2012 IEEE 15th International Conference on Computational Science and Engineering

Novel Design Methods and a Tool Flow for Unleashing Dynamic Reconfiguration

K. Papadimitriou[†], C. Pilato[*], D. Pnevmatikatos [†], M. D. Santambrogio [*], C. Ciobanu[§], T. Todman[¶], T. Becker[¶], T. Davidson[‡], X. Niu[¶] G. Gaydadjiev[§], W. Luk[¶], D. Stroobandt[‡],

[†] Foundation for Research and Technology - Hellas

[*] Dipartimento di Elettronica e Informazione, Politecnico di Milano

[§] Chalmers University of Technology

[¶] Imperial College London

[‡] Ghent University

Abstract— **During the last few years, there is an increasing interest in mixing software and hardware to serve efficiently different applications. This is due to the heterogeneity characterizing the tasks of an application which require the presence of resources from both worlds, software and hardware. Controlling effectively these resources through an integrated tool flow is a challenging problem and towards this direction only a few efforts exist. In fact, a framework that seamlessly exploits both resources of a platform for executing efficiently an application has not yet come into existence. Moreover, reconfigurable computing often incorporated in such platforms due to its high flexibility and customization, has not yet taken off due to the lack of exploiting its full capabilities. Thus, the capability of reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) to be dynamically reconfigured, i.e. reprogramming part of the chip while other parts of the same chip remain functional, has not yet taken off even in small-scale basis. The inherent difficulty in using the tools to control this technology has kept it back from being adopted by academia and industry alike.**

The FASTER (Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration) project aims at introducing a design methodology and a tool flow that will enable designers to implement effectively and easily a system specification on a platform combining software and reconfigurable resources. The FASTER framework accepts as input a high-level description of the application and the architectural details of the target platform, and through certain steps it can enable the full use of the capabilities of the platform, while at the same time it should be flexible enough so as to balance efficiently performance, power and area. One of the main novelties is the incorporation of partial reconfiguration as an explicit design concept at an early stage of the design flow. We target different applications from the embedded, desktop and high-performance computing domains. In all cases we will demonstrate the effectiveness of the proposed framework in exploiting the inherent parallelism of applications and enabling the runtime adaptation of the platforms to the changing needs of the applications.

I. Introduction

Combining data-flow and control-flow components, e.g. FPGAs and software microprocessors respectively, into a single platform can offer great advantages for a wide range of application domains. For this to be viable, it is important to provide a framework through which the designers will effectively access such platforms to deploy their applications. This constitutes an interesting problem, similar to the parallel programming of multicore and manycore systems, and currently the field lacks of such solutions. In particular, a tool that would manipulate rationally the resources in accommodating the application tasks, trade-off factors such as speed, power and area according to the application and user needs, and eventually distribute the computational effort to the resources fairly so as to avoid their starvation or overloading, is missing. In fact, a system should be regulated so as to sustain as much as possible high availability, meaning that the system should be able to accept - always if possible - arriving tasks of an application, update or alter existing work, and avoid entering long periods of unavailability, i.e. downtime.

Furthermore, altering the hardware functionality at runtime so as to adapt to new requirements offers a new level of flexibility which can benefit many applications. For example, this functionality can benefit Network Intrusion Detection Systems (NIDS), which aim at scanning all incoming packets for suspicious content [1], [2]. Scanning has to be carried out at line-speed so that the communications are not slowed down, while the list of threats to check for may be extended and updated on a daily basis. Fixed hardware solutions achieve high performance, and software solutions easily adapt to the new set of threats, but neither can achieve adaptivity and high performance at the same time. Reconfigurable logic allows the definition of new functions to be defined in hardware units, combining hardware speed and efficiency, with ability to adapt and cope in a cost effective way with expanding functionality, changing environmental requirements, and improvements in system features. For the NIDS the new rules can be hard-coded into the reconfigurable logic, thus retaining the high performance, while providing the necessary adaptivity and extensibility to new threats.

