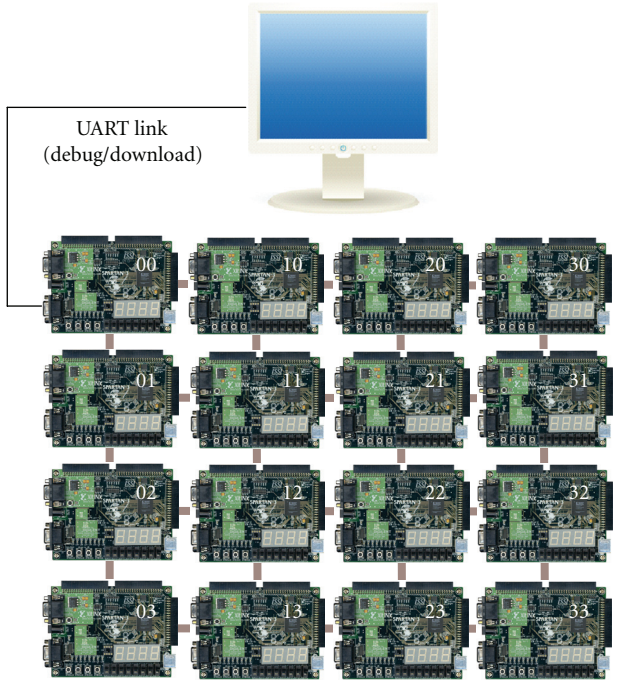


Figure 16: Array of 4 × 4 NPU multiboard.

Table 4: Operating system time and memory costs (KB) size.

Microkernel	10
Com. primitive	5,12
I/O primitive	3,12
DATA section	8,48
STACK section	0,56
OS Size	27,28

been designed for later exploring reliability issues for reconfiguring the system if an NPU becomes unreachable (defective hardware) for instance. In such a case, the faulty NPU can be removed from the table of available processing units.

4.2.2. Kernel Characteristics. Table 4 provides an overview of the memory footprint of our Operating System. In terms of time penalty, each time the OS is invoked (each time a timer interrupt happens), it requires 218 cycles to perform its job. In terms of memory overhead, it requires 27.28 KB. The communication primitives represent almost one fifth of the total memory required by our OS.

4.3. Description of the Experiments. The FPGA platform is the basis of our experiments. Those have been carried out on the HS-Scale system in order to study and characterize the strengths and the weaknesses of our approach.

4.3.1. Set of Applications. We have chosen 3 different applications: a 2-TAP Finite Impulse Response (FIR) filter, a Data Encryption Standard (DES) encoder, and an MJPEG decoder. The main motivation for using such applications was to cover a wide range of possible dataflow applications in terms of granularities and regularities of the tasks. The FIR filter is based on fine grain tasks with a task graph requiring multiple dependencies. Compared to FIR filter, the granularity of DES tasks and MJPEG tasks is coarser. DES tasks are regular and not data dependent, while MJPEG tasks are irregular and depend on the image characteristics. Some dummy applications have been created for better highlighting the capabilities of different policies.

4.3.2. Experimental Protocol. We have developed a set of self-adaptive features for the HS-Scale system: the purpose of this study is to evaluate and to measure the impact of the self-adaptability on application performance. The main metric presented in the next section is the Throughput (TP). It is computed as follows:

$$\mathrm{TP}(\mathrm{KB}\times \mathrm{s}^{-1}) = \frac{\text{Number of Computed Data (KB)}}{\text{Number of Cycles}} \times f(\mathrm{Hz}). \tag{1}$$

Our experiments were performed with each NPU running at $f = 50$ MHz. We have implemented different application scenarios as follows.

(i) Monoprocessor Implementations. Each application of our test set is programmed as a monolithic task (with or without the operating system) in order to calculate a reference throughput.

(ii) Multiprocessor Implementations with Static Mappings. Each application is described as a task graph application. Performing figures for various static mappings have been collected.

There is a certain degree of randomness in the execution of a scenario due to different reasons. Firstly, the communications between routers are asynchronous. Secondly, the execution of a task on a given NPU depends on several varying parameters such as the presence of data in its FIFO (may depend on other tasks placed on different NPUs communicating through the asynchronous network) and the timer interrupts regarding application start that can induce a different scheduling and a different timing in the decision making process. This is the reason why, for (1) and (2), the experiments were repeated 10 times in order to expose the average throughput and its standard deviation.

(i) Multiprocessor Implementations with Dynamic Mappings (Migrations). This case represents our main contribution with self-adaptability features. The studied scenario relies

Table 5: FIR, DES, and MJPEG monolithic implementations and OS cost.

