FASTER: Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration

D. Pnevmatikatos [a,*], K. Papadimitriou [a], T. Becker [b], P. Böhm [b], A. Brokalakis [h], K. Bruneel [c], C. Ciobanu [d], T. Davidson [c], G. Gaydadjiev [d], K. Heyse [c], W. Luk [b], X. Niu [b], I. Papaefstathiou [h], D. Pau [g], O. Pell [f], C. Pilato [e], M.D. Santambrogio [e], D. Sciuto [e], D. Stroobandt [c], T. Todman [b], E. Vansteenkiste [c]

[a] Foundation for Research and Technology-Hellas, Heraklion, Greece
[b] Imperial College London, London, UK
[c] Ghent University, Ghent, Belgium
[d] Chalmers University of Technology, Göteborg, Sweden
[e] Politecnico di Milano, Milan, Italy
[f] Maxeler Technologies, London, UK
[g] STMicroelectronics, Agrate, Italy
[h] Synelixis, Chalkida, Greece

ARTICLE INFO

ABSTRACT

The FASTER (Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration) EU FP7 project, aims to ease the design and implementation of dynamically changing hardware systems. Our motivation stems from the promise reconfigurable systems hold for achieving high performance and extending product functionality and lifetime via the addition of new features that operate at hardware speed. However, designing a changing hardware system is both challenging and time-consuming.

FASTER facilitates the use of reconfigurable technology by providing a complete methodology enabling designers to easily specify, analyze, implement and verify applications on platforms with general-purpose processors and acceleration modules implemented in the latest reconfigurable technology. Our tool-chain supports both coarse- and fine-grain FPGA reconfiguration, while during execution a flexible run-time system manages the reconfigurable resources. We target three applications from different domains. We explore the way each application benefits from reconfiguration, and then we asses them and the FASTER tools, in terms of performance, area consumption and accuracy of analysis.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Extending product functionality and lifetime requires constant addition of new features to satisfy the growing customer needs and the evolving market and technology trends. Software component adaptivity is straightforward, but in many cases it is not enough. Recent products incorporate hardware accelerators to satisfy performance and energy requirements. These accelerators also need to adapt to the new requirements. Reconfigurable logic allows the definition of new functions to be implemented in dynamically instantiated hardware units, combining adaptivity with hardware speed and efficiency. However, designing a hardware system that changes over time is a challenging and time-consuming task.

We propose a methodology enabling designers to easily implement applications on platforms with one or more general-purpose processors and multiple acceleration modules implemented in reconfigurable hardware. Our main contribution is that we introduce partial reconfiguration from the initial design stage all the way down to the runtime use of the system. Fig. 1 depicts the tool-chain of FASTER project [1]. Its input is the description of the application in a high-level programming language; the initial decomposition into tasks is described in OpenMP. The corresponding task graph is then partitioned in space and time to identify candidates for reconfiguration. FASTER supports coarse-grain reconfiguration and fine-grain reconfiguration. The former allows for swapping the hardware modules identified at design time as reconfigurable ones in/out of FPGA regions; this is called region-based reconfiguration. The latter allows for reconfiguring small parts of the FPGA with circuits synthesized at run-time; this technique is called micro-reconfiguration and enables the creation of specialized circuits containing infrequently changing parameters. We also address the verification of static and dynamic aspects of
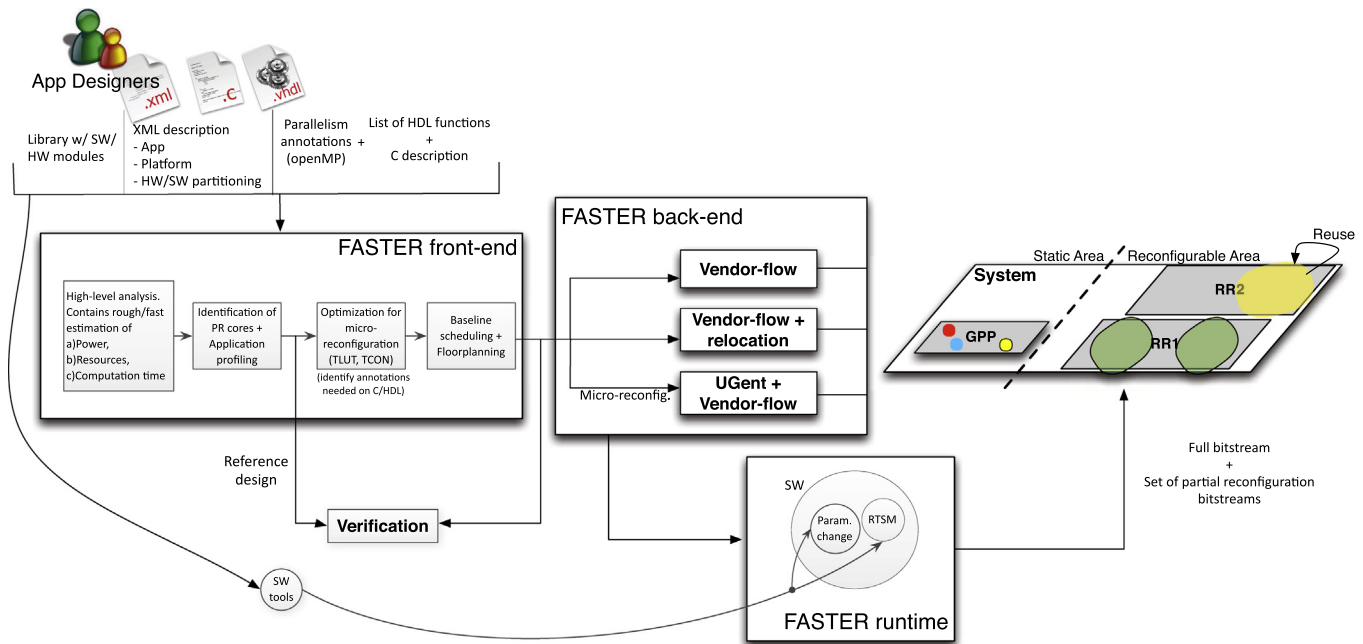
Fig. 1. Abstract view of FASTER tool-chain.

a reconfigurable design while minimizing run-time overheads on speed, area and power consumption. Finally, we developed a run-time system for managing the various aspects of parallelism and adaptivity of an application by taking into account the run-time availability of reconfigurable resources. To test our methodology we employ applications from the embedded, desktop and high-performance computing domains, using as metrics performance, area consumption and accuracy of analysis at an early stage of the development cycle.

This paper extends our previous work in [2], providing more details on all the parts of FASTER project. It is structured as follows: Section 2 overviews previous works on methods and tools for reconfigurable system design, and exposes the relevance and novelty of the FASTER project. In Section 3 we delve into the details of the front-end tool-chain by presenting the connections between the different stages, and discussing the XML exchange format. Section 4 describes briefly region-based reconfiguration supported by FPGA vendors, and extends our previous work on micro-reconfiguration with reconfiguration of routing and with a profiler to assist the designer prior to taking decisions. Section 5 discusses our verification approach, and Section 6 presents the runtime system operation and its input requirements at design- and run-time. Section 7 discusses the target applications coming from the industrial side, the way we design them to explore reconfiguration capabilities, their performance evaluation, and evaluates some of our tools. Section 8 summarizes our contributions, and Section 9 concludes the paper.

2. Related work and motivation

Reconfigurable computing has been extensively studied in the academic literature. The authors in [3] presented a survey covering reconfigurable architectures and design methods. The FASTER project targets two system-level architectures; the stand-alone reconfigurable logic, and the organization that embeds the processor in the reconfigurable fabric. Regarding design methods, we focus on run-time customization using partial reconfiguration. We developed a run-time system to hide low-level system details from the designer, which handles scheduling, placement, and

communication with the reconfiguration port. In addition, the work in [3] identified the compilation tool-chain as an important challenge in reconfigurable systems. Towards this direction, we propose tools that assist the designer in deciding which part of the application should be mapped to the reconfigurable hardware and when reconfiguration should occur. In specific, we offer a semi-automatic approach to leverage designer's familiarity with the application to ensure high quality of results.

A detailed survey in [4] summarizes research on compilation techniques for reconfigurable architectures and categorizes them based on their features and target platforms. The authors focus mainly on High Level Synthesis (HLS) tools both for fine- and coarse-grained reconfigurable hardware. Furthermore, they present a generic compilation flow, highlighting the interaction between the various transformations and optimization stages, and discussing hardware-software partitioning. That work categorizes relevant research according to the supported programming languages and the intermediate representations. Also, it provides a classification of the most important code transformations at different levels (bit, instruction, loops, etc.), as well as insights in both temporal and spatial partitioning techniques. Temporal partitioning is relevant in the context of dynamic partial reconfiguration, using time-multiplexing when hardware resources are insufficient while minimizing the reconfiguration overhead. In addition, the authors state that the adoption of reconfigurable computing is limited by the cumbersome process of programming these platforms. Similarly, the work in [5] points out the lack of adequate development methodologies and EDA tools for reconfigurable systems. Our work address this issue by providing easy-to-use and flexible tools.

Further information related to reconfigurable architectures and devices, application development and tools is discussed in [6], while [7] studies the above aspects concentrating on dynamically reconfigurable systems only.

Different frameworks have been proposed to address the concurrent development of architecture and application for heterogeneous systems. For example, Ptolemy [8] is an environment for simulating and prototyping heterogeneous systems with mechanisms for modeling multiple abstraction levels and heterogeneous mixtures of models of computation. Daedalus [9] is a system-level

design framework, composed of several tools that range from automatic parallelization for Kahn Process Networks to design space exploration of both the architectural and platform levels, and to the synthesis of the candidate platform architecture. The different components are interfaced through XML files. Within hArtes project [10] an integrated methodology from the automatic parallelization to the generation of heterogeneous systems was developed, but without considering the reconfiguration aspects. FASTER extends this approach by adopting different estimation tools and partitioning algorithms (interfaced with XML files), while the task partitioning of the input application is specified by means of OpenMP pragmas proposed by [11] as produced by the partitioning methods in hArtes. Another framework for programming heterogeneous platforms is OpenCL [12], an open royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices.

EU-funded projects such as hArtes [10], REFLECT [13], ACOTES [14], ANDRES [15], and Morpheus, conducted research on the necessary stages of a tool-chain and addressed similar issues with FASTER, but they focused more on system-level or architectural aspects of reconfiguration. Moreover, they do not explicitly emphasize on the design and runtime aspects of partial and dynamic reconfiguration, or, on choosing the best reconfiguration grain-size. On the contrary, we introduce partial and dynamic reconfiguration from the initial design of the system all the way down to its runtime use.

To the best of our knowledge, the existing approaches do not abstract from the designer complex manipulations needed to control effectively hardware accelerators, in particular when these are designed as dynamically reconfigurable modules. Towards this direction, we aim at providing a general formulation capable to deal with different multiprocessor systems and different hardware implementations for the tasks (also by exploiting micro-architectural optimizations), and proposing a tool-chain that efficiently supports partitioning of the application, while performing exploration on the possible solutions for the problem. In addition, we consider reconfiguration from the early stages of the design process, hiding most of the implementation details from the user.

3. The FASTER front-end

The present Section discusses the discrete stages and the way we interconnected them to form the FASTER tool-chain. Then it explains the structure of the XML exchange format we use to pass information amongst the stages.

3.1. The front-end tool-chain

The input of the FASTER front-end is an application in C – gcc C11 – whose initial decomposition is described with OpenMP pragmas, and an XML file containing information about the target architecture, such as the number of HW/SW processing elements, characteristics of reconfigurable regions, and the different implementations of hardware accelerators. The corresponding task graph is partitioned to determine which processing element will execute each application task. Every hardware task is treated as a static IP core, a region-based reconfigurable module, or a micro-reconfigurable module. We do not focus on the automatic generation of the HDL implementations for the hardware cores; these are provided by the user either using traditional HDL design or high-level synthesis tools. We target systems with partially reconfigurable FPGAs, either in a single-FPGA or a multi-FPGA environment. In order to support the analysis for region-based and micro-reconfigurable actions we include additional steps. Note that the solution is represented by the mapping of each task not only to a processing

element, but also to one of its available implementations; this allows for exploring alternative trade-offs between performance and usage of resources. The methodology is outlined in Fig. 2, and is organized in four phases: *Application profiling and identification of reconfigurable cores*, *High-level analysis*, *Optimizations for region- and micro-reconfiguration*, and *Compile-time scheduling and mapping onto reconfigurable regions*.