The above operations fall into the scope of the present project. One of our major concerns is that the FASTER framework should accept and handle properly the designer's input for segmenting effectively the application in software and hardware tasks. Emphasis is placed in identifying the

978-0-7695-4914-9/12 $26.00 © 2012 IEEE
DOI 10.1109/ICCSE.2012.61

static and dynamic parts of the hardware and harnessing partial reconfiguration abstractly, so as to hide the low-level details of the technology from the user. Towards this direction, the environment should be friendly and easily accessible. This is what we intend to do within the context of the FASTER project; elevate the abstraction of application deployment up to a level in which performance will not be sacrificed for transparency. The framework accepts as input the high-level description of the application and the platform architecture. Then, it should ensure that factors such as performance, power consumption and area will be balanced effectively and according to the user needs throughout the entire system operation, by employing a run-time system.

In order to provide the above features the framework should be characterized by a high level of flexibility. This can be realized with intervention from the designer in certain steps of the design methodology. Thus we envision a framework that will offer the option of controlling manually specific operations; although this might increase the complexity level of the design methodology at the same time it increases the flexibility. We intend to study the level up to which we will provide flexibility in application deployment, while keeping complexity at low level.

The above form the main axes along which the FASTER framework will be developed. The basic contributions of the FASTER project can be summarized in the following:

- including reconfiguration as an explicit design concept, for large changes identified at compile time and for small changes identified at runtime;
- balancing effectively execution speed, power consumption and available resources;
- integrating seamlessly software and reconfigurable components under a unified tool;
- providing flexibility to the user during application deployment while keeping complexity at low level;
- providing efficient and transparent runtime support

The paper is structured as follows: Section II presents related efforts with similar objectives and discusses the open issues that motivate our work. Section III describes the design methods we employ to achieve our goals. Section IV shows the integration of the design methods along with their connection points to form the new tool flow. In Section V we discuss the runtime system for managing the platform with a focus on dynamic reconfiguration. Section VI presents the platforms we are currently using as well as the ones we are planning to use in order to demonstrate the use of the framework in all the domains. Finally, Section VII concludes the paper and presents our next steps.

II. Motivation

In the recent years, several efforts have been funded aiming at forming a methodology to control the deployment of applications in similar platforms. Below we report some of them, and we also point out their differences with the FASTER project.

A. Related Efforts

hArtes [3] was an FP6 EU project targeting automatic parallelization and generation of heterogeneous systems. They adopted OpenMP pragmas to specify the parallelism automatically extracted from the initial sequential specification but they do not address any aspect related to reconfiguration or dynamic execution. Even verification issues were not taken into account. On the other side, we adopt the same formalism to represent the parallel application, even if the partitioning is provided by the designer since the automatic parallelization is out of the scope of this project. Similarly, the ALMA project [4] focuses on parallelization and optimization algorithms focusing on specific architecture templates provided by the partners. On the contrary, FASTER project aims at providing a unified environment where the tool flow can adapt the mapping of tasks to fully exploit the underlying architecture. Both heterogeneous embedded systems (e.g., Xilinx FPGA-based architectures) and high-performance computing (e.g., Maxeler workstations) will be targeted by the proposed framework.

The 2PARMA project [5] focuses only on the exploration of multi- and many-core architectures, without hardware accelerations. On the other hand, the REFLECT [6] project bridges the gap between multi-core processing and FPGA acceleration. In specific, it focuses more on the systematic control of all the compilation stages and the relationship with non-functional requirements, along with the generation of the hardware specifications, rather than on reconfiguration and verification aspects as in the FASTER project. Similarly, the MADNESS project [7] adopts FPGAs for the prototyping of heterogeneous architectures, by focusing on system-level design, fault tolerance and dynamic adaptivity.