		$-OS^{(a)}$	$+OS^{(b)}$
FIR	Average TP (KB/s)	315.92	313.08
	Standard deviation of TP (Kb/s)	0	0.09
DES	Average TP (KB/s)	6.56	6.54
	Standard deviation of TP (Kb/s)	0	0.06
MJPEG	Average TP (KB/s)	35.39	34.98
	Standard deviation of TP (Kb/s)	0	0.14

[a]Without the operating system.
[b]Without the operating system.

on an application that is sequentially injected on a single NPU which triggers remapping decisions. These remapping decisions are all based on nearest-free neighbor policy where every time the FIFO utilization reached the 80% threshold, a migration was triggered. We have monitored dynamically the throughput and the FIFO utilization ratio in order to plot these metrics as temporal functions.

5. Applications and Results

This section is devoted to the analysis of the self-adaptive results obtained on the FPGA prototype. Three classes of results are exposed: (i) monoprocessor implementations used as reference, (ii) multiprocessor static mappings, and (iii) self-adaptive implementation where tasks freely migrate from NPU to NPU.

5.1. Monoprocessor Study. Each application (FIR, DES, and MJPEG) was programmed as a monolithic task with or without the Operating System. The aim of this study is to evaluate the impact of the use of our OS on performance and also to provide a reference performance for further implementations.

Table 5 summarizes the results. The important information from these results are (1) that the impact of the Operating System on the throughput performance is relatively low (less than 0.95% for each application of our test set) and (2) that it introduces a certain level of randomness (shown by the standard deviation). As a conclusion, the OS provides low-cost multitasking capabilities and implies a very little reduced quality-of-service as the throughput is not deterministic anymore.

5.2. Static Placement Study. Each application of our test set was partitioned into several tasks. Our objective was to distribute the computations of a given application onto several processors in order to evaluate the impact on the throughput. This distribution was hand-made (static placement), and no migration was allowed for discarding the influence of transient phenomena. All references to task placements made throughout this section rely on the addressing mode presented in Figure 16.

5.2.1. Data Encryption Standard. The Data Encryption Standard (DES) algorithm is composed of several computational steps as depicted in Figure 17. All 16 rounds are functionally equivalent but operate with different keys ($K_1 \cdots K_{16}$).

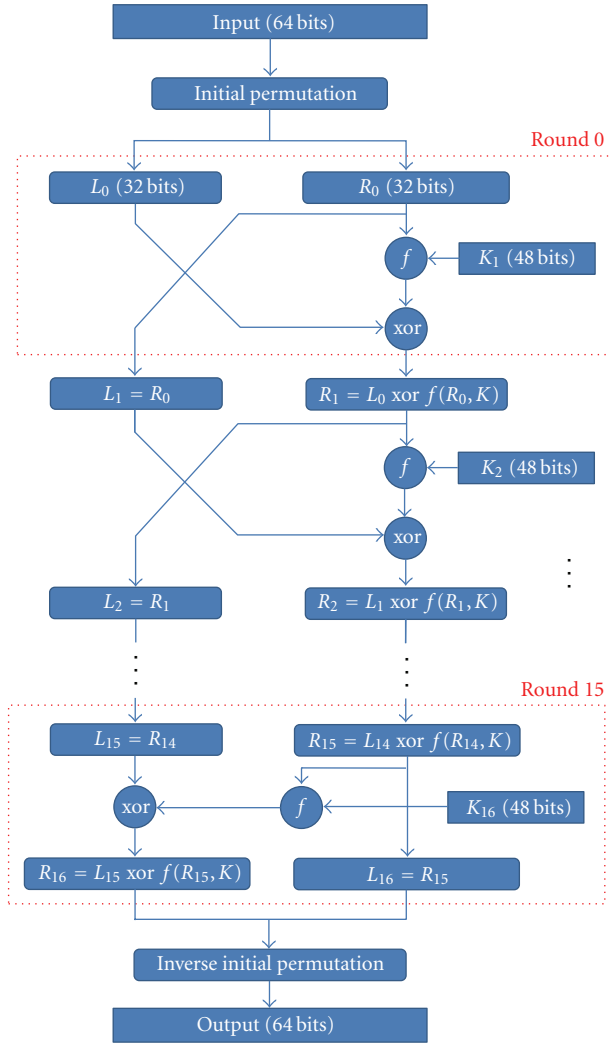


Figure 17: The DES algorithm.

We have chosen to implement it in a pipeline fashion, by decomposing the rounds (16 rounds). Figure 18 shows the DES performance results for different partitioning of the DES algorithm. We devised 4 different pipelines, with 2, 4, 6, and 8 tasks which therefore correspond to tasks embedding 8 to 2 rounds. Due the communication overhead introduced by the task partitioning, we observed that the performance is decreased when running all tasks on the same NPU. Then, when expanding the task graph to other NPUs, we observe that the throughput increases rapidly until it reaches its maximum value when n tasks are mapped to n NPUs. The OS overhead is generally hidden when the n tasks can be mapped to $n/2$ NPU. Finally, the performance improvement of n task partitioning corresponds to n stage pipeline, that is, the reference throughput is at the maximum approximately multiplied by n.