The *Application profiling and identification of reconfigurable cores* based on the initial source code of the application and the description of the target architecture, decomposes the application into tasks and assigns them to the different components of the architecture. It can also use information about the performance of the current tasks, and feedback after the execution of the schedule, e.g. how the partitioning affects the computed schedule, in order to iterate and improve gradually the solution. In addition, it determines (i) the proper level of reconfiguration, i.e. none, region-based, or micro-reconfiguration, for each of the hardware cores by including different analyses (either static or dynamic), and (ii) the properties of the identified tasks, such as the frequency of call functions, the frequency of micro-reconfiguration parameter change, the resources required for each implementation, and the execution performance.

The scope of the *High-level analysis* phase is to explore various implementation options for applications (or parts of applications) that target reconfigurable hardware and to automatically identify opportunities for run-time reconfiguration. The analysis is based on an application description in the form of a hierarchical Data Flow Graph (DFG), application parameters such as input data size, and physical design constraints such as available area and memory bandwidth. The high-level analysis relies on DFGs for functions to estimate implementation attributes such as area, computation time and reconfiguration time, in order to avoid time-consuming iterations in the design implementation process. The hierarchical DFG contains function DFGs to represent algorithm details in application functions. For a function DFG, arithmetic nodes are mapped as data-paths, and data access nodes are mapped as memory architectures. The area and the bandwidth attributes are estimated based on the mapped nodes. The computation time is calculated by relying on data-path performance and data size, and the reconfiguration time is calculated using area consumption and reconfiguration throughput. These estimations are used as input to the previous processing step, i.e. *Application profiling and identification of reconfigurable cores*, to perform design optimizations including arithmetic operations presentation, computational precision, and parallelism in the implementation. Compared with Design Space Exploration (DSE) process, our high-level analysis relies on hardware design models to actively estimate design properties. Once function properties are estimated, the high-level analysis examines the interaction between application functions to suggest opportunities for reconfiguration. Application functions are partitioned into several reconfigurable components, to separate functions that are active at different time. Inside a reconfigurable component, previously idle functions are removed. This can increase the throughput while using the same area, or, reduce the area while providing the same throughput. As depicted in Fig. 2,, high-level analysis interacts with *Application profiling and identification of reconfigurable cores*, as it provides key information for the overall design partitioning and hardware/software co-design process. Several iterations of these two processing steps might be needed. Information between them is exchanged through an XML file.

The third phase, the *Optimizations for region- and micro-reconfiguration*, receives the descriptions of the tasks, i.e. source code, that could benefit from the reconfiguration and produces new and optimized implementations for them to be considered during task mapping. This analysis also profiles the application tasks to determine the slow-changing parameters for the micro-reconfiguration.
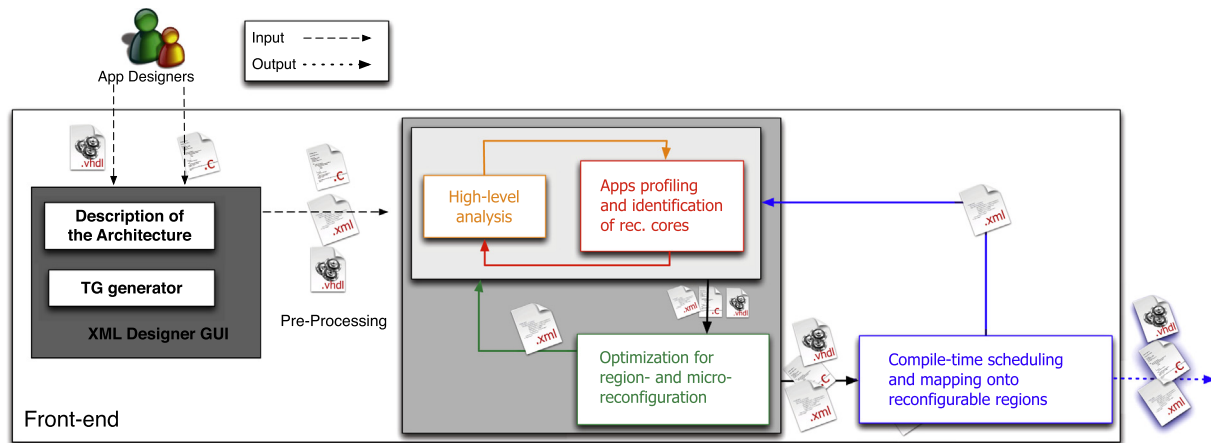
Fig. 2. Front-end of tool-chain.

Finally, the *Compile-time scheduling and mapping onto reconfigurable regions* phase receives information about the application and the architecture from the two previous processing steps, focusing on the tasks assigned to the reconfigurable hardware and it determines the task schedule, along with the mapping of the cores onto the reconfigurable regions. It also determines the number and characteristics of these regions, e.g. size, the number and size of each input/output point, and also takes into account the interconnection infrastructure of the system, e.g. bus size. Finally, it annotates the tasks with information about the feasibility of the implementation where the solution is specified (i.e. if the reconfigurable region can satisfy the resource requirements), and it provides feedback to the partitioning methodology to further refine the solution.

3.2. The XML exchange format

We adopt an exchange format based on the Extensible Markup Language (currently XML v2.0 is supported), which allows to easily integrate the different methodologies developed in parallel, as well as the manual decisions performed by the designer. Below we describe the interfaces between the different activities. The FASTER XML has a modular format and contains four independent but related sections that can be processed by different modules:

- tag `<architecture>`: the architecture is defined here in advance, at least in terms of the number of processing elements and area dedicated to hardware cores, either reconfigurable or not. Communication architecture and memory hierarchy are also provided here;
- tag `<application>`: this part describes high-level information about the application, e.g. source code files, profiling information, workload or data input characterization, without any connection with the architecture or its implementation;
- tag `<library>`: it contains the available implementations for the application tasks, along with the performance and corresponding resource requirements. It takes into account the implementations derived from the application, e.g. produced by the partitioning methodology along with the high-level analysis methods, and the ones from external sources, e.g. obtained through external tools and provided by the designer to the methodology;
- tag `<partitions>`: the structure of the partitioned application in terms of tasks and data transfers, along with the corresponding static schedule and mapping solutions, both for hardware and software tasks.

The architecture and application parts are independent of each other, while the library brings together information from the partitions (this tag contains info on the tasks) and the architecture (this tag contains info on the processing elements) by means of the description of the available implementations. Fig. 3 highlights how the different parts of the FASTER tool-chain interact through the XML file format; it reflects to the first two processes of Fig. 2 and illustrates how different parts of the XML file structure are analyzed, generated or updated:

- *Application analysis and profiling*: it corresponds to the analysis performed on the initial application code. It includes the profiling of the call graph and the function call parameters to improve the HW/SW partitioning and the identification of cores that can benefit from micro-reconfiguration.
- *Partitioning and optimization*: it includes the HW/SW partitioning stage, along with the optimization of the task implementations, especially for exploiting micro-reconfiguration.
- *High-level analysis*: it produces estimates for hardware implementations of tasks such as area and computation time. This is based on analyzing the application, its input data and design constraints. The estimates are used for partitioning and optimization.
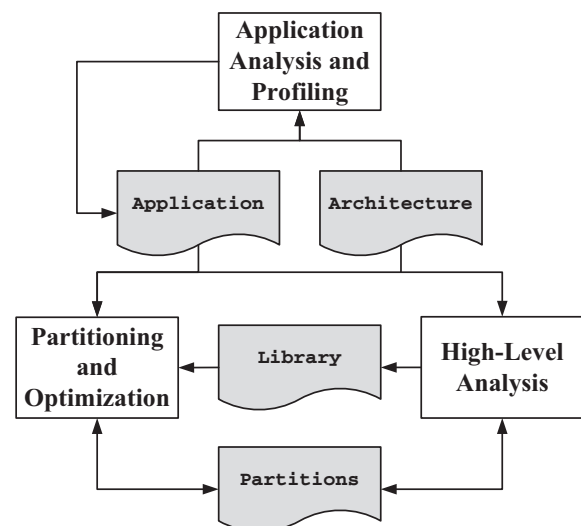

Fig. 3. Interaction between the initial steps of the FASTER tool-chain through the XML file structure.

4. Region-based and micro-reconfiguration support

FASTER supports both region-based and micro-reconfiguration. Each of these two options offers advantages in different conditions.

4.1. Region-based reconfiguration

Region-based reconfiguration describes the concept of instantiating a new function in a particular region of the FPGA. The generation of configuration bitstreams takes place at design time. The designer marks a certain functionality as reconfigurable and confines its logic to a dedicated region on the FPGA by means of floorplanning. This is shown in Fig. 4a, while Fig. 4b illustrates that a number of different reconfigurable functions can be implemented targeting the same region. This region can be reconfigured at run-time with the desired functionality while the rest of the chip remains operational. An FPGA design can contain multiple reconfigurable regions, and in general, reconfigurable functions are loaded only into the region they were originally implemented for.

The challenge when designing such systems is identifying functionality that can be reconfigured, effectively allocating them to dedicated regions and floorplanning the entire design. The problem of floorplanning in the domain of partially reconfigurable FPGAs steadily attracts the interest of researchers [16,17]. In FASTER project we use the floorplanning published in [18], while for supporting region-based reconfiguration we rely mainly on vendor's tools.

4.2. Micro-reconfiguration

One or more of the aforementioned regions can also be reconfigured in a finer granularity to implement Runtime Circuit Specialization (RCS) [19,20]. RCS is a technique for specializing an FPGA configuration at runtime according to the values of a set of parameters. The main idea is that before a task is deployed on the FPGA, a configuration that is specialized for the new parameter values is generated. Specialized configurations are smaller and faster than their generic counterpart, hence RCS can potentially result in a more efficient implementation. Currently, the design tools of FPGA manufacturers support region-based reconfiguration only, where a limited number of functionalities are time-shared on the same piece of FPGA region.

The problem of mapping a hardware specification to FPGA resources is NP-complete [21], and a specialization process could generate sub-optimal solutions. There is a trade-off between the resources used for the specialization process and the quality of the resulted specialized FPGA configuration; the more resources spent on generating the specialized functionality, the fewer resources needed to implement the specialized functionality. The extent to which an optimal implementation can be achieved depends o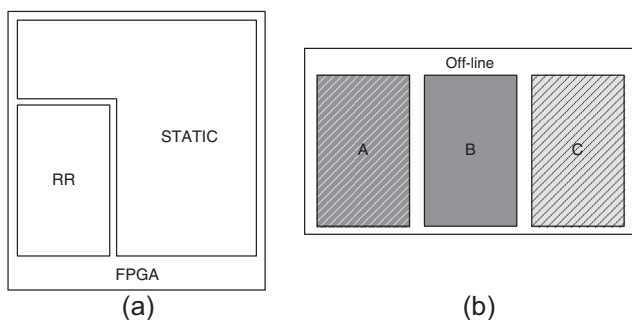n the design specification. RCS could be realized using the vendor region-based tool-chain, but this is adequate only if the number of reconfigured circuits is limited. In RCS the number of parameter values grows exponentially with the number of bits needed to represent the parameter data; generating all configurations off-line and storing them in a repository becomes infeasible for real-life applications. Instead, in FASTER project we use specialization at run-time, i.e. on-line generation of partial configurations. We do this using the method of parameterized configurations described in [22] that relies on a simplified low-overhead run-time tool-chain. Using parameterized configurations, RCS implementations having similar properties to handcrafted applications are built automatically. We build on the observation that specialization process actually implements a multivalued Boolean function, which is called Parameterized Configuration (PC). In fact, both the input, i.e. a parameter value, and the output, i.e. a specialized configuration, of the specialization process are bit vectors.

We use the staged compilation illustrated in Fig. 5. First, a parameterized configuration is constructed and represented in a closed form starting from a parameterized HDL description, shown in Fig. 5(a). This is executed at compile time, when the parameter values are unknown. Then, a specialized configuration is produced by evaluating the parameterized configuration given a parameter value, which is shown in Fig. 5(b). This is executed at run-time after the parameter values have become available. The specialized configuration is then used to reconfigure the FPGA.