Finally, the ERA project [8] adopts dynamic reconfiguration (with low-level OS support) but on a specific platform developed by the consortium and composed of a VLIW processor, a reconfigurable NoC and a memory subsystem. On the contrary, the aim of FASTER is to provide a more general approach able to take into account the specifics of the target platform. Similarly, the FlexTiles project [9] aims at developing automated methodologies for developing an energy-efficient yet programmable heterogeneous manycore platform with self-adaptive capabilities.

B. Aim of the FASTER Project

The FASTER project aims at introducing a complete methodology to allow designers to easily implement and verify a system specification on a platform that includes one or more general purpose processor(s) combined with multiple acceleration modules implemented on one or multiple reconfigurable devices. This will try to bridge the gap, but also to connect, the two worlds under within a single framework that hides the details from the user. An imporant contribution will be the micro-reconfiguration. Finally, although there are several runtime systems around targetting it, we are trying to make a flexible one able to be customized according to the application needs. Runtime will control the operation of the system by following certain directions by a baseline scheduler built at design time. Another novelty of the FASTER project is the

adoption of an XML file to exchange information between the different phases. This also allows to develop the methodologies in parallel, but also to easily integrate them into a unique tool flow, compare them in terms of the results and determine the advantages/disadvantages for each of them with respect to the application under analysis.

As a result, we expect that the envisioned tool flow will be able to reduce the design and verification time of complex reconfigurable systems by at least 20% for selected application domains, by providing additional novel verification features that are not available in existing tool flows. In terms of performance, for these application domains the tool flow could be used to achieve the same performance with up to 50% smaller cost compared to programmable SoC based approaches, or exceed the performance by up to a factor of 2 for a fixed power consumption envelope.

III. Design Methods

In order to serve the goals of the FASTER project we are employing different design methods.

A. High-Level Analysis

The FASTER tool flow begins with a high-level analysis of the design. The goal of this approach is to automatically identify and exploit run-time reconfiguration opportunities while optimising resource utilisation of the design. To avoid time-consuming iterations in the design implementation process, the optimisations are performed on a high level. The high level analysis is based on a hierarchical Data Flow Graph (DFG), additional application parameters such as input data size, and physical design constraints such as available area and memory bandwidth. The analysis produces estimates of implementation attributes such as area, computation time and reconfiguration time. The analysis also automatically explores opportunities for reconfiguration, i.e. a partitioning of the application into several reconfigurable components that increases throughput while using the same amount of area or reduces area while providing the same throughput.

The DFG represents applications with interconnected arithmetic nodes at arithmetic level, and with interconnected function nodes at function level. At arithmetic level, arithmetic operations and data access patterns within functions are identified and the resource usage for a single implementation is estimated. At function level, application functions are separated into various partitions. Each partition is a group of functions that are bundled to become a reconfigurable component. The partitions are mapped into hardware as scheduled to ensure functions are only implemented when they are active. Idle functions are eliminated and reconfiguration opportunities are identified. Data dependency between separated partitions are analysed, and memory architectures are generated based on extracted data access patterns and interactions between partitions. The partitions are then optimised as reconfigurable components. Optimisation variables include arithmetic operations presentation, computational precision and implementation concurrency. Meanwhile, implementation attributes such as area, throughput, memory usage, and memory bandwidth are estimated for optimised partitions. Given application data size, the high-level analysis can produce a fully partitioned and scheduled implementation of the application. Alternatively, the design estimates can also be provided to other parts of the FASTER tool flow to support the design analysis and reconfigurable core identification.

B. Partitioning Methodology

In this phase, we aim at providing efficient methods to partition the tasks between hardware and software, and to determine the proper level of reconfiguration for the hardware ones. In particular, after the selection, each hardware task will be tagged for:

- *no reconfiguration*: the task will be implemented as a static core;
- *region-based reconfiguration*: the task will be implemented in a reconfigurable region;
- *micro-reconfiguration*: this technique will specialize the resulting hardware implementation with respect to some slow-changing input parameters.