5.2.2. MJPEG Decoder. Figure 19 shows the processing pipeline of a JPEG encoder operating on grey-coded images.

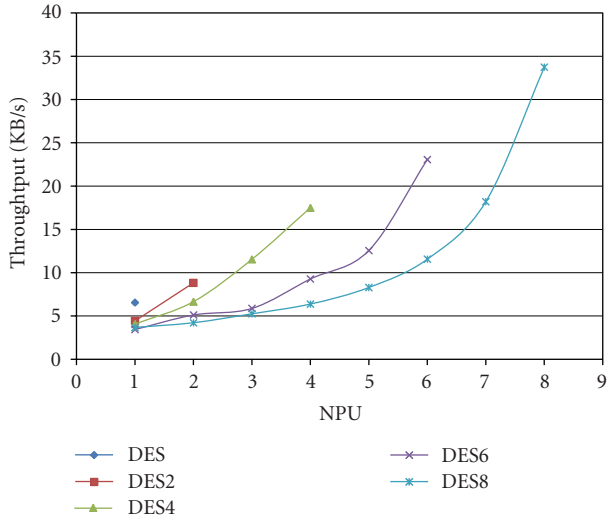


Figure 18: DES performances results with 1, 2, 4, 6, and 8 tasks.

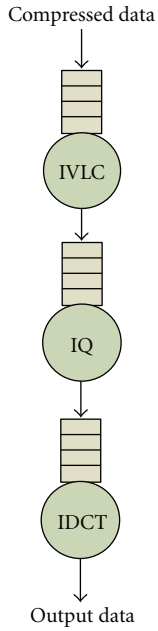


Figure 19: MJPEG data flow.

The first step of this application is the inverse variable length coding (IVLC) which relies on a Huffman decoder. This processing time for that task is data dependent. The two last tasks of the processing pipeline are, respectively, the inverse quantization (IQ) and the inverse discrete cosine transform (IDCT). The atomic data transmitted from task to task is an 8×8 pixel block which has a size of 256 bit. We naturally chose to use a traditional task partitioning as depicted in Figure 19 with both a task-level dataflow description.

Similarly to the FIR and DES applications, the operating system communication primitives induce a performance overhead when the decoder is splitted into 3 tasks (Table 6, column 1) compared to the performance shown in Table 6, column 2 (39.11 KB/second). Distributing the processing of

Table 6: Throughput evolution with task graph expansion.

#NPU	1	2	3
Task placement	3L[(a)]	2L, 1R	3R
Average TP (KB/s)	29.21	39.11	39.05
Standard deviation of TP (Kb/s)	0.21	0.52	0.40

[(a)]The notations L and R refer to "Local" and "Remote" executions.

IVLC on a remote NPU (Table 6, column 3) immediately pays with a significant increase in the throughput. The fully distributed implementation exhibits a very small performance improvement when comparing to a local implementation, which is due, as we will show in the next section, to the fact that a critical task in the processing pipeline already fully employs the processing resources of a given NPU. The standard deviation, as previously observed for DES and FIR applications, increases with task partitioning and distribution.

5.3. Dynamic Placement Study (with Migrations). The aim of this section is to analyze the dynamic behavior of HS-Scale. We will study on the different remapping policies of the transient phenomenon in time, and the impact on performance will be measured.

5.3.1. Diagnostic and Decision Based on Communication Load. The first migration policy corresponds to a percentage of the FIFO utilization threshold (80% in our experiments). In this case, when the software monitor detects that the FIFO is filled over 80% for a given task, this task is automatically moved to another NPU according to a first neighbor policy.

Two scenarios DES4 and MJPEG applications are exposed in this section to prove the validity of this policy, starting from the neighbor on the east.

5.3.2. DES4. In this scenario, we used the DES application partitioned into 4 tasks: task 1 (T1) corresponds to the rounds 1 to 4, task 2 (T2) corresponds to the rounds 5 to 8, task 3 (T3) corresponds to the rounds 9 to 12, and task 4 (T4) to rounds 13 to 16. Figure 20 depicts the measured throughput of the application with the normal policy.

During the first seconds, the tasks (T1, T2, T3, T4) are manually sent from the external PC and placed on the NPU(1, 1). At t1 = 25.43 seconds, the DES starts the computation: the four tasks are running sequentially on the same NPU. Three migrations are performed by the policy in less than 200 milliseconds; the FIFO levels of T1, T3, and T4 are above the threshold at times $t = 26.21$ seconds, $t = 27.02$ seconds, and $t = 27.78$ seconds, respectively. After the last migration, we can see an increase of the throughput that also causes a decrease T3 FIFO filling. The throughput then stabilizes around 17 KB/s.