Fig. 6 represents the results of each stage of micro-reconfiguration. The parameterized configuration generated off-line, is first loaded to the RR shown in Fig. 4(a). It includes a number of static bits, i.e. the rectangle with the missing circles, and a number of parameter dependent bits, i.e. the circles; this is illustrated on the left side of Fig. 6. The PC evaluation step of the on-line stage determines the values of the parameter dependent bits. Then, the RR is reconfigured at run-time with these values, by overwriting the parameter dependent bits without disrupting the static bits of RR. Fig. 6 shows that different parameter dependent bits are generated, as a result of evaluating on-line the parameter values.
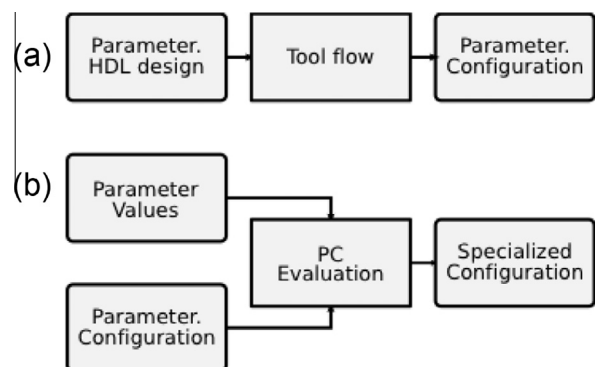
Fig. 5. Staged compilation in RCS techniques using parameterized configurations: (a) off-line stage of the tool-chain and (b) on-line stage.
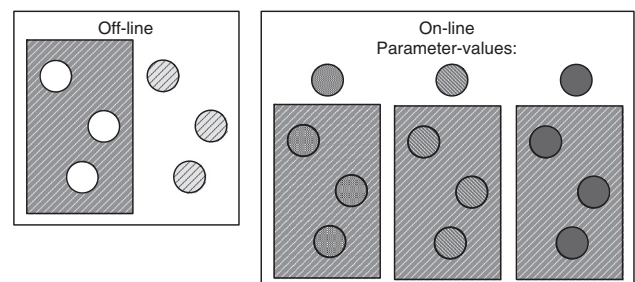
Fig. 6. Off-line and on-line results of micro-reconfiguration.

Fig. 4. (a) FPGA fabric with a pre-defined Reconfigurable Region (RR) and (b) A, B, C functions are generated off-line and each one can be loaded during run-time into the RR.

It is important to notice that the parameterized HDL description of off-line stage is an HDL description that distinguishes regular input ports from parameter input ports. The parameter inputs are not inputs of the final specialized configurations. Instead, they will be bound to a constant value during the specialization stage.

The early version of micro-reconfiguration flow in [22] was able to support reconfiguration of LUTs only. Within the context of FASTER project we extended it so as to support reconfiguration of the routing architecture as well. In particular, in [23] we extended a technology mapping algorithm so as to use the run-time reconfigurability of the routing infrastructure, which led to larger area gains as compared to the initial version [22]. Then in [24], we proposed an efficient router for handling connections that are reconfigured at run-time.

Another limitation of the initial micro-reconfiguration flow was the difficulty in determining at an early design stage whether an application will benefit from a micro-reconfigurable implementation. To this direction we developed an RTL profiler that analyzes the RTL description of the application and uses a metric called functional density in order to compare different micro-reconfigurable implementations. Functional density was introduced in [20] for measuring the amount of computation that can be placed within a certain logic area. Our RTL profiler aims at finding the most interesting parameters and estimate the functional density of the corresponding implementations. The designer then evaluates the most interesting parameter choice, and the extent to which the design will benefit from micro-reconfiguration. We also studied the feasibility of a high-level profiler able to explore the potential gains from micro-reconfiguration earlier in the design cycle. This profiler focuses on the data path of the high-level descriptions. Our case study comes from the domain of multi-mode applications, and in a recent work we discuss the best parameter candidates for micro-reconfiguration in this specific domain [25]. In our tool-chain, after the high-level profiling takes place, the part of the design that get gains from micro-reconfiguration is annotated in the XML file.

5. Verification of changing systems

Verification ensures that an optimized, reconfiguring design preserves the original behavior. In the FASTER workflow, there are two complementary aspects to validate and verify reconfiguring designs: first, given a set of configurations, ensuring the correct one is loaded, verifying the correctness of reconfiguration at run-time; second, verifying the correctness of a reconfigurable design compared to a traditional design. The novelty of our approach lies in (i) verifying streaming designs including metaprogramming; (ii) verifying designs using run-time reconfiguration; and (iii) verifying co-design of systems containing hardware and software.

Section 5.1 outlines our approach to the first challenge using traditional approaches such as checksums; the rest of the Section deals with the remaining challenges by combining symbolic simulation and equivalence checking.

5.1. Micro-architectural support for run-time signature validation

At run-time, FASTER-based systems may change due to region-based reconfiguration, micro-reconfiguration, or Custom Computing Unit (CCU) relocation. To this end, the FASTER system matches each CCU with its corresponding *signature* to check integrity and validity. The signature type is chosen to suit available resources and validation requirements: for simple signatures (checksums or cryptographic hashes), validation checks that the signature matches the CCU. For complex signatures (proof traces or complete symbolic proofs), signature validation also verifies functional correctness.

On loading a new partial reconfiguration, the system first validates its bitstream using the signature. For complex signatures, the system also verifies functional correctness of the CCU using the signature. The bitstream will be loaded into the target device only if this process succeeds.

The FASTER tool-chain provides hardware support to the run-time system for signature validation and verification. Basic support includes, but is not limited to:

- dedicated storage space for previous verification points;
- a signature checker to verify that CCUs and signatures match;
- counters to track the number of verifications and verification results statistics.

5.2. Equivalence checking of reconfigurable streaming designs

Our work concerns the correctness of reconfigurable designs rather than the correctness of the reconfiguration process. Traditional approaches to design validation simulate reference and optimized designs with test inputs, comparing the outputs. Such approaches, e.g. Universal Verification Methodology [26], use verification goals and automation to improve coverage; however, there is always a danger that the test inputs do not cover all cases, or that outputs are only coincidentally correct.

Instead of numerical or logical simulation, our approach combines *symbolic simulation* with *equivalence checking*. Symbolic simulation uses symbolic rather than numeric or logical inputs; the outputs are functions of these symbolic inputs. For example, symbolically simulating an adder with inputs a and b could result in $a + b$. For larger designs, it is hard to distinguish different but equivalent outputs ($b + a$ instead of $a + b$) from incorrect ones. The equivalence checker tests if the outputs of transformed designs are equivalent to those of the reference design.

Previous work: Industrial tools for formal verification include Formality [27], working with existing hardware flows to ensure the equivalence of RTL designs with optimized and synthesized netlists. An academic approach published in [28], verifies equivalence of FPGA cores using a model checker, and proposes run-time verification by model checking at run-time, which is necessarily restricted to small designs such as adders. Another approach in [29] verifies run-time reconfigurable optimizations using a theorem prover. Other researchers have considered verification of properties of discrete event systems (such as deadlock freedom) by model checking [30], verifying programs running on FPGA-based soft processors [31], verifying declarative parameterized hardware designs with placement information using higher-order logic [32], and verifying that hardware requested at run time implements a particular function using proof-carrying code [33,34].

Our approach relates to work on design validation of imaging operations using symbolic simulation and equivalence checking [35]. This work embeds a subset of a C-like language for FPGA design
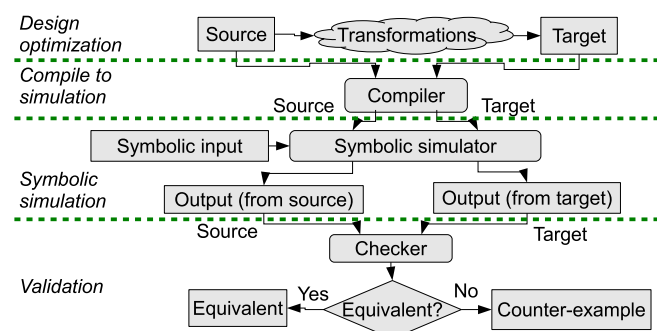

Fig. 7. Verification design flow.

into a theorem prover, using symbolic simulation and an equivalence checker to verify the correctness of transformed designs. Unlike that work, we verify optimizations of streaming designs, with our implementation using Maxeler MaxCompiler. This means that we must (i) preserve the order of inputs to and outputs from the design and (ii) allow for user metaprogramming, since Maxeler designs are Java programs. Furthermore, we extend our approach to hardware-software co-design and run-time reconfiguration.

Fig. 7 shows our approach, comparing a reference design (the *source*) with a transformed design (the *target*). For the FASTER project, we compare designs implemented as Maxeler MaxJ kernels; our approach could apply to other hardware design descriptions, or to software. The verification happens in four phases:

- *Design optimization*: the rest of the FASTER design flow transforms a source design to a target;
- *Compilation for simulation*: compile the MaxJ kernel for the symbolic simulator in two steps: (i) interpret the program to unroll any compile-time loops in the MaxJ design, and (ii) compile the design to a symbolic simulation input using a syntax-directed compile scheme;
- *Symbolic simulation*: a symbolic simulator applies symbolic inputs to source and target designs;
- *Validation*: the Yices equivalence checker [36] compares the outputs of source and target, resulting in either success (source and target designs match), or failure, with a counter example showing why the designs are not equivalent.

5.3. Verifying dynamic aspects of the design

The FASTER tool-chain generates run-time reconfigurable designs that are not supported by symbolic simulators or equivalence checkers. Rather than modifying these tools, we adapt an approach modeling run-time reconfiguration using *virtual multiplexers* [37], enclosing mutually-exclusive configurations within virtual multiplexer-demultiplexer pairs. We compile the run-time reconfigurable parts of designs to be enclosed by such pairs. We modify the configuration controller to generate the control inputs to the multiplexers to choose the appropriate configuration. Our approach applies equally to static, region-based reconfiguration, or micro-reconfiguration.

5.4. Hardware-software co-design

Hardware designs are rarely developed in isolation; often, software is a part of an overall design. Furthermore, designers often start with a software reference design, (e.g. a textbook algorithm implementation), which they accelerate with reconfigurable hardware. Hence, we extend our approach to verify hardware-software co-designs.

We model hardware-software codesign by compiling from software to the symbolic simulator. We adapt a syntax-directed hardware compilation scheme, which has the advantage that the number of simulation cycles is statically determinate, making it easier to compare software and hardware simulation outputs. To interface hardware and software, we use a synchronous API (application programming interface); this limits parallelism but simplifies software design. The API contains three calls:

- *load*: loads a streaming hardware design compiled with our hardware compiler,
- *run*: runs a previously-loaded hardware design for a given cycle count, with one or more input or output arrays, which must match stream inputs and outputs on the hardware design,
- *set_scalar*: sets a scalar hardware input value, which will apply to the hardware design on the next call to *run*.

To model runtime reconfiguration, we add an API call to load multiple streaming hardware designs and switch between them by writing to a particular scalar input, whose value controls the virtual multiplexers selecting which design is configured into the reconfigurable region.

6. Run-time system support

The Run-Time System Manager (RTSM) is a software component controlling the execution of application workloads. It undertakes low-level operations so as to offload the programmer from manually handling fine grain operations such as scheduling, resource management, memory savings and power consumption. In a partially reconfigurable FPGA-based system, in order to manage dynamically the HW tasks, the RTSM needs to support specific operations [38]. Fig. 8 illustrates our target system model along with the components participating in run-time system operation [39]. The FPGA is managed as a 2D area with regard to the HW task placement (a HW task corresponds to a HW module). Loading of tasks is controlled by a General Purpose Processor (GPP), while programming of FPGA configuration memory is done through the ICAP configuration port. All tasks can have both SW and HW versions available. HW tasks are synthesized at compile time, and stored as partial bitstreams in a repository (omitted from Fig. 8 for clarity), which accords with the restrictions of Xilinx FPGA technology. Each task is characterized by three parameters: task area (width and height), reconfiguration time, and execution time. In Fig. 8, four distinct components implemented outside the reconfigurable area participate in the control of tasks:

- Placer (P): responsible for finding the best location for the task in the FPGA.
- Scheduler (S): finds the time slot in which a task is loaded/starts execution.
- Translator (T): resolves the task coordinates by transforming a technology independent representation of the available area into the low-level commands for the specific FPGA.
- Loader (L): communicates directly with the configuration port for FPGA programming.

The system of Fig. 8 is general enough to describe similar systems. Hence, instead of ICAP, external configuration ports can be employed such as the SelectMAP or JTAG. The GPP can be a powerful host processor (implementing Placer, Scheduler and Translator) communicating with the FPGA device through a PCI bus (e.g. desktop computing with OS), or, it can be an embedded processor (with/
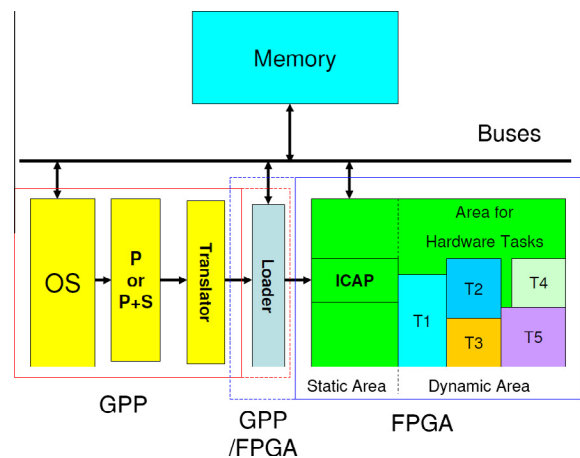

Fig. 8. System model showing the components of the run-time system.

without OS) connected to the FPGA through a dedicated point-to-point link. In any case, communication latency and bandwidth should be evaluated and used effectively for optimal results.