These selection methods incorporate static and dynamic analysis of the application and they also take into account revelant characteristics about the target architecture (e.g., communication costs). They also need information about other constraints, such as the logic area dedicated to such cores.

Task graph transformations will be also performed to improve the application after the mapping and the scheduling such as clustering consecutive tasks assigned to the same processing element to avoid unnecessary communications.

C. Region-based Reconfiguration

Region-based reconfiguration deals with the instantiation of a new function that is encapsulated in a region of the FPGA. Traditionally, configuration generation takes place at design time [10]. The designer marks a certain functionality as being reconfigurable and confines its logic to a dedicated region of the FPGA by means of floorplanning; such a region can be reconfigured while the rest of the chip is operational. Typically, a number of reconfigurable functions are allocated to the same region, resulting in partial bitstreams that can be loaded and swapped at run-time to change the desired functionality. The research challenge is the proper identification of the regions and the support of relocatable modules at run-time, which could involve additional constraints for floorplanning and placement. Bitstream relocation allows for loading a configuration bitstream into a different region than it was originally created for.

D. Micro-reconfiguration

This method is used for dynamic circuit specialization. It optimizes a circuit that is implemented on the FPGA. This original circuit is optimized for the specific values of slowly changing inputs. The new specialized circuit is smaller and faster than the original one, but is only correct for one specific value. If one of the slowly changing signals actually changes, then a new circuit is generated and the FPGA is reconfigured

with this new circuit. A method has been developed to implement this efficiently [11].

Opportunities for micro-reconfiguration can be hard to identify, thus a profiler [12] has been developed to assist the designer in finding them. The reason these opportunities can be hard to find is because they depend on the dynamic behaviour of the design. The profiler analyses the dynamic behaviour of the design and uses this information to determine whether micro-reconfiguration would improve the design. Since there are many ways to make a micro-reconfigurable implementation, it also suggests the most beneficial one. In the FASTER framework, this profiler will be used to analyse all designs that are flagged as potentiality micro-reconfigurable. This designation is added by the designer. The profiler will update this flag, based on its analysis. The designs with gains will be permanently flagged as micro-reconfigurable, the others will lose it. Alternatively, the designer can choose to bypass the profiler, and directly flag designs as micro-reconfigurable himself.

E. Baseline Scheduling and Mapping onto Reconfigurable Regions

After the actual generation of the hardware cores and their interfaces, the FASTER framework also requires to characterize such modules in terms of required resources (i.e., LUTs, BRAMs and DSP blocks). This information is necessary to evaluate the compatiblity of the implementation with a reconfiguration region candidate for the mapping. The resulting data will then be annotated into the corresponding implementations associated with each task. On this basis, the framework will determine the actual number of regions which the reconfigurable area can be partitioned in, along with their characteristics (e.g., size, position, constraints). Then, it will provide an initial assignment of the tasks tagged for region-based reconfiguration onto these regions, verifying its feasibility. If the assignment results are feasible, this phase also produces the scheduling of both these tasks and the corresponding reconfigurations, intended as a partial ordering of these activities. Such an information can be taken into account later on by the runtime scheduler to determine which task has to be executed at each time based on the ready ones and this ordering.

It is worth noting that this phase can also compute alternative mapping that can be adopted in case of external events (e.g., interrupts). In this case, the runtime manager will need to check the current status of the resources with respect to the newer configuration in order to determine which reconfigurations have to be actually performed.

F. Verification

The role of verification is to check that a simple, unoptimized design (the source) implements the same behaviour as an optimized, possibly reconfiguring design (the target). Traditionally, hardware designers have used extensive logic simulation to verify that their designs implement the desired behaviour. The downside of this approach is that the number of test inputs required to exhaustively test even a simple design can be impractically large. We thus adopt an approach combining symbolic simulation with equivalence checking. The source and target designs are first compiled to suitable input for a symbolic simulator. Symbolic simulation stimulates the design with symbolic inputs, rather than the numerical or Boolean inputs used in traditional approaches, for example simulating an adder with symbolic inputs a and b might result in a symbolic output $a + b$. Equivalence checking is used to check symbolic outputs from source and target designs that may differ but still be equivalent (for example $b + a$ instead of $a + b$). If symbolic outputs from source and target designs are equivalent for all inputs, the designs are equivalent, otherwise, the first input with different outputs can be used to debug the target design.