Comparing the performance of the last placement with the static mapping presented previously, we observe identical figures; this policy has rapidly found (1.59 seconds) one of the best placement. In order to better observe the evolution of performance in the different steps, results presented in

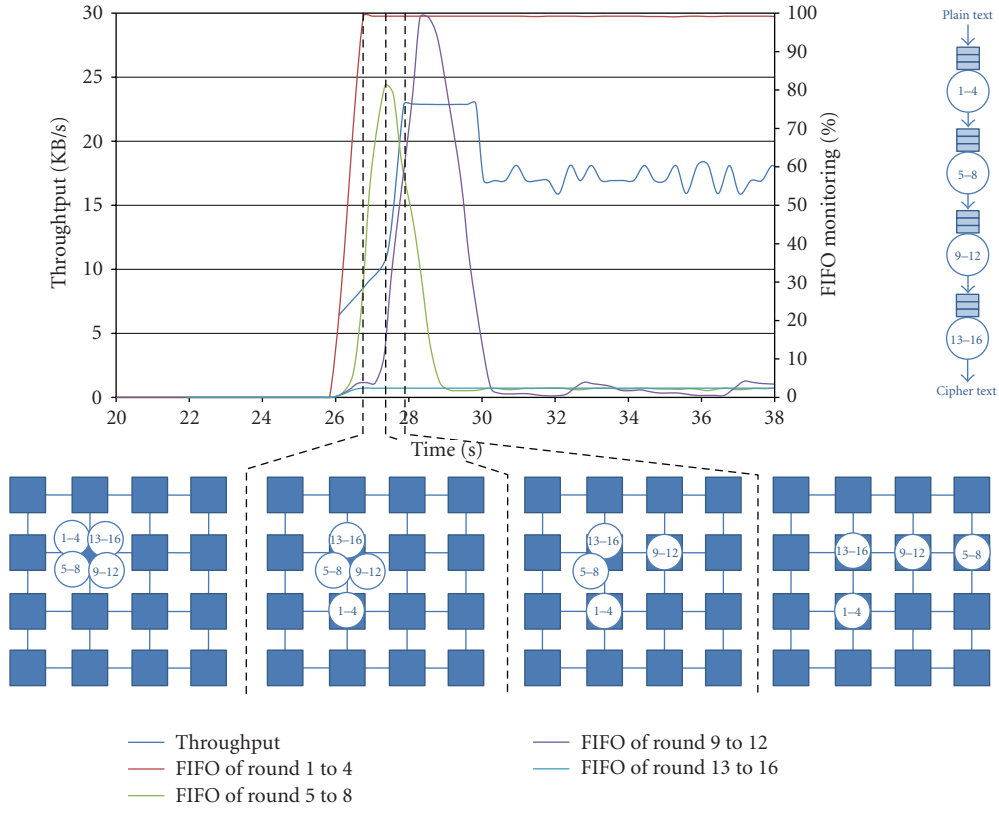


Figure 20: DES4 execution in time without delay.

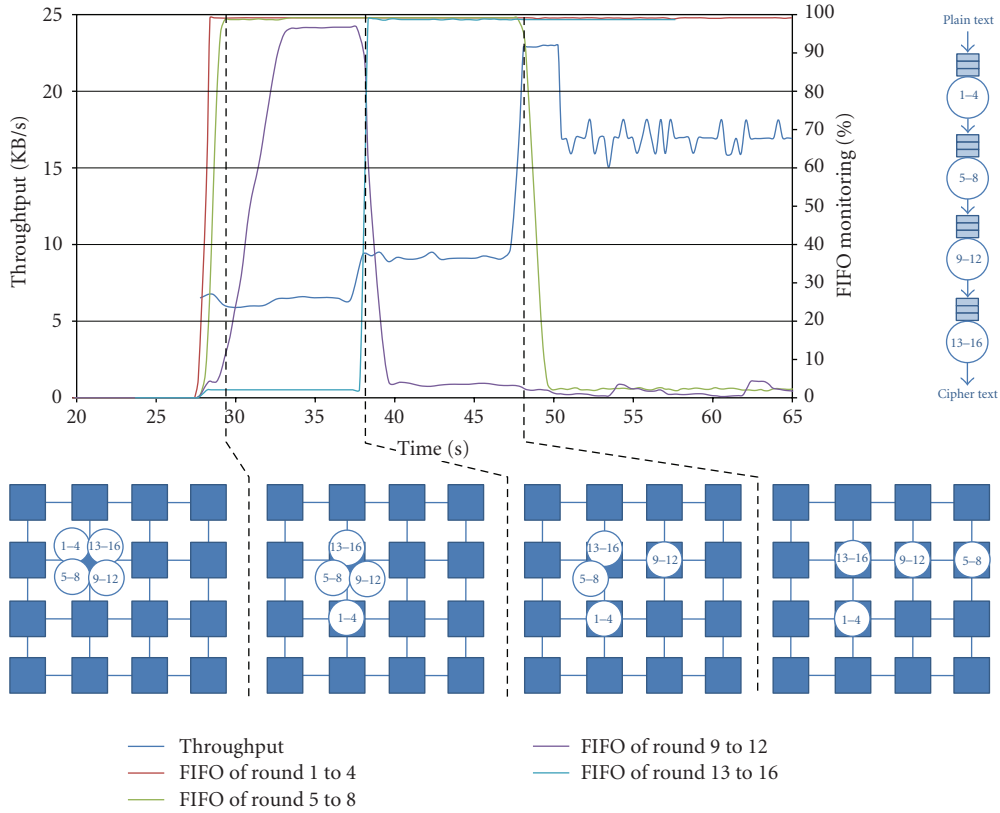


Figure 21: DES4 execution in time with delay.