Our RTSM supports the following operations. *Scheduling* handles tasks by placing them in proper time slots; tasks can be immediately served, or can be reserved for later reconfiguration or/and execution. *Placement* should be efficiently supported so as to find the most suitable reconfigurable region. *Configuration caching*, which concerns placing the configuration data that will be required in the future close to the configuration memory. Different configuration times have been observed depending on the type of memory used for caching and the configuration controller [40]. The RTSM also supports *configuration prefetching*, which alleviates the system from the reconfiguration overhead by configuring a task ahead of time.

6.1. The FASTER architectural interface

The architectural interface consists of the configuration agnostic ISA extensions and the technology independent bitstream format.

6.1.1. Configuration content agnostic ISA interface

Our ISA interface resembles the Molen programming paradigm introduced in [41]. It represents a sequential consistency paradigm for programming Custom Computing Machines (CCUs), consisting of a General Purpose Processor (GPP) and a number of Reconfigurable Units (RUs), implemented using FPGAs. The FPGA is viewed as a co-processor that extends the GPP architecture. An arbiter module is introduced between the main memory and the GPP that partially decodes the instructions and forwards them either to the GPP or to the reconfigurable units. The software instructions are executed by the GPP, and the hardware operations by the Reconfigurable Units. In order to pass parameters between the GPP and the RUs, an additional Register File is used the XREGs. A multiplexor facilitates the sharing of the main memory between the GPP and the Reconfigurable Units. There are dedicated instructions for passing values between the GPP and the XREGs. In the case of micro-reconfiguration, configuration memory is expressed as a function of a set of parameters. This function takes the parameter values as input and outputs an FPGA configuration that is specialized for these parameter values; the function is called a parameterized configuration. The corresponding parameterized configuration is evaluated after the bitstream is loaded from the memory, and a specialized component generates the final reconfiguration data before sending them to the configuration port.

6.1.2. FASTER technology independent bitstream format

Fig. 9 illustrates the operation of task creation at design time. The configuration data and task specific information are merged together in a so-called Task Configuration Microcode (TCM) block introduced in [39]. TCM is pre-stored in the memory at the Bitstream (BS) Address. It is assumed that each task requires reconfigurable area with rectangular shape. The configuration data is obtained from the vendor specific synthesis tools. After this, we can create the technology independent bitstream format, shown in Fig. 9. The task specific information includes the length of the configuration data, the Task Parameter Address (TPA), the size of the task, the Execution Time per Unit of Data (ETPUD) and a flag that specifies the reconfiguration type (RT) region-based or micro-reconfiguration.

The length of the configuration data, the throughput of the reconfiguration port and the type of reconfiguration controller are used to estimate the reconfiguration time, in terms of clock cycles. The size of the task is expressed by the width and height of the task, expressed in terms of atomic reconfigurable units, e.g. CLBs. The TPA contains pointers to the locations of the input and output parameters of the task. Using the ETPUD in conjunction
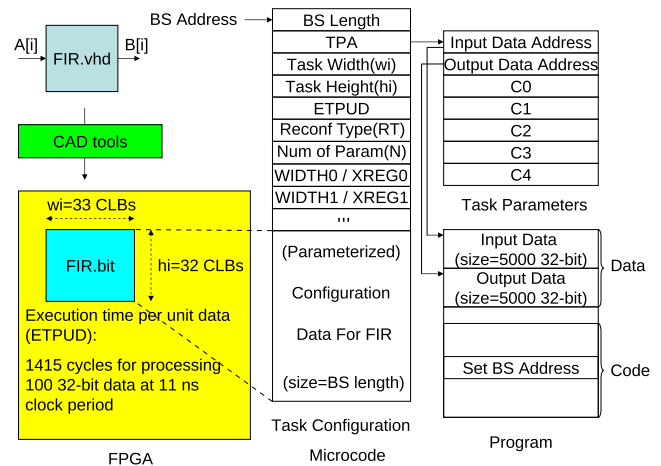

Fig. 9. Technology independent bitstream format.

with the size of the input and output parameters, the execution time (in clock cycles) can be estimated. In the case of micro-reconfiguration, additional parameters are included. The first is the number of parameters of the parameterized configuration (N), followed by N pairs of parameter width/index of the XREG containing the parameter value. Finally, a binary representation of the parameterized configuration data is included.

By utilizing a generic model for the size of the tasks, the low level details of the FPGA are abstracted, avoiding the need to expose proprietary details of the bitstreams, which may differ amongst vendors.

6.2. Input at compile-time and run-time

In the FASTER project we target platforms combining software and reconfigurable elements. The RTSM determines which task of an application is going to run on which processing element (PE) of the platform; a PE can be a discrete component such as a CPU, a static hardware part, or, a partially reconfigurable part of an FPGA. The RTSM is fed with information by the baseline scheduler created at compile time. Such information is the task graph for representing the task dependencies, and the initial mapping of the tasks in the PEs. During execution, the RTSM finds the time slot in which a task should be loaded (for a partially reconfigurable region this means that a reconfiguration should occur) and the task execution starting time. Also, in case a request for loading a HW task to the reconfigurable logic is made but there is no space available, the RTSM either activates a SW version of the task (if it is available), or reserves the task for future execution based on predicted free space and specific starting times.

Below we identify the parameters determined at compile time that do not change, as well as the ones that do change during runtime. The former ones called static parameters should be saved in storage means that will be accessed during execution only for reading, e.g. initially stored in a file system and then loaded to the data segment of the CPU memory, while the latter ones called dynamic parameters are saved in dynamic allocated storage updated at runtime. Below we list the set of these parameters mainly focused to the information needed to manage the reconfigurable elements of the platform.

Static parameters

- Reconfigurable regions: designated at compile time using the region-based method.
- Reconfiguration time: this attribute is related directly with the size of reconfigurable region. Other contributing factors are the

throughput of reconfiguration port, the memory storing the bit-stream, and the reconfiguration controller.

- ETPUD (Execution Time Per Unit of Data): it is fixed, as it concerns the time elapsed to process a specific amount of data, and affects the task overall execution time. However, the task overall execution time can be also influenced by the overall size of the data to be processed, which might not be fixed, and by the nature of the data to be processed.
- Tasks that at compile time are assigned to be executed in fixed PEs, i.e. CPU or static HW taks and tasks assigned to certain reconfigurable areas.

Dynamic parameters

- The current status of each reconfigurable region. Possible conditions are: empty, (re)configuring, busy executing, not empty but idle, the ID of the task placed on the reconfigurable region.
- The current status of each task. Possible conditions are: already placed in a reconfigurable region and ready for execution, (re)configuring, running, to be executed in a partially reconfigurable region but not yet configured, the reconfigurable region in which the task is loaded.
- Task execution time. It can depend on: (i) the amount of data to be processed, such as real-time data entering the system through a network link, or, (ii) the nature of data to be processed, or, (iii) the amount of times a loop iterates before completing data processing.

7. Evaluating FASTER tools on industrial applications

To asses FASTER applicability we used our tools and explored reconfiguration capabilities on three industrial applications. Beyond describing the applications, the contributions of present Section are:

- identification of the application functions to be accelerated in hardware;
- the way each application benefits from reconfiguration and identification of application parts that worth to be reconfigured;
- potential parallelism of applications and profiling;
- use of FASTER tools for analyzing and implementing the applications;
- qualitative and quantitative analysis of performance and area consumption results and their trade-offs;
- evaluation of the high-level analysis tool of Section 3; and
- overhead of the run-time system of Section 6.

The FASTER project aims at serving different application domains, i.e. high-performance computing, desktop and low-cost embedded, thus our tools target different platforms.

7.1. Reverse time migration

We employed FASTER tools to implement Reverse Time Migration (RTM), a seismic imaging technique used in oil and gas industry to detect terrain images of geological structures based on Earth's response to injected acoustic waves [42].

7.1.1. Application description

The objective of RTM is to create images of the subsurface of Earth from acoustic measurements performed at the surface. This is done by activating a low frequency acoustic source on the surface, and recording the reflected sound waves with tens of thousands of receivers for several seconds (typically 8–16 s). This

process is called a "shot" and is repeated many thousands of times while the source and/or receivers are moved to illuminate different areas of the subsurface. The resulting dataset is dozens or hundreds of terabytes in size, and the problem of transforming it into an image is computationally intensive.

The concept behind RTM operation is simple. It starts with a known "earth model", which is the best known approximation to the subsurface geology, indicatively represented with acoustic velocity data. Scientists conduct simultaneously two computational modeling experiments through the earth model, both attempting to simulate the seismic experiment conducted in the field – one from sources perspective and one from receivers perspective. The source experiment involves injecting our estimated source wavelet into the earth and propagating it from $t0$ to our maximum recording time t_{max}, creating a 4D source field $s(x, y, z, t)$; typical values for x, y, z, t are 1000–10000. At the same time, we conduct the receiver experiment; we inject and propagate the recorded data starting from t_{max} to $t0$, creating a similar 4D volume $r(x, y, z, t)$. We have a reflection where the energy propagated from the source and receiver is located at the same position at the same time, thus an image can be obtained by summing the correlation of the source and receiver wavefield at every time point and every "shot".

7.1.2. Application analysis and parallelism

A 3D subsurface image is generated by simultaneously propagating two waves through a model of the earth, and correlating the results of the simulations. These operations are carried out by the propagate and image kernel respectively. The propagate kernel computes the wavefield state at the next timestep based on the current and previous timesteps. The image kernel performs the cross-correlation of the source and receiver wavefields. These form the main computational kernels of RTM.

Propagating source and receiver in opposite directions in time leads to high memory requirements as two state fields at different points in time must be maintained. In our implementation, to avoid storing full 4D data volumes (can be many terabytes in size), we compute the source wavefield fully forward in time and then back in time in parallel to the receiver field. This approach propagates the source twice, and thus requires 50% more computation than the naive approach, but avoids the data management problem and reduces the memory footprint. Algorithm 1 shows the pseudo-code for running RTM algorithm on a single "shot".

Algorithm 1. RTM pseudo-code for a single "shot".

```
migrate_shot(shot_id) {
    src_curr = zeros(nx,ny,nz);    src_prev = zeros(nx,ny,nz);
    rcv_curr = zeros(nx,ny,nz);    rcv_prev = zeros(nx,ny,nz);
    image = zeros(nx,ny,nz,nh);
    model = load_earthmodel(shot_id);
    for t = 0 ... tmax {
            add_stimulus(shot_id, t, src_curr);
            propagate(src_curr, src_prev, model);
    }
    swap(curr_src, prev_src);       // reverse time direction
    for t = tmax ... 0 {
            propagate(src_curr, src_prev, model);
            add_receiver_data(shot_id, t, rcv_prev)
            propagate(rcv_curr, rcv_prev, model);
            if (i % image_step == 0); // typically every 5–10
    steps
            image (src_curr, rcv_curr, image);
    }
}
```

A typical RTM has tens of thousands of "shots", with each "shot" taking minutes, hours or days to compute, and this axis is almost embarrassingly parallel. Each "shot" can be computed independently from any other "shot", and eventually all "shots" are combined with a simple addition to form the final image result. Going into further details of the application is out of the scope of present work; more information is available in [43]. Ideally we opt to parallelize "shots" over multiple nodes in a cluster. For performance purposes (and when considering the potential impact of reconfiguration) it makes sense to consider only the "shot" computation since this dominates the runtime, and examine a single "shot" as a test case. Thus, in the context of the FASTER project we restrict the RTM test case to the single shot/single node case.

7.1.3. Reconfiguration opportunities

We focused on implementing the propagate and image kernels as distinct partially reconfigurable modules sharing the same reconfigurable region within an FPGA. There is no feedback of values within a single timestep, thus we implemented both kernels as a streaming datapath with a feed-forward pipeline. The two kernels are amenable to time-multiplexing using partial reconfiguration because they run sequentially and perform fundamentally different computations; stating otherwise the two kernels are mutually exclusive, i.e. when propagate kernel executes, the imaging kernel is idle and vice versa. Imaging typically runs less frequently than propagation. This is observed in the above pseudo-code, which shows that the propagate calculation runs for all timesteps and the imaging runs only every N timesteps. Time-multiplexing allows for saving FPGA resources, which can instead be used to increase the parallelism of the individual kernels and potentially improve the runtime performance.