For reconfigurable designs, we distinguish static and dynamic aspects of their behaviour. Static aspects, which are fixed at compile-time, can be verified at compile-time. For dynamic aspects, we further distinguish (i) those that can be verified at compile-time and (ii) those that must be verified at run-time, since there is not enough information to do so at compile-time. For those that can be checked at compile-time we adopt the concept of virtual multiplexers, which models mutually exclusive reconfigurable regions of a designs as being connected by virtual multiplexer-demultiplexer pairs, sharing a control input which selects between the reconfigurable regions. This allows for verifying reconfigurable designs at compile-time using our existing approach. Work is ongoing into verifying dynamic aspects at run-time.

We aim at incorporating the above methods in the FASTER tool flow as distinct stages. Some stages will be completely hidden to the user, while other stages can be carried manually, either fully or partially, depending on the user and application needs.

IV. Tool Flow

The aforementioned design methods are used in order to shape the FASTER tool flow illustrated in Figure 1. Starting from the left side of the Figure, it begins with the description of the application (in HLL, HDL or other formats such as task graph) plus the application requirements and an abstract description of the reconfiguration capabilities of the target platform. In particular, to exchange the information among the different parts, an XML file format has been defined and it includes the following parts:

- *architecture*: the description of the target platform in terms of processing elements, along with information about the interconnections and the memories;
- *application*: the information about the initial application (e.g., source files, profiling information, call graph);
- *library*: the list of available implementations for each of the tasks which the application has been partitioned into, decorated with high-level information and estimations;
- *partitions*: the different solutions in terms of partitioning, mapping and scheduling.

When starting from HLL such as C with OpenMP annotations, analysis in the front-end part determines which portions of the application can be accelerated in hardware and which will