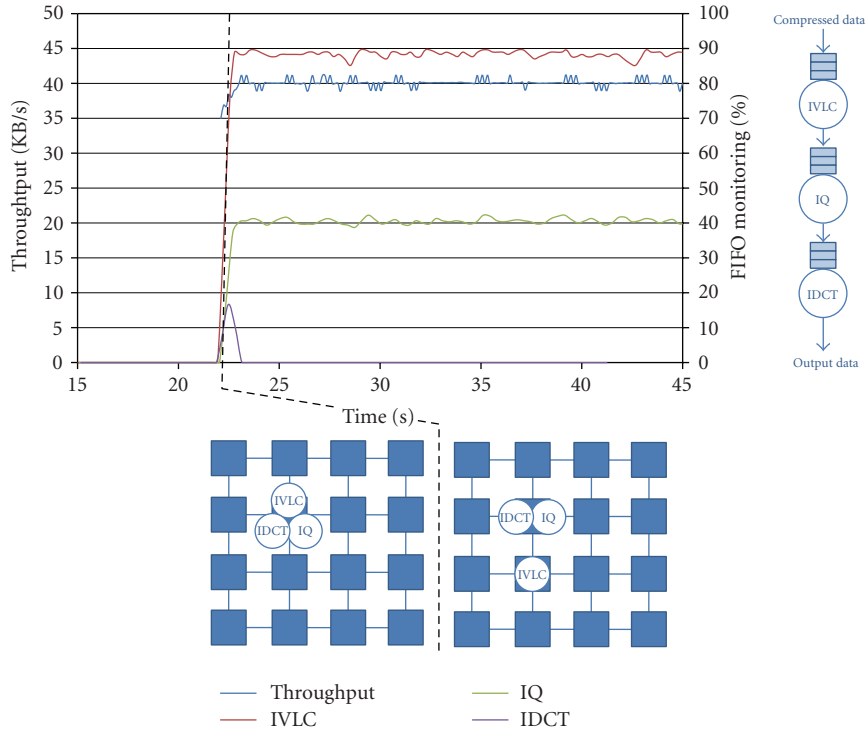


Figure 22: MJPEG execution in time.

Figure 21 were implemented using a policy that allows one migration every 10 seconds at most.

At the beginning after the start of the DES application at t1 = 28.71 seconds, all tasks are running in the same NPU, and the throughput of the application is around 4 KB/s. At t2 = 29.02 seconds, the FIFO monitor of T1 indicates a FIFO usage greater than 80%: it then begins automatically to move the task (i.e., (1) waiting a migration point, (2) looking for a free NPU, (3) migrating the task, and (4) restoring the context). Due to the migration policy, the task is placed on the NPU(1, 2). Then, the throughput stabilizes around 6.5 KB/s. After this migration, FIFO fillings of T2 and T3 increase quickly above 80% because T1 shows a higher throughput due to the fact it benefits from an entire NPU. As soon as the policy re-enables migrations (t3 = 39.03 seconds), T3 migrates on NPU(2, 1). FIFO usage of T4 then increases because of the time-multiplexing execution which results ten seconds later into another migration (t4 = 49.04 seconds).

The same final state is observed as previously with an average throughput of 17 KB/s. This clearly demonstrates for the DES application that such a simple policy is capable of rapidly converging to a best placement.

5.3.3. MJPEG. In this scenario, we use the MJPEG decoder application partitioned into three tasks: IVLC, IQ, and IDCT. Figure 22 depicts the application throughput and the FIFO usage of each task in the pipeline.

During the very first seconds, all tasks are instantiated manually on the NPU(1, 1). From t1 = 21.35 seconds to t2 = 22.26 seconds, these tasks are executed sequentially on the same NPU which provides an average throughput of 30 KB/s. At t2, the IVLC FIFO reaches a value greater than 80%: this leads to a migration process that involves the following steps.

(i) Freezing task execution and search for a free NPU (3.11 milliseconds).

(ii) Migrating the task (14.05 milliseconds).

(iii) Restoring the context (3.32 milliseconds).

Due to the migration policy, the task is moved from NPU(1, 1) to NPU(1, 2): it takes 20.48 milliseconds for the whole task migration process. During this time, the OS consumes CPU time for the migration process which decreases the application throughput. After the migration completion, the average throughput reaches 40 KB/s: it takes 153.6 milliseconds until a performance benefit is observed.