7.1.4. Implementation

We implemented the RTM on a platform from Maxeler Technologies targeting HPC applications. It provides fully integrated computer systems containing FPGA-based dataflow engines (DFEs), conventional CPUs and large storage means. We target one of the standard compute nodes within such platforms, the MPC-C series MaxNode containing 12 Intel Xeon CPUs, 144 GB of main memory, and 4 DFEs. Each DFE utilizes a large Xilinx Virtex-6 FPGA attached to 48 GB of DDR3 DRAM. DFEs are connected to the CPU via PCI Express, and in addition have a high-bandwidth low-latency direct interconnect called MaxRing for communicating between neighboring engines. The system architecture is shown in Fig. 10. Maxeler nodes run standard Red Hat/CentOS operating systems and provide management tools for debugging and event logging of both the CPU and DFE subsystems. The MaxelerOS runtime software includes a kernel device driver, status monitoring daemon and runtime libraries for use by individual applications.
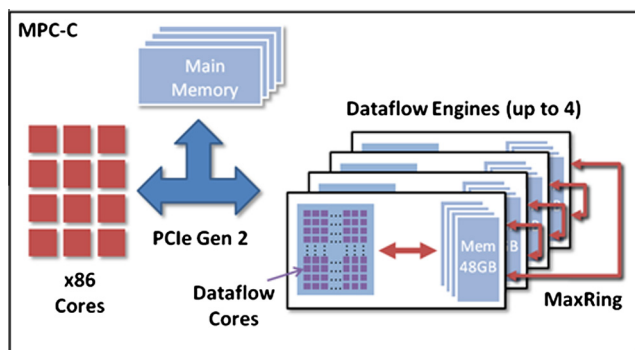
The MaxelerOS management daemon coordinates resource use, scheduling and data movement including automatic configuration of the FPGAs and allocation to different CPU processes.

We broke the RTM application code down into a DFE and a CPU part. The DFE part executes the RTM kernels. The CPU part is responsible to load the configuration onto a DFE, transfer data to and from the DFE, and run the kernels. The CPU program runs on the main processor calling simple functions to execute operations on the DFE. Initially, we implemented a static design as baseline for comparing it with the partially reconfigurable design; in this design the propagate and image kernels co-exist and the proper kernel is enabled as needed. On the other hand, in the reconfigurable version the CPU controls swapping in and out the two kernels into the same region according to the phase of execution. Reconfiguration occurs relatively rarely and it is triggered from the CPU, which streams the corresponding partial bitstream from CPU main memory to the DFE as needed.

Below we discuss the important findings from implementing and executing the partial reconfiguration version of RTM:

- Due to the nature of application, during reconfiguration the rest of the FPGA does not function. The host is also idle, waiting for imaging data, making it impossible to hide the reconfiguration time under any useful operations. Our findings show that for small problems this has a significant impact on performance, while for larger problem sizes it becomes negligible. Indicatively, for a small problem in which $100 \times 100 \times 100$ adjacent spatial points are computed, 71% of the total end-to-end runtime was spent in reconfiguration. This was drastically reduced down to 0.75% for computing a problem size of $800 \times 800 \times 800$ adjacent spatial points. Commercially interesting problem sizes scale up from the tested sizes up to several thousand cubed. Larger problem sizes increase the compute time, rendering the reconfiguration time a less significant portion of the overall wall clock time.

- Compared to the baseline non-reconfigurable implementation, performance in the partially reconfigurable design was reduced due to that designing with PR affected the clock. In addition, although in PR design the two kernels share the same region, while in the non-reconfigurable version they co-exist in the chip, we found that the overhead from the resources added from the PR flow is considerable, especially for relatively small problem sizes. Thus, we obtained a modest reduction of the resources required for the propagate and image reconfigurations. This is mainly due to the increased static resource usage from the extra logic being introduced to perform partial reconfiguration, and to the additional dummy inputs and outputs needed for both kernels to maintain a stable IO interface.

- Huge amount of data must be preserved during reconfiguration. In the first implementation we used the FPGA's attached DRAM memories to store the seismic wavefields and earth model volumes during the computation of a "shot". If the FPGA is partially reconfigured during the execution of a "shot" computation, it is important to preserve the DRAM contents in order to enable the computation to proceed with the correct data.

- A major trade-off between performance and area comes from the instantiation of a DRAM controller in the same FPGA that hosts the RTM kernels. In specific, in the first version we implemented a DRAM memory controller using the FPGA resources. Holding data, i.e. earth model, current states of the wavefield propagations and accumulated image, in the on-card memory connected directly to the FPGA on the DFE card is more efficient since the DFE memory provides much greater bandwidth than the PCI Express link. However, we discovered that the DRAM memory controller consumes a large amount of chip area, which restricts the area that can be used for implementing

Fig. 10. Maxeler system architecture.

the application kernels and reduces the possible gains from utilizing reconfiguration. In the second implementation we moved the storage from the on-card memory to the CPU, and consequently omitted the memory controller. Now, we store the data in the host CPU memory and thus all transactions between CPU and FPGA are performed through the PCIe. This reduces the achievable throughput, but it also relieves the FPGA resources, thus allowing for more space in designing either more modules, or more parallelism within the module, or more pipelining or more parallelized modules running.

7.1.5. Using FASTER high-level analysis tool on RTM

RTSM is based on Earth's response to injected acoustic waves. The wave propagation within the tested media is simulated forward, and calculated backward, forming a closed loop to correct the velocity model, i.e. the terrain image. The propagation of injected waves is modeled with the isotropic acoustic wave equation:

$$\frac{d^2p(r,t)}{dt^2} + dvv(r)^2 \bigtriangledown^2 p(r,t) = f(r,t) \tag{1}$$

The propagation involves stencil computation, as the partial differential equation is approximated with the Taylor expansion. In our implementation, the propagation is approximated with a fifth-order Taylor expansion in space, and first-order Taylor expansion in time. The constant coefficients are calculated using finite difference methods. In this part, we focus on the forward propagation function of an RTM application.

The high-level analysis results and the measured results for RTM are compared in Table 1. For the target RTM application, design properties are estimated with high-level analyses, and customized designs are developed with MaxCompiler version 2012.1 to a Xilinx Virtex-6 SX475T FPGA hosted by a MAX3424A card from Maxeler Technologies. As shown in Table 1, the design parallelism (i.e. the number of duplicated data-paths) is limited by LUT and BRAM resource usage. In current experiments, we set the available resources to be 90% of the available FPGA resources, to reduce the design routing complexity. In terms of estimation accuracy, the resource usage and execution time are more than 90% accurate. This indicates that the high-level analysis captures the design properties without going through the time-consuming synthesis tool chain. Moreover, this implies the efficiency of the RTM design. The listed execution time consists of the execution and reconfiguration time of RTM design. The high-level analysis estimates the design execution time based on the theoretical peak performance: all implemented data-paths are assumed to be running full speed in parallel. In other words, with the analytical model taking care of application and circuit details, the applications are running with almost the theoretical performance.

7.2. Ray tracing

In modern graphic applications it is important to achieve photo-realistic rendering in a coherent manner in order to improve picture quality with increased scene complexity, and visualize accurately characteristics such as real reflection, soft shadows, area light source, and indirect illumination. Rendering in 3D graphic design is the process of generating an image from a model (or models in what collectively can be called a scene file), by means of computer programs; this adds shading, color and lamination to a 2D or 3D wireframe to create life-like images of a screen. A scene file contains objects in a strictly defined language or data structure; it can include geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The problem of performing 3D rendering effectively is important due to the continuous strive for realistic images in several domains, as for instance in movies and video games. In this context we study the ray tracing scheme, which belongs to the global illumination group of algorithms [44] that aim to add more realistic lighting in 3D scenes.

7.2.1. Application description

Ray tracing simulates the physics of a light ray to produce realistic results. Its classical implementation consists in defining a rendering point in a 3D scene and shooting light rays from that point, simulating their reflections and refractions when these rays intersect objects in the scene. The objects are described as a composition of geometric primitives (2D or 3D) such as triangles, polygons, spheres, cones and other shapes. The computational complexity of a rendering scene is proportional to the number and the nature of these primitives, along with their positions in the scene itself.

The ray tracing algorithm we use as benchmark in FASTER starts from a description of the scene as a composition of certain geometric 2D/3D primitives: triangles, spheres, cylinders, cones, toruses, and polygons. Each primitive is described by a set of geometric properties such as position in the scene, orientation, scale, height of the primitive or rays of the circles composing the primitive. The algorithm then performs the following steps:

1. the scene is divided in blocks, called voxels, and the number of these voxels is one of the contributors to determine the complexity of the algorithm; the more the voxels are, the more intersections between rays and primitives have to be computed;
2. the algorithm generates a certain amount of rays from the current rendering point of the image, and it computes the set of voxels traversed for each of these rays;
3. it then iterates all over these voxels and computes the intersection between the primitives in the voxel and the current ray;
4. the nearest intersection, if any, is considered and the algorithm computes the reflection and refraction of the light ray on the surface of the object;
5. the rays generated by this physic simulation continue to be propagated into the image until a maximum number of intersection (an input parameter of the application) is reached or no intersection is found at all.

Ray tracing operation can be thought of as by having a virtual camera placed into the scene, an image is rendered by casting rays simulating the reverse path of ray of lights, from the origin of the camera through every pixel of its virtual focal plane. The color of a pixel is determined by the potential intersections of the primary ray cast through it, with the 3D scene. Photorealistic results can be achieved when sufficient rays are cast, simulating with fidelity the behavior of light. Realistic description of the objects of the scene and proper simulation of the materials' behavior is important for correct results.

During rendering, the camera creates a single primary ray originating at the camera location, and pointing in the direction of a single sample inside a given pixel of a virtual focal plane. The 3D scene is interrogated for potential intersection with this primary ray, using a structure that contains the geometric primitives. The closest intersection to the camera between the ray and the scene

Table 1
Estimated and analyzed design properties for RTM. The usage of a resource type is divided by the amount of available FPGA resources of the specific type.

RTM	Frequency (GHz)	LUT	FF	DSP	BRAM	Time (s)
Estimated	0.1	0.86	0.53	0.4	0.89	33.02
Measured	0.1	0.91	0.54	0.41	0.887	36.05

is determined by computing the intersection with all the primitives of the scene. If an intersection with a primitive is found, its shading is computed using the material of the corresponding primitive. Shading computation may generate additional rays, cast in turn into the scene following a similar process. The result of shading stage is a color associated with the intersection point. For a primary ray, this color is used to compute the color of the original pixel by the camera. For a secondary ray (or shadow ray), this color is used to compute the color of the surface hit by the primary ray. More specifically, several rays (shadow rays) are shot from the intersection point in the direction of the light sources. For area light sources, a random sample is computed over the surface and chosen as direction of the corresponding shadow ray. If a shadow ray reaches its target without occlusion, then the considered point is receiving direct light from that source; otherwise, the point is in shadow. The accumulation of contributions of shadow rays permits the rendering of soft shadows and penumbras. Reflections are taken into account by casting a new ray (secondary ray or reflected ray) in the reflection direction, and the process starts again for that ray. Its contribution is accumulated to its corresponding primary ray for the final result.

7.2.2. Parallelism and profiling

To study the potential for parallelism we optimized the software code by exploiting multithreaded rendering on a symmetric multiprocessor architecture. Ray tracing is embarrassingly parallel, but the level of parallelism that can be exploited depends on platform capabilities. On a multi-processor/multi-threaded platform the image can be decomposed in independent sub-areas, and each sub-area can be rendered independently by a single-processor/thread. Indicatively, we divided the screen in tiles, e.g. 8×8, and used one thread per tile for accessing shared scene data. Table 2 has the execution time for processing a 800×800 image size in a 4-core Intel CPU using different number of threads, and Fig. 11 shows the achieved speedup in each case. The results indicate that ray tracing application benefits considerably from deployment onto parallel processing elements.

Due to the complexity of application, before proceeding with the hardware implementation we profiled it to identify the most-time consuming functions that worth to be accelerated. We analyzed it using a profile data visualization tool [45]. Tests were carried out on a fixed scene by varying the number of lights (one or three) and shadow sampling and reflection depth values (from two to ten), obtaining 26 profiles. We then compared the 26 profiles to identify the 10 most time-consuming functions and the fraction of execution time spent in each of them. Fig. 12 shows the profiling results.