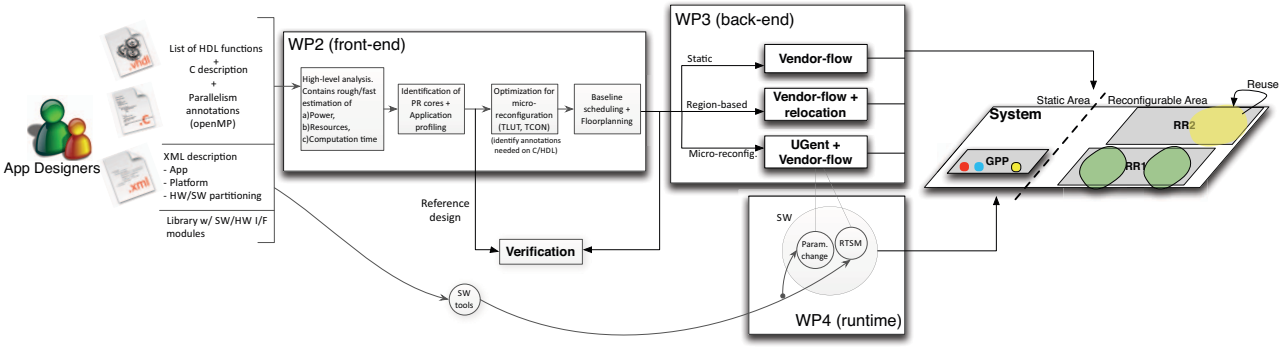

Fig. 1. FASTER design flow broken down into different parts

be executed in software. The FASTER framework focuses on the portions to be executed in hardware, and the analysis on the corresponding HDL determines which parts will be reconfigured dynamically and which parts will be static. Further analysis in the front-end will estimate the performance to offer feedback to the user, and will also determine the most proper granularity for reconfiguration. The partitioned application description is annotated with dependency information to drive the dynamic aspects of the reconfiguration. At this point the verification described in the previous Section is performed to check that the partitioned dynamically reconfigurable version of the application is equivalent to the original source. The back-end part of the tool flow performs synthesis, floorplanning, placement and routing for reconfigurable portions that have been selected for region-based reconfiguration or micro-reconfiguration. In particular, in the FASTER framework three different tools are mixed to provide all options to the designer, i.e. static, region-based and micro-reconfiguration. The final bit files will be produced by vendor-specific tools. Finally a run-time manager will be employed to control the operation of the complete reconfigurable system.

It should be noted that during the deployment of the application in the target platform, some stages of the tool flow can be carried out either automatically or manually. For instance, for micro-reconfiguration the designer can rely completely on the profiler available for suggesting the reconfiguration opportunities or by finding them manually.

One of our main concerns is the quality of the outcome of the FASTER tool flow. In order to asses it we have defined certain evaluation criteria. A main criterion is the amount of FPGA resources needed to implement a set of operations that can be reduced by utilizing partial reconfiguration, as for the static counterpart these operations should coexist in the chip. Another important criterion is the clock frequency; an increase in the frequency could be achieved due to the specialization of compute kernels, which allows to remove unnecessary logic and reduce propagation delay. Other criteria are the reconfiguration time which is added as overhead in the total execution time of the application, and whether power and energy consumption can be reduced as compared to the static design.

V. Run-Time System Manager

The Run-Time System Manager (RTSM) supports the execution of application workloads in systems usually controlled by an Operating System. It undertakes low-level operations so as to offload the programmer from manually handling delicate operations such as scheduling, resource management, memory savings, power and energy consumption. RTSM can reside always in memory and is usually implemented as a standard library. It includes subroutines that realize functions by accessing the Operating System (system calls).

In a partially reconfigurable system, in order to manage dynamically HW tasks, the RTSM needs to be extended with certain operations. The basic components of the RTSM model we adopted in the FASTER project were presented in [13]. The RTSM would incorporate operating-system style services that will allow high level operations on the hardware circuits. In its simplest form the RTSM is a software simply selecting a precompiled circuit, transmitting the corresponding configuration bitstream to the FPGA, initiating the execution and controlling the delivery of the results back to the user; this process is carried out transparently to the user. Such a system targeting the desktop domain was presented in [14]. We are interested in deploying more complex systems in different domains. Thus a sophisticated RTSM is needed able to control different operations and react accordingly even in case a certain scenario was not predicted at compile time. The bottom line is that we should ensure the high availability of target platform and minimize system downtime, e.g. cases in which a task cannot be scheduled and thus the system enters a starvation phase. In addition, we consider whether the RTSM will be generic so as to assist a wide range of systems, or, it will be generated every time a new system is implemented with the FASTER tool flow. Such an automatic run-time system is presented in [15].

The remaining section provides more details on the programming model we are planning to follow in order to create the RTSM. Also, we identify the input requirements of the RTSM determined at compile time in order to drive decisions at runtime.

A. Configuration Content Agnostic ISA Interface

The ISA (Instruction Set Architecture) interface we will use is based on the Molen programming paradigm [16]. 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), which can be implemented using FPGAs. The FPGA is viewed as a co-processor which extends the GPP architecture. An arbiter module lying between the main memory and the GPP, decodes partially the instructions and forwards them either to the GPP or to the RUs. Software instructions are executed by the GPP and hardware operations by the RUs. In order to pass parameters between the GPP and the RUs, an additional Register File is used, called XREGs. A multiplexor facilitates the sharing of the main memory between the GPP and the RUs.