We then observe the following behavior of the system: on one hand, the IQ FIFO utilization remains stable meaning that it has just enough CPU time to process its data, and on the other hand the IDCT FIFO decreases meaning that it has enough CPU time to process all the data in its FIFO. In this situation, the mapping is stable from the migration policy point of view.

This clearly suggests that the policy is adequate with respect to the optimization of performance: as seen previously in Table 6 since IVLC proves the most time consuming task, therefore any further migration would not help improving performance.

5.3.4. Diagnostic and Decision Based on CPU Workload. The example on Figure 23 shows the results for this migration policy based on the CPU workload. The experimental

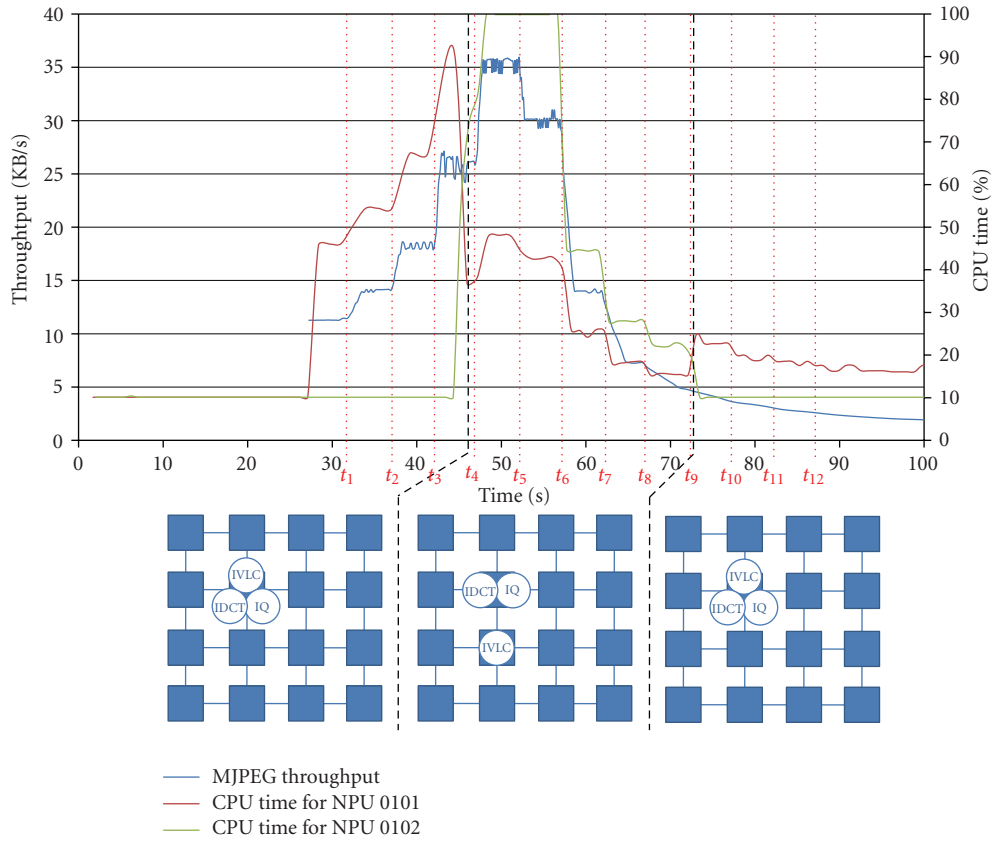


Figure 23: MJPEG decoder throughput based on CPU workload.

protocol used for these results relies on varying the input data rate for observing how the system adapts.

At the beginning all tasks (IVLC, IQ, and IDCT) are running on the same NPU(1, 1), but the input throughput on the MJPEG application is lower so the CPU time consume is around 47%. At each step t1, t2, and t3 the input throughput is increased, so we can see an increase of the CPU time consumed step by step. When the CPU time used exceed the threshold (i.e., 80%), the operating system detects that the NPU(1, 1) is overloaded (at 45 seconds) so it decided to migrate the task which uses the most of CPU time on a neighboring NPU. In this example IVLC task migrates on NPU(1, 2) which decreases the CPU time used by NPU(1, 1) around 35% and an increase of CPU used by NPU(1, 2) around 80%. At t4 the input throughput increases more which leads to an MJPEG throughput increase around 35 KB/s and overloaded the NPU(1, 2) at 100% but migration is not triggered because just one task is compute.

From t5 to t12 the input throughput of the MJPEG application is decreased step by step, and when the CPU time of NPU(1, 2) is less than 20% (at 72 seconds), the operating system decides to closer task on the same NPU (the NPU(1, 1)). We can see after this migration a decrease of CPU time used by the NPU(1, 2) and an increase of CPU time used by NPU(1, 1) but without saturate it.

We can observe that MJPEG application performances are lower than in the static mode; this is because the operating system uses more CPU time (around 10%) to monitor CPU time and average sample.

5.3.5. Diagnostic and Decision Based on Locality. The third migration policy corresponds to optimizing placement for decreasing Manhattan distance [23] (number of hops) between communication tasks: whenever the software monitor detects a closer placement of a given task, this task is automatically migrated to this NPU.