We examine the self cost of each function without taking into account their callers. Results can be better explained referring to the mean value and to the variance of the fraction of execution time spent by each function. The chart in Fig. 12 depicts the fraction of time per function over the total execution time. We obtained that the most demanding function is *Intersect* with an average value of $21,49\%$. Variance has a negligible value for all selected functions except for the three (3) most expensive ones. This behavior is due to the algorithm operation; *Intersect*,

Table 2
Performance of ray tracing executed in different number of threads.

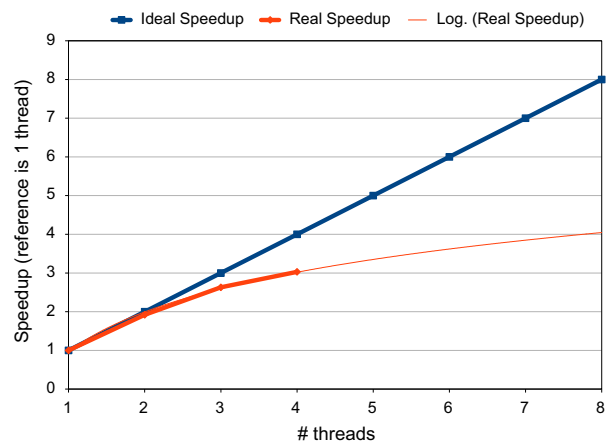| Thread count	Time (ms)
1	9674.01
2	5092.78
3	3709.03
4	3195.60

Fig. 11. Ideal and measured speedup from the execution of ray tracing in a multi-threaded environment in a 4-core Intel CPU. Graph includes a rough estimation of the expected (or logical) speedup for up to 8 threads.

Ispointinsidepolygon and *Intersectcone* show a dependence on the total amount of generated rays, while the remaining seven (7) functions keep closer to the mean value. In general, variance becomes negligible as the number of rays increases; this keeps constant for different scenes and different settings, and no big variations in the values can be seen.

A basic ray tracing application must compute the intersections between a ray and the complete synthetic scene in order to evaluate visibility. It should be noted here that for primary and secondary rays, only the intersection closest to the source of the ray must be found, while for shadow rays, it is sufficient to determine if there is any intersection between the source and the destination of the ray. However, in order to find the proper intersection a considerable amount of such computations might be performed. In fact, an important finding of the profiling was that the most computational intensive part is related to the computation of ray intersections within the voxels, i.e. the 3D partition grid we adopted to reduce the computational cost. The intersection functions are called potentially hundreds of time per ray, for hundreds of thousands or millions of rays. Ray tracing spends a lot of time in computing intersections between a ray and a geometric primitive of the scene, as the rendering of an image requires casting rays through each pixel. Hence, we decided to accelerate the intersection functions by implementing them in hardware. On the other hand the complexity of intersection computation depends itself on the nature of the primitive, e.g. intersection ray-sphere is cheaper to compute than ray-polygon.

7.2.3. Reconfiguration opportunities

The ten functions in Fig. 12 are the best candidates to be implemented as hardware IPs. Opportunities for reconfiguration arise if we consider that most of them cannot run in parallel with others, i.e. they are mutually exclusive. For this reason, it is possible to assign all hardware implementations of these functions to a single reconfigurable region (or a few of them) and reconfigure the functions so as to keep the execution time in line with that of a design in which all functions co-exist implemented as static cores, but with far fewer resources. This is beneficial in two scenarios:

- The first one is when the target device does not fit the whole architecture. This is useful when ray tracing is ported to small embedded devices for wearable solutions running for example augmented reality tasks. The device is reconfigured at runtime on the basis of type of intersections needed at a certain point in the computation of final image. This solution has been evaluated with our current implementation.
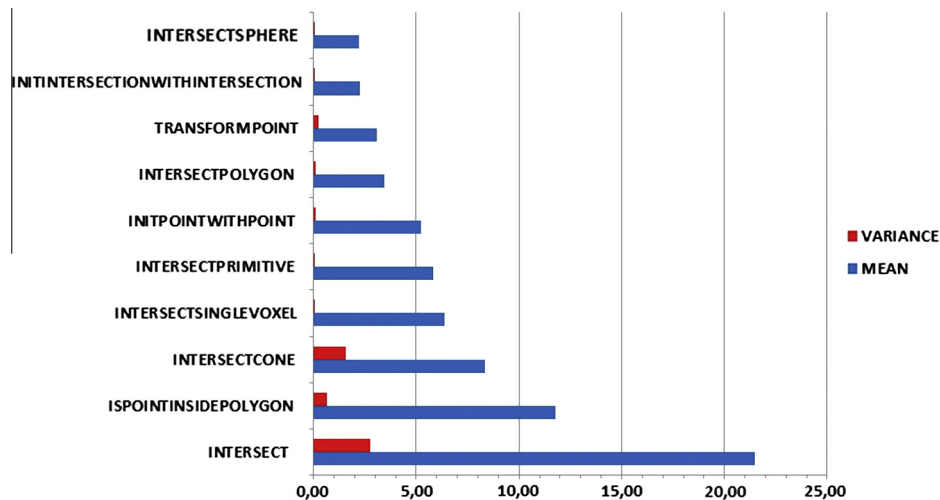
Fig. 12. Profiling of the 10 most CPU-time intensive functions of ray tracing. They consume most of the total execution time.

- The second scenario comes from altering the mix of HW cores used to speed up the computation at runtime, in order to adapt their type and number to the current processing needs. In this case, with a change to the SW of our current implementation, it is possible to detect at runtime if it is beneficial to parallelize the computation of a certain number of intersections on multiple HW cores.

The benefit from this type of reconfiguration is similar to that in Reverse Time Migration application described above, i.e. time-multiplexing kernels for saving resources. Different levels of reconfiguration can be considered to address adaptable aspects of ray tracing in terms of performance or flexibility, such as reconfiguring the type of supported geometric primitives, or, the complexity of primitives for trading-off intersection computation-cost, or, the shading complexity with the computation of color at intersection point for trading-off the level of realism with computation-cost.

7.2.4. Implementation

We created a reconfigurable design based on the first scenario described previously targeting the low-cost Zedboard, which is based on the Xilinx Zynq platform. Zynq is an ARM-based SoC with limited reconfigurable logic. We used our tools for analyzing and implementing the application in HW, in particular the mapper, scheduler and floorplanning. Initially, we restructured the original code so as to become more parallelizable; this way multiple intersections can be computed at the same time. We then encoded the task graph along with the related profiling information in the FASTER XML format. The mapper and scheduler tool determined that the application can benefit in terms of performance by implementing the intersections in HW instead of SW. Then we used the floorplanning tool to build the partially reconfigurable system. We should note here that we put considerable effort to better understand how to efficiently manipulate the ray tracing data structures for hardware implementation.

The partially reconfigurable system alters at run-time the mix of cores available to compute intersections. At design time we explored the solution space by varying the number of reconfigurable regions. From a preliminary study this seems to vary from 1 up to 4, but these boundaries depend also on the underlying physical architecture, mainly on the amount of reconfigurable resources. The interface to the reconfigurable regions is fixed and their resource occupation must be enough so as to serve every core the designer plans to load into the reconfigurable region. In our case, for sake of flexibility we mapped all intersection cores in all

reconfigurable regions, i.e. for every intersection we created one bitstream for every reconfigurable region.

7.3. Network intrusion detection

Network Intrusion Detection Systems (NIDS) are widely adopted, as high-speed and always-on network access demand more sophisticated packet processing and increased network security [46].

7.3.1. System description

Instead of checking only the header of incoming packets (as for example firewalls typically do), NIDS also scan the payload to detect suspicious contents. The latter are described as signatures or patterns and intrusion signature databases are made available that include known attacks. The databases are regularly updated and an NIDS should be able to provide a certain degree of flexibility to incorporate the updated security information. In the past, NIDS used mostly static patterns to scan packet payload. Recently, regular expressions have also been adopted as a more efficient way to describe hazardous contents. As such, modern rulesets consist of both static patterns and regular expressions.

The general requirements of NIDS involve high processing throughput, flexibility in modifying and updating the content descriptions and scalability as the number of the content descriptions increases. The performance requirements are fundamental to the correct functioning of an NIDS system, as if it cannot meet them, the system itself is susceptible to specific types of attacks, i.e. overload and algorithmic attacks. However, equally crucial is the ability of the NIDS to adapt to updated rules and content descriptions. This is why software NIDS systems have been widely used, however they require substantial hardware resources (in terms of general-purpose CPUs) to achieve link-speed performance.

NIDS implemented in reconfigurable hardware have the potential to combine the high performance of hardware-based systems with the flexibility of software solutions. Specific rules can be mapped to custom logic for maximum performance and rule changes/updates can be reconfigured into the device. Typically, a NIDS has to deal with: (i) small incremental updates may be required to add, change or expand certain IP addresses or address ranges that appear in detection rules, (ii) new static pattern rules may have to be added to the static rule set or changes to the current patterns included in the rule set may have to be applied, (iii) updates in the regular expressions to cover more cases or correct

mishandling of certain detection rules, (iv) new regular expressions may have to be added, and (v) overall updates to the system might needed in case of new policies or large scale update to the operation of the NIDS system.

7.3.2. Reconfiguration opportunities

Depending on the nature of the update (incremental versus more extensive ones), different reconfiguration approaches can be used. If a new rule is added in the system (the usual case), micro-reconfiguration can dedicate free resources to this new rule. Furthermore, micro-reconfiguration could be used in the cases where small changes to certain patterns or rules are required, e.g. changes in an IP address or address range or maybe updates to certain rules defined by a specific regular expression. Since these changes/updates are expected to be frequent, micro-reconfiguration is a very promising approach as it can be applied quickly and minimize system down-time. On the other hand, for major restructuring of the rules used in the NIDS (either as initial setup, a major upgrade, or to respond to new requirements of the organization), partial or even full reconfiguration is needed. The ability to support large-scale changes to the operation of the NIDS provides high value to the system, prolongs system life and protects the investment of the customer.

7.3.3. Implementation

The NIDS system has been tuned to support micro-reconfiguration for updating static rules related to IP checking. The original NIDS was a static hardwired implementation of a subset of the Snort rules, without any software programmable units. Fig. 13 overviews the main modules of the system. The use of micro-reconfiguration mandates the use of a processor for certain tasks related to reconfiguration and calculation of parameters, therefore a number of changes were made to the system. Fig. 14 has the micro-reconfigurable version of NIDS. The main NIDS System unit is practically the original implementation of the hardwired NIDS system; however it comes with several changes that include the interface to the PLB bus, and the use of micro-reconfigurable parameters for the IP checking rules.

The performance of the original non-reconfigurable system and the micro-reconfigurable system is identical and set at wire-speed, thus both systems produce results at the same rate as they receive packets from the gigabit ethernet interface. Also, the micro-reconfigurable system operates under the same timing constraints as the original. It should be noted that the main NIDS of Fig. 14 operates on a separate clock domain than the rest of the system (PLB buses, MicroBlaze processor and hardware ICAP).
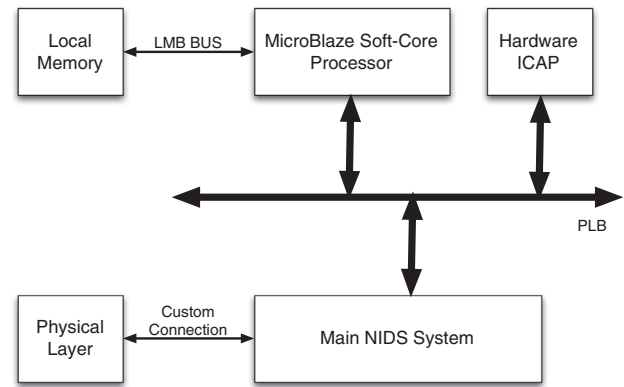

Fig. 14. Micro-reconfigurable NIDS system.

As expected, there is an impact on the physical resources that the micro-reconfigurable system needs compared to the original implementation. For a Xilinx Virtex-5 XC5VLX110T device, the overhead is almost 6% for Slice LUTs and 11% for BlockRAMs. Although these are not negligible values, they can be considered small.