In the minimal case the ISA supports the following instructions: SET, EXECUTE, MOVTX and MOVFX. A CCU description file needs to be provided which contains the input/output, SET/EXECUTE parameters and information regarding the memory regions accessed by the CCU. This configuration file, together with the configuration and execution microcode, is used by the compiler to generate the Molen binaries.

The SET instruction has a single parameter - the address at which the configuration microcode is defined. When encountering a SET instruction, the arbiter will continue loading sequential memory addresses until a terminating condition in satisfied, such as an end_op microinstruction. The SET phase prepares the CCUs for execution. Like SET, the EXECUTE instruction also has a single parameter, the address pointing to the execution microcode, which will perform the CCU operations such as hardware initialization, reading the input parameters, the computation itself and the results writeback. The end_op microinstruction marks the end of the execution microcode. For a given CCU, the SET phase should be completed before the EXECUTE stage.

In the case of micro-reconfiguration, the 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. When a SET instruction is executed, the corresponding parameterized configuration is evaluated after the bitstream is loaded from the memory. The reconfiguration controller generates the final reconfiguration data before it reaches the reconfiguration port.

The MOVE instructions are used for passing values between the GPP register file and the XREGs. In particular, MOVTX copies the content of a GPP register to an XREG, and MOVFX copies an XREG to a GPP register. If the number of XREGs is insufficient, pointers can be used to pass large data structures between the GPP and the CCUs.

Finally, the extended ISA offers instructions used to fetch the SET or EXECUTE microcode into an on-chip cache in order to minimize the reconfiguration time.

B. Actions at Design Time

The configuration data, i.e. bitstreams, is produced with the vendor specific synthesis tools at design time. Each bitstream corresponds to a HW task. Each HW task requires a reconfigurable area with rectangular shape. The FPGA is managed as 2D area in order to place the HW tasks.

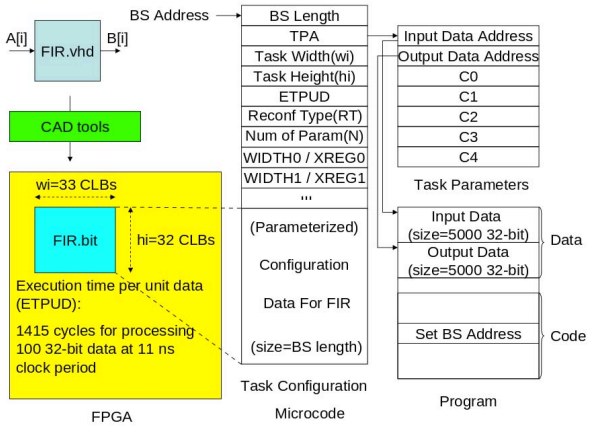

Fig. 2. System at design time

Figure 2 illustrates the detailed 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 [17], shown in the middle of Figure 2. TCM is pre-stored in the memory at the Bitstream (BS) Address. The first field, Bitsream (BS) length, corresponds to the size of the configuration data field. This value is used by the component that fetches the configuration data from memory in order to load them to the configuration port. The Task Parameter Address (TPA) defines where the task input/output parameters are located; this is done using pointers to these locations. In specific, the input data address represents the location of the data which will be processed, and the location where the output data should be stored is defined by the output data address. Task Width and Task Height hold the size of the task expressed in terms of atomic reconfigurable units (e.g. CLBs), while Execution Time Per Unit of Data (ETPUD) has the task execution time per a specific amount of data. Using the ETPUD in conjunction with the size of the input and output parameters, the execution time (in clock cycles) can be estimated. A flag called Reconfiguration Type (RT) specifies whether the bitstream concerns region-based or micro-reconfiguration. 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 decoupling the area model from the fine-grain details of the FPGA fabric, we defined an FPGA technology independent environment where different vendors can provide their consumers with full access to partially reconfigurable resources without exposing the low level details of the bitstream format.