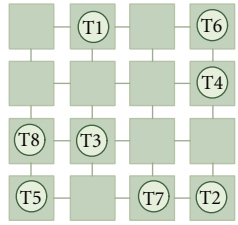
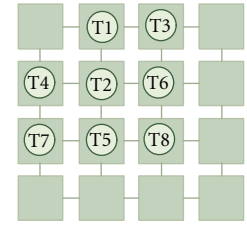
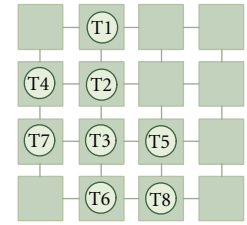
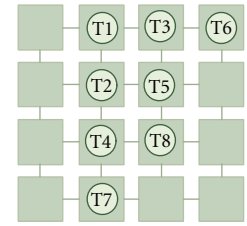
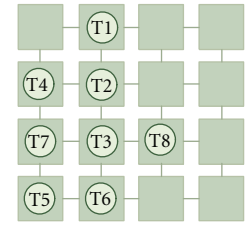
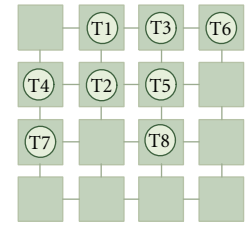
To prove the validity of this policy, we will expose results on one scenario with a dummy application that is made of numerous communicating tasks.

In this scenario, we use a dummy application which generates intensive traffic between communicating tasks. Each task consumes a different value of CPU time to compute this data. The graph task of this dummy application is presented on Figure 24.

Table 7 presents the results obtained with the policy based on the closest placement. For four initial placements, we have executed five simulations and observed the final placements obtained. This table summarizes these experiments and gives the initial placement, the final placements as well as the cumulated distance for these.

The best obtained placements exhibit 13 hops which is a satisfying result with respect to the best placement presented previously. These placements were furthermore obtained with a limited number of migrations (around 10) which suggests that deriving policies which would take into account

Table 7: Dummy application with closest placement policy.

Initial placement		Number of migrations	Final placement	
Number of hops between tasks	Graph		Number of hops between tasks	Graph
31		10	13	
		7	14	
		7	13	
		6	18	
		10	13	

more parameters or even take sub-optimal decisions for avoiding local minima could help further improving these results.

6. Conclusions

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

MPSoCs are certainly the natural target for bringing these techniques into practice: provided they comply with

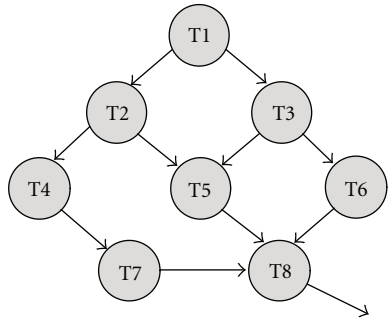


Figure 24: Dummy task graph.

some design rules they may prove scalable from a performance point of view, and since they are in essence distributed architectures, they are well suited to locally monitoring and controlling system parameters.

From the software point of view, the system presented in this paper relies on a tiny operating system that is used on every processing element. This decision certainly helps in improving system scalability as the adaptability policies that were proposed rely on a purely decentralized decisionmaking approach. The three presented policies suggest that although distributed and operating on either local or possibly non up-to-date system information, performance versus static scenarios are comparable. Finally we also demonstrated that such as solution could be viable even considering economic constraints such as silicon cost: the overhead incurred by this approach has been quantified in term of logic and memory thanks to the realization of the 16 processors prototype which is fully functional.

This paper describes an adaptive MPSoC framework that utilizes a message passing programming model. Based on information implemented in the form of distributed software monitors, the user can specify migration policies that enable the architecture to refine task placement. The conducted experiments show the following.

(i) Migration helps better balancing processor load at run-time for achieving better performance.

(ii) Task migration incurs a minimal performance overhead.

The important characteristics that have been considered are mostly flexibility, scalability, and adaptability. The remapping policies adopted show that it is possible to have a very flexible, scalable, and self-adaptable MPSoC by using monitoring systems not complex which does not affect the overhead of the system.

The proposed architecture relies on the following design decisions.

(i) Decentralized control.

(ii) Homogeneous array of processing elements.

(iii) Distributed memory.

(iv) Scalable NoC-style communication network.

Although such principles have so far demonstrated the interest in the sole context of performance, it suggests that other benefits could be achieved such as power management where highly communicating task could be mapped to neighbor processors whereas this constraint is relaxed for others. Future work aims at exploring these functionalities in connection with voltage and frequency scaling techniques since these approaches combined could reveal useful for better balancing performance and power consumption.

Another promising perspective of this work relies on task replication which helps further taking advantage of processing resources whenever needed. Although this is applicable for certain applications only, our first experiment on the MJPEG application shows that only critical task gets replicated with similar mapping/replication policies as the one presented in this paper. This should prove particularly useful for tasks which data dependant processing load such as the IVLC in our case.