7.4. Overhead of the run-time system manager

We used the RTSM described in Section 6 to control a partially reconfigurable design. In specific, we used it to execute an edge detection application implemented on an FPGA platform with Xilinx Virtex-5. Initially, we evaluated it on a x86 desktop system connected with a serial interface to the FPGA platform. The desktop system was a linux PC with an i3 CPU@3.1 GHz, 3 MB cache, and 2 GB DDR3. The time we measured per call of RTSMs main core was 1.3us on average, while the best and worst times were 0.9us and 2us respectively. These values correspond to the overhead of the RTSM itself, i.e. the time it takes to perform one scheduling decision and update the internal structures representing the status of the tasks and the FPGA. This measurement excludes the times for task reconfiguration and execution, and communication between the CPU and the FPGA. We then transited to a completely embedded version implemented on the same FPGA platform by altering the RTSM code so as to port it on a Microblaze, running at 100 MHz, 32 KB cache. In that case the RTSM overhead is 520us, thus 2 orders of magnitude larger as compared to the x86 implementation. In both cases the RTSM itself is very low overhead; it is the reconfiguration and execution time of the tasks that dominate the total time for carrying out the application execution.
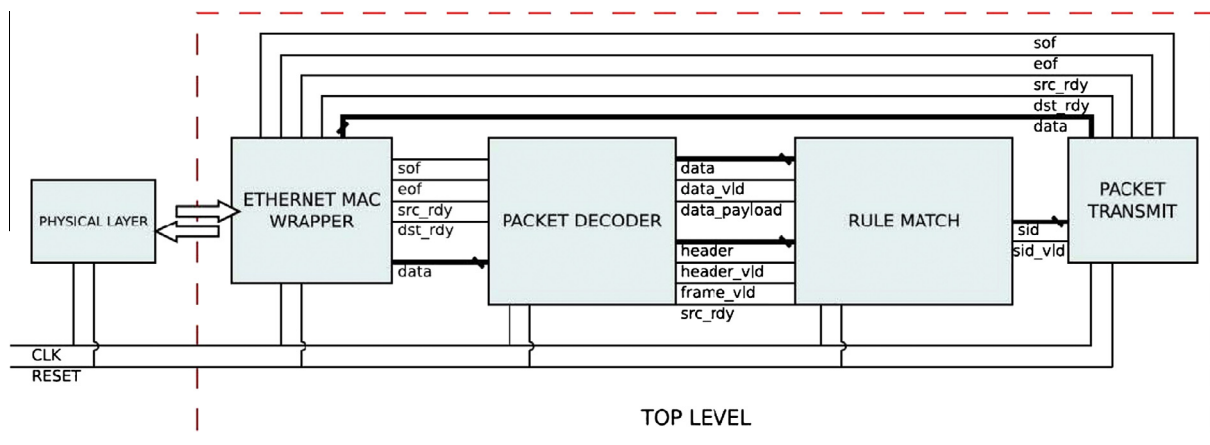

Fig. 13. Overview of the hardware NIDS system.

8. Summary of contributions

This section consolidates the project novelties and exposes its differences over previous related efforts discussed in Section 2. Our contributions are:

- The interaction amongst the front-end stages of the tool-chain, which is performed through the XML file. During this operation different segments of the XML structure are automatically analyzed and updated. Several solutions are explored requiring limited-to-none intervention from the user, while details and implications related with dynamic reconfiguration remain hidden. XML format offers a convenient representation and has been used by similar projects in the past [10]. However, the type of information contained into FASTER XML that is then extracted for feeding the different stages of the tool-chain is new, e.g. baseline scheduler provided in XML format that is then used as input to the run-time system manager. A new feature of the proposed XML is that it contains attributes characterizing the HW and SW tasks, which are used as input to the other FASTER tools for making decisions. Such attributes are execution time, reconfiguration time, and power consumption per task; the way these are balanced at compile time affects the system implemented at the first place, but also affects the run-time decisions. This feature is unique as compared to previous research projects, not only due to the information contained in XML, but also due to that it affects both compile- and run-time decisions.
- Micro-reconfiguration is a technique released some time ago, but we study for the first time its integration into a tool-chain. We are also extending it so as to support routing resources; previous version of micro-reconfiguration was targeting Look-Up-Tables only. Moreover, we developed a profiler to assist the user in determining whether an application will benefit from a micro-reconfigurable implementation. Finally, we evaluated micro-reconfiguration by employing it for the first time in one of the target applications, i.e. Network Intrusion Detection.
- The proposed tool-chain supports different implementation options, static or reconfigurable; for the latter case two options are available, either region-based reconfiguration or micro-reconfiguration. Putting these sub-flows into work under a unified environment targeting heterogeneous reconfigurable platforms constitutes itself a challenging problem in terms of complexity. To the best of our knowledge existing integrated solutions do not support such feature, i.e. analyzing and identifying within the tool-chain which option serves better the given application/platform, and enabling accordingly the proper sub-flow.
- A verification approach that applies equally to static, region-based, or micro-reconfiguration without modification. The novelty of this approach lies into three aspects: (i) verifying streaming designs using meta-programming; (ii) verifying run-time reconfigurable designs; and (iii) verifying co-design of HW/SW systems. Sections 5 discusses these aspects in detail, including an extensive comparison over the dominant solutions.
- We implemented a Run-Time System Manager (RTSM) and evaluated it on an embedded and a desktop platform. We studied the extent to which it is feasible to develop a generic library that can support different platforms. For the transition from the desktop implementation to the embedded one only a few changes were needed in the RTSM code. Our code has been structured in a way that can be extended to support different scheduling policies. The RTSM basic operation is driven by the baseline scheduler represented in XML.
- We explored the reconfiguration aspects of three applications, which are of great interest to the industrial partners. We gathered significant insights by progressively re-designing and implementing the partially reconfigurable versions on certain platforms, and assessed two of them in terms of performance and resource savings. Our experience showed that this constitutes a time-consuming task; important role played the platforms that our tool-chain targets, and the initial available implementations of the target applications (static HW or SW only). In addition, we employed our tools for analyzing and implementing the applications. In particular, we used the High-level analysis tool for analyzing the Reverse Time Migration and we found that the user can rely on its output in order to estimate the expected performance and resource consumption, without going through the time-consuming synthesis tools. We used our mapper, scheduler and floorplanning tools to analyze and implement the ray tracing application. Prior to this we profiled the application, and encoded its task graph along with the related profiling information in the FASTER XML format. Finally, we designed and evaluated the micro-reconfigurable version of the Network Intrusion Detection System.

Recapitulating, the FASTER project is novel both from research and practice points of view. The basic functionality for some of the described tools was provided by the partners already, but we are continuously modifying them so as to integrate all under a unified tool-chain. Furthermore, FASTER tools support the analysis and design of applications from different domains.

9. Conclusion

Creating a changing hardware system is a challenging process requiring considerable effort in specification, design, implementation, verification, as well as support from a run-time system. We attempt to alleviate this effort and streamline the design and implementation process by providing a new design environment friendly to reconfiguration. Our contributions span the analysis phase and the reconfigurable system definition, the support for multi-grain reconfiguration, the verification for the changing system, and the run-time system to handle the reconfiguration requirements.

Acknowledgement

References

[1] FASTER, <http://www.fp7-faster.eu/> (accessed 2014).
[2] D. Pnevmatikatos, T. Becker, A. Brokalakis, K. Bruneel, G. Gaydadjiev, W. Luk, K. Papadimitriou, I. Papaefstathiou, O. Pell, C. Pilato, M. Robart, M.D. Santambrogio, D. Sciuto, D. Stroobandt, T. Todman, FASTER: Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration, in: Euromicro Conference on Digital System Design (DSD), 2012.
[3] T. Todman, G. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, P.Y.K. Cheung, Reconfigurable computing: architectures and design methods, Comput. Digital Tech., IEE Proc. 152 (2) (2005) 193–207.
[4] J. a. M.P. Cardoso, P.C. Diniz, M. Weinhardt, Compiling for reconfigurable computing: a survey, ACM Comput. Surv. 42 (4) (2010) 13:1–13:65.
[5] L. Jówiak, N. Nedjah, M. Figueroa, Modern development methods and tools for embedded reconfigurable systems: a survey, Integr. VLSI J. 43 (1) (2010) 1–33.
[6] S. Hauck, A. DeHon, Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
[7] M. Platzner, J. Teich, N. Wehn (Eds.), Dynamically Reconfigurable Systems – Architectures, Design Methods and Applications, Springer, 2010.
[8] PTOLEMY, <http://ptolemy.eecs.berkeley.edu/> (accessed 2012).
[9] DAEDALUS, <http://daedalus.liacs.nl/> (accessed 2012).
[10] hArtes, <http://hartes.org/hArtes/> (accessed 2012).
[11] M. Sato, OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors, in: Proceedings of the 15th International Symposium on System Synthesis, 2002, pp. 109–111.

[12] OpenCL, <http://www.khronos.org/opencl/> (accessed 2012).

[13] REFLECT, <http://www.reflect-project.eu/> (accessed 2012).

[14] ACOTES, <http://www.hitech-projects.com/euprojects/ACOTES/> (accessed 2012).

[15] ANDRES, <http://andres.offis.de/> (accessed 2012).

[16] A. Montone, M.D. Santambrogio, F. Redaelli, D. Sciuto, Floorplacement for partial reconfigurable FPGA-based systems, Int. J. Reconf. Comput. 2011 (2) (2011) 12.

[17] C. Bolchini, A. Miele, C. Sandionigi, Automated resource-aware floorplanning of reconfigurable areas in partially-reconfigurable FPGA systems, in: Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL), 2011, pp. 532–538.

[18] M. Rabozzi, J. Lillis, M.D. Santambrogio, Floorplanning for partially-reconfigurable FPGA systems via mixed-integer linear programming, in: Proc. of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2014.

[19] P.W. Foulk, Data-folding in SRAM configurable FPGAs, in: Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), 1993, pp. 163–171.

[20] M.J. Wirthlin, B.L. Hutchings, Improving functional density through run-time constant propagation, in: Proc. of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), 1997, pp. 86–92.

[21] K. Shahookar, P. Mazumder, VLSI cell placement techniques, Comput. Surv. (1991) 143–220.

[22] K. Bruneel, Efficient Circuit Specialization for Dynamic Reconfiguration of FPGAs, PhD thesis, Ghent University, 2011.

[23] K. Heyse, K. Bruneel, D. Stroobandt, Mapping logic to reconfigurable FPGA routing, in: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 315–321.

[24] E. Vansteenkiste, K. Bruneel, D. Stroobandt, Maximizing the reuse of routing resources in a reconfiguration-aware connection router, in: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 322–329.

[25] B.A. Farisi, K. Bruneel, J.M.P. Cardoso, D. Stroobandt, An automatic tool flow for the combined implementation of multi-mode circuits, in: Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition, 2013, pp. 821–826.

[26] Accellera, Universal Verification Methodology (UVM) 1.1 user's guide. <http://www.accellera.org/downloads/standards/uvm>.

[27] Synopsis, Formality: equivalence checking for DC Ultra, Tech. rep., 2012.

[28] S. Singh, C.J. Lillieroth, Formal verification of reconfigurable cores, in: Proceedings of FCCM, 1999, pp. 25–32.

[29] K.W. Susanto, T.F. Melham, Formally analyzed dynamic synthesis of hardware, J. Supercomput. 19 (1) (2001) 7–22.

[30] F. Madlener, J. Weingart, S.A. Huss, Verification of dynamically reconfigurable embedded systems by model transformation rules (2010) 33–40.

[31] K.W. Susanto, W. Luk, Automating formal verification of customized soft-processors, in: FPT, 2011, pp. 1–8.

[32] O. Pell, Verification of FPGA layout generators in higher-order logic, J. Autom. Reason. 37 (1–2) (2006) 117–152.

[33] S. Drzevitzky, U. Kastens, M. Platzner, Proof-carrying hardware: towards runtime verification of reconfigurable modules, in: ReConFig. 2009, pp. 189–194.

[34] S. Drzevitzky, Proof-carrying hardware: runtime formal verification for secure dynamic reconfiguration, in: FPL, 2010, pp. 255–258.

[35] K.W. Susanto, T. Todman, J.G.F. Coutinho, W. Luk, Design validation by symbolic simulation and equivalence checking: a case study in memory optimization for image manipulation, in: SOFSEM, 2009, pp. 509–520.

[36] B. Dutertre, L. de Moura, The YICES SMT Solver, Tech. rep., Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, 2006.

[37] W. Luk, N. Shirazi, P.Y.K. Cheung, Modelling and optimising run-time reconfigurable systems, in: Proceedings IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 1996, pp. 167–176.

[38] C. Steiger, H. Walder, M. Platzner, Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks, IEEE Trans. Comput. 53 (11) (2004) 1393–1407.

[39] T. Marconi, Efficient Runtime Management of Reconfigurable Hardware Resources, PhD thesis, TU Delft, 2011.

[40] K. Papadimitriou, A. Dollas, S. Hauck, Performance of partial reconfiguration in FPGA systems: a survey and a cost model, ACM Trans. Reconf. Technol. Syst.(TRETS) 4 (4) (2011) 36:1–36:24.