Although this has not yet been demonstrated, the structure also intrinsically supports fault tolerance to a certain degree since each processor is aware of all others in the system. A faulty processor identified as such using mutual testing techniques could be discarded from the list of functional units, and further mapping decisions would therefore target the remaining processors only. Extending the control to some different system parameters such as processing elements frequency and supply voltage would certainly help further improving observed performance. Among the considered possible strategies, task replication is certainly of significant interest as it would help better matching the number of processing resources to the performance demand. This technique would prove applicable in some restricted domains (mostly for dataflow applications) but could probably be extended.

Similarly, for the growing concern of fault tolerance the developed techniques could here constitute a viable solution; since the system is entirely distributed, faulty units could be identified by others and therefore functionality would rely on the remaining processing units.

Finally, we think that adaptability is an approach that will in the near future be widely adopted in the area of MPSoC. Not only because of the here mentioned limitations such as technology shrinking, power consumption, and reliability but also because computing undoubtedly goes pervasive. Pervasive or ambient computing is a research area on its own and in essence implies using architectures that are capable of self-adapting to many time-changing execution scenarios. Examples of such applications range from ad-hoc networks of mobile terminals such as mobile phones to sensor networks systems aimed at monitoring geographical or seismic activity. In such application fields, power efficiency, interoperability, communication and scalability are primary concerns such systems have to cope with many limitations such as limited power budget, interoperability, communication issues, and finally, scalability.

References

[1] D. E. Culler, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 1st edition, 1998.

[2] W. M. Collier, *Reasoning about Parallel Architectures*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1992.
[3] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948–960, 1972.
[4] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*, O'Reilly, Sebastopol, Calif, USA, 1996.
[5] "The OpenMP API specification for parallel programming," http://openmp.org/wp.
[6] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, Scientific and Engineering Computation Series, MIT Press, Cambridge, Mass, USA.
[7] G. Kahn, "The semantics of a simple language for parallel programming," in *IPIP Congress*, pp. 471–475, 1974.
[8] M. Saldana and P. Chow, "TDM-MPI: an MPI implementation for multiple processors across multiple FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–6, Madrid, Spain, August 2006.
[9] J. Kohout and A. D. George, *A High-Performance Communication Service for Parallel Computing on Distributed DSP Systems*, Elsevier Science, Amsterdam, The Netherlands, 2003.
[10] J. J. Murillo, D. Castells-Rufas, and J. C. Bordoll, "HW-SW framework for distributed parallel computing on programmable chips," in *Proceedings of the Conference on Design of Circuits and Integrated Systems (DCIS '06)*, 2006.
[11] "MPCore Linux 2.6 SMP kernel and tools," ARM Limited, http://www.arm.com/products/os/linux_download.html.
[12] A. Barak, O. La'adan, and A. Shiloh, "Scalable cluster computing with MOSIX for Linux," in *Proceedings of the Linux Expo*, pp. 95–100, Raleigh, NC, USA, May 1999.
[13] J. Robinson, S. Russ, B. Heckel, and B. Flachs, "A task migration implementation of the message-passing interface," in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC '96)*, p. 61, 1996.
[14] A. R. Dantas and E. J. Zaluska, "Improving load balancing in an MPI environment with resource management," in *High-Performance Computing and Networking*, pp. 959–960, Springer, Berlin, Germany, 1996.
[15] L. Chen, C.-L. Wang, F. C. M. Lau, and K. K. Ricky, "A grid middleware for distributed Java computing with MPI binding and process migration supports," *Journal of Computer Science and Technology*, vol. 18, no. 4, pp. 505–514, 2003.
[16] S. Carta, M. Pittau, A. Acquaviva, et al., "Multi-processor operating system emulation framework with thermal feedback for systems-on-chip," in *Proceedings of the 17th Great Lakes Symposium on VLSI (GLSVLSI '07)*, pp. 311–316, Stresa-Lago Maggiore, Italy, March 2007.
[17] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 1, pp. 1–6, Munich, Germany, March 2006.
[18] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: exploring the multi-processor SoC design space with systemC," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 41, no. 2, pp. 169–182, 2005.
[19] M. Pittau, A. Alimonda, S. Carta, and A. Acquaviva, "Impact of task migration on streaming multimedia for embedded multiprocessors: a quantitative evaluation," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '07)*, pp. 59–64, October 2007.
[20] MIPS corp., http://www.mips.com.
[21] F. Moraes, et al., "Hermes: an infrastructure for low area overhead packet-switching networks on chip integration," *VLSI Journal*, vol. 38, pp. 69–93, 2004.
[22] J. R. Levine, *Linkers and Loaders*, Morgan Kaufmann, San Francisco, Calif, USA, 1999.
[23] P. E. Black, "Manhattan distance," http://www.itl.nist.gov/div897/sqg/dads.