[41] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, E. Panainte, The MOLEN polymorphic processor, IEEE Trans. Comput. 53 (11) (2004) 1363–1375.

[42] E. Baysal, D.D. Kosloff, J.W.C. Sherwood, Reverse time migration, SEG-Geophysics 48 (11) (1983) 1514–1524.

[43] X. Niu, Q. Jin, W. Luk, Q. Liu, O. Pell, Exploiting run-time reconfiguration in stencil computation, in: Proc. of the 22nd IEEE International Conference on Field Programmable Logic and Applications (FPL), 2012.

[44] K. Myszkowski, T. Tawara, H. Akamine, H.-P. Seidel, Perception-guided global illumination solution for animation rendering, in: Proceedings of the ACM Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), 2001, pp. 221–330.

[45] KCachegrind, <http://kcachegrind.sourceforge.net/html/Home.html> (accessed 2014).

[46] I. Sourdis, D. Pnevmatikatos, S. Vassiliadis, Scalable multi-gigabit pattern matching for packet inspection, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 16 (2) (2008) 156–166.

Dionisios Pnevmatikatos is a Professor and former Chair of the Electronic and Computing Engineering Department, Technical University of Crete and a Researcher at the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science, FORTH in Greece. He received his B.Sc. degree in Computer Science from the Department of Computer Science, University of Crete in 1989 and M.Sc. and Ph.D. degrees in Computer Science from the Department of Computer Science, University of Wisconsin-Madison in 1991 and 1995 respectively. His research interests are in the broader area of Computer Architecture, where he investigates the Design and Implementation of High-Performance and Cost-Effective Systems, Reliable System Design, and Reconfigurable Computing. He is currently the Coordinator of the FASTER project (ICT, STREP) and has participated in numerous national and European research projects.

Kyprianos Papadimitriou is a Research Associate at the Computer Architecture and VLSI Systems (CARV) Laboratory, FORTH-ICS, and Scientific Staff at the School of Electronic and Computer Enginneering, Technical University of Crete, in Greece. He received his Diploma and M.Sc. in Electronic and Computer Engineering from Technical University of Crete, Greece, in 1998 and 2003 respectively. During 1998–1999 he was with the R&D department of ATMEL working on hardware implementation of wireless protocols. In 2003 he co-initiated an effort to establish a spin-off company involved with motion recognition technology. In 2012 he was granted with a Ph.D from the Electronic and Computer Engineering Department, Technical University of Crete, with a special focus on reconfigurable computing. He has participated in several European and national reserarch projects.

Tobias Becker received a Dipl. Ing. degree in Electrical Engineering from the University of Karlsruhe, Germany in 2005 and a PhD in computer science from Imperial College London, UK in 2011. He is currently a research associate in the Department of Computing at Imperial College London. His research includes work on reconfigurable computing and high-performance computing. He has extensive experience on FPGA design and he has worked in collaboration with several industrial partners including Xilinx and Nokia.

Peter Böhm received his Ph.D. from the Department of Computer Science at Oxford University, UK, and his M.Sc. (Honour's) degree from the Department of Computer Science at Saarland University, Germany. Currently, he is a research associate at Imperial College London. His academic background is in formal verification and specification of hardware and communication architectures. At Imperial College, his research focuses on the formal verification of re-configurable hardware designs.

Andreas Brokalakis is a Senior Computer Engineer at Synelixis Solutions Ltd. He graduated from the Computer Engineering and Informatics Department, University of Patras (Greece) in 2004 and received his master's degree from the same department in 2007. Since 2008 he has been working on several European Research projects through his affiliation with the Tele-communication Systems Institute (Technical University of Crete) and Synelixis Solutions. His focus is on hardware design for reconfigurable systems or ASICs as well as computer architecture.

Karel Bruneel (Ph.D. Ghent University, 2011) is a postdoctoral researcher at Ghent University, affiliated with the Department of Electronics and Information Systems (ELIS), Computer Systems Lab (CSL), Hardware and Embedded Systems (HES) group. His research is situated in the domain of EDA tools for FPGAs with a strong focus on dynamic reconfiguration. He is interested in a large number of topic including synthesis, technology mapping, placement and routing.

Catalin Ciobanu is a researcher in the Computer Science and Engineering department at Chalmers University of Technology, Sweden. His research interests include computer architecture, reconfigurable computing and parallel processing. He has an M.Sc. in Computer Engineering from Delft University of Technology, and a Ph.D. on customizable vector register files.

Tom Davidson (Master of Applied Science – Electronics, Ghent University, 2009) is a Ph.D. researcher at Ghent University, affiliated with the Department of Electronics and Information Systems (ELIS), Computer Systems Lab (CSL), Hardware and Embedded Systems (HES) group. He works on run-time reconfiguration of FPGAs, more specifically Dynamic Circuit Specialization. He has presented and published several papers on this subject. His current research focus is on identifying opportunities for DCS from higher abstraction levels, such as System-C or C.

Georgi Gaydadjiev is a professor at the Department of Computer Science and Engineering at Chalmers University of Technology. His research interests include computer systems design, advanced computer architecture and micro architecture, reconfigurable computing, hardware/software co-design and computer systems testing. He is a Senior IEEE and ACM member.

Karel Heyse (Master of Applied Science – Computer Science, Ghent University, 2011) is a Ph.D. researcher at Ghent University, affiliated with the Department of Electronics and Information Systems (ELIS), Computer Systems Lab (CSL), Hardware and Embedded Systems (HES) group. His research is about run-time reconfiguration of FPGAs and in particular the synthesis tools that make it possible to use run-time reconfiguration to increase performance and functional density of digital circuits. His other research interests are in coarse grain architectures and regular expression matching on FPGAs.

Wayne Luk is Professor of Computer Engineering at Imperial College London. He was a visiting professor with Stanford University, California, and with Queen's University Belfast, United Kingdom. His research includes theory and practice of customizing hardware and software for specific application domains, such as multimedia, financial modeling, and medical computing. His current work involves high-level compilation techniques and tools for high-performance computers and embedded systems, particularly those containing accelerators such as FPGAs and GPUs. He received a Research Excellence Award from Imperial College, and 12 awards for his publications from various international conferences. He is a fellow of the IEEE, the BCS, and the Royal Academy of Engineering.

Xinyu Niu received his B.Sc. degree in School of Information Science and Technology from Fudan University, and obtained his M.Sc. degree from Department of Electrical and Electronic Engineering at Imperial College London. He is currently completing his Ph.D. studies in the Department of Computing, Imperial College London. His research interests include run-time reconfiguration, heterogeneous computing, high performance computing and computer-aided design (CAD) tools for hardware design optimization.
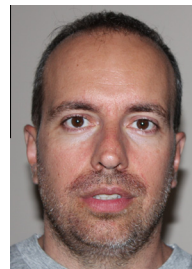
Ioannis Papaefstathiou is the Manager of the Systems Research Groups of Synelixis Solutions Ltd and an Associate Professor at the ECE Department at the Technical University of Crete. He was granted a Ph.D. degree in computer science at the University of Cambridge UK, in 2001, an M.Sc. (Ranked 1st) from Harvard University, USA, in 1996 and a B.Sc. (Ranked 2nd) from the University of Crete, Greece in 1996. He has published more than 70 papers in IEEE-sponsored journals and conferences. He has been the prime Guest Editor for an issue of IEEE Micro Magazine and for one in IEEE Design & Test Magazine.

Danilo Pietro Pau received the Electronic Engineering degree from Politecnico of Milano in 1992. He has, since 1991, joined STMicroelectronics Advanced System Technology working on digital video processing algorithms and architectures for Set Top Box and Mobile application processors. He moved to STMicroelectronics Ltd, Bristol (UK) for a period of 1 year, in 2004, where he worked mainly on algorithms and architectural studies in the field of 3D graphics for mobile processors compliant with the worldwide standard OpenGL-ES. He then moved back to Agrate, where he continued to manage 3D graphics developments and started 2D scalable graphics algorithms and architectural studies for Set Top Box and Thin Clients. He is currently managing 3 teams based in UK, Italy and China. During his career he has authored or co-authored many European, U.S. and Japan granted patents, as well as international publications in conferences or technical journals.

Oliver Pell is Vice President of Engineering at Maxeler Technologies. His experience ranges from accelerating Reverse Time Migration and Lattice-Boltzmann simulations to credit derivatives pricing. He is currently responsible for Maxeler's technology research and development as well as the technical architecture and project management of acceleration efforts for clients including Tier 1 oil companies and investment banks. He has an M.Sc. in advanced computing from Imperial College London. He's a member of IEEE and the ACM.

Christian Pilato is a research assistant at Politecnico di Milano. His research interests include high-level synthesis, evolutionary algorithms for design space exploration and multi-objective optimization, and design of heterogeneous and multiprocessor embedded systems. He has an M.Sc. in Computing Systems Engineering from Politecnico di Milano, received in 2007, and a Ph.D. in Information Technology received from the same University in 2011. He is a member of the IEEE and a member of the IEEE Computer Society.

Marco D. Santambrogio received his laurea (M.Sc. equivalent) degree in Computer Engineering from the Politecnico di Milano in 2004, his second M.Sc. degree in Computer Science from the University of Illinois at Chicago (UIC) in 2005 and his Ph.D. degree in Computer Engineering from the Politecnico di Milano in 2008. He was at the Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT as postdoc fellow and he is now assistant professor at Politecnico di Milano, research affiliate at MIT and adjunct professor at UIC. He has also held visiting positions at the EECS Department of the Northwestern University (2006 and 2007) and Heinz Nixdorf Institut (2006). He is a member of the IEEE, the IEEE Computer Society (CS) and the IEEE Circuits and Systems Society (CAS). He has been with the Micro Architectures Laboratory at the Politecnico di Milano, where he founded the Dynamic Reconfigurability in Embedded System Design (DRESD) project in 2004 and the CHANGE project (Self-Aware and Adaptive Computing Systems) in 2010. He conducts research and teaches in the areas of reconfigurable computing, computer architecture, operating system, hardware/software codesign, embedded systems, and high performance processors and systems.

Donatella Sciuto is a full professor in Computer Science and Engineering at the Politecnico di Milano, where she holds also the position of Vice Rector of the University. She is a Fellow of IEEE and currently President of the Council of EDA. Her main research interests cover the methodologies for the design of embedded systems and multicore systems, from the specification level down to the implementation of both the hardware and software components, including reconfigurable and adaptive systems. She has published more than 200 papers. She is or has been member of different program committees of ACM and IEEE EDA conferences and workshops. She has served as Associate Editor of the IEEE Transactions on Computers, and serves now as associate editor to the IEEE Embedded Systems Letters for the design methodologies topic area and as associate editor for the Journal of Design Automation of Embedded Systems, Springer. She is in the executive committee of DATE for the past ten years and she has been Technical Program Chair in 2006 and General Chair in 2008. She has been General Co-Chair for 2009 and 2010 of ESWEEK. She is Technical Program Co-Chair of DAC 2012 and 2013.
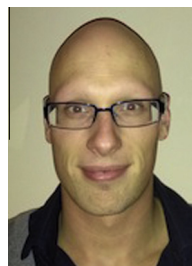
Dirk Stroobandt (Ph.D. Ghent University, 1998) is a professor at Ghent University, affiliated with the Department of Electronics and Information Systems (ELIS), Computer Systems Lab (CSL). He currently leads the research group HES (Hardware and Embedded Systems) of about 10 people with interests in semi-automatic hardware design methodologies and tools, run-time reconfiguration, and reconfigurable multiprocessor networks. He is the inaugural winner of the ACM/SIGDA Outstanding Doctoral Thesis Award in Design Automation (1999), and he initiated and co-organized the International Workshop on System-Level Interconnect Prediction (SLIP) since 1999. He is also lead editor of a special issue of the International Journal of Reconfigurable Computing and he has been associate editor of ACM's TODAES for three years.

Tim Todman received the B.Sc. degree from the University of North London, and the M.Sc. and Ph.D. degrees from Imperial College London, in 1997, 1998 and 2004, respectively. He is currently a research associate in the Department of Computing, Imperial College London, London, UK. His research interests include hardware compilation, verification and implementation of graphics algorithms on reconfigurable architectures.

Elias Vansteenkiste (Master of Applied Science – Electronics, Ghent University, 2011) is a Ph.D. researcher at Ghent University, affiliated with the Department of Electronics and Information Systems (ELIS), Computer Systems Lab (CSL), Hardware and Embedded Systems (HES) group. His research is focused on a new tool flow for the dynamic reconfiguration that will allow run-time reconfiguration of the FPGA's interconnect network, in order to increase performance and functional density of digital circuits. He targets the back end of the tool flow. Placement and routing algorithms and configuration bitstream generation are his main interests.