C. Input Requirements

In order to provide the proper inputs to the run-time scheduler, we first explore which parameters should be determined at compile time. The HW tasks are predesigned, i.e. synthesized at compile time, and stored as partial bitstreams in a repository, according to the restrictions of the particular FPGA technology used (e.g., Xilinx). Each HW task is characterized by three parameters: task area (width and height), reconfiguration time, and execution time [17].

In the example of Figure 2, the HW task shown in the left side of the Figure is a simple Finite Impulse Response (FIR) filter. The task consumes input data from array A[i] and produces output data stored in B[i]. The latter is a weighted sum of the current and a finite number of previous values of the input. The filter coefficients (tap weights) are task parameters accessed through the TPA. Other task parameters are the input and output data locations and the number of data elements to be processed. The task implemented in HDL (FIR.vhd) is synthesized by commercial CAD tools to produce the partial bitstream file (FIR.bit) along with the additional synthesis results for that task. The bitstream contains the configuration data that should be loaded into the configuration memory to instantiate the task at a certain location on the FPGA fabric. Synthesis results, or even better results from the stage in which the creation of reconfigurable areas is performed, are used to determine the rectangular area consumed by the task, in terms of configurable logic blocks (CLBs), specified by the width and the height of the task; in the example, task width is 33 and task height is 32 CLBs for Xilinx Virtex technology.

The task should be tested by the designer to determine how fast the input data can be processed. For the example of Figure 2, the FIR task needs 1415 cycles to process 100, 32-bit input data elements at 11 ns clock period making its ETPUD 1415*11 = 15565 ns per 100 32-bit unit data. Based on this ETPUD number, we can compute the expected task execution time for any input data size.

In a realistic scenario, one additional design space exploration step can be added to steer task shapes towards an optimal point, a concept that will be studied within the context of FASTER project. In particular, in a stage of the FASTER tool flow, both task sizes and reconfiguration times are predicted using high-level models; these values will be available in the XML file and will feed the RTSM. A set of parameters should be taken into account in order to predict the reconfiguration time of a certain reconfigurable area: bitstream size, throughput of reconfiguration port, the characteristics of the memory the bitstream is fetched from and of the reconfiguration controller. Other information that should be available in the XML file in order to feed the RTSM is the ETPUD, the Reconfiguration Type (RT) and the parameters for micro-reconfiguration.

The above constitute the input requirements of the RTSM determined at compile time that are not changed at runtime. The following steps of our work will specify the parameters changed at runtime.

VI. Target Platforms and Experiments

Currently, one of the supported platforms is an FPGA-based embedded system on a Xilinx XUPV5 FPGA device. For targeting such platform, we are implementing a C++ framework by merging and extending the ones proposed in [18] and [19]. In particular, with respect to [18], we are introducing the support for reconfiguration and the integration of cores generated by commercial tools or provided by hand. Indeed we defined a common interface (memory-mapped registers to exchange the parameters and memory interface to access the DDR2 SRAM) that has to be respected by all the cores to deal with the runtime manager. In such a way, newer scheduling policies only deal with specific APIs to issue the execution of the tasks, while the implementations details for transferring the data and issuing the command signals are managed at a different layer. With respect to [19], we are introducing automatic and efficient algorithms for performing the scheduling and mapping onto reconfigurable regions, along with a better support for recent FPGA devices and for software cores to be taken into account during the exploration. We also integrated a minimal Graphical User Interface (GUI) to guide the designer during the different phases and minimize the errors while manipulating the XML file.

We applied the proposed framework to design a reconfigurable system for a point-wise filtering algorithm to compute the edge detection. The test application is composed of four main steps: a gray scale conversion (GS), a Gaussian blur filter (GB), an edge detection filter (ED) and finally a threshold phase (TH). Each of these steps can be partitioned to compute on different image blocks in parallel. VHDL code for the cores and their interfaces have been implemented by hand since this is out of the scope for this project. The initial template architecture contains two processing elements (i.e., MicroBlaze soft cores) and a reconfigurable area splitted into two different regions to implement hardware cores. The first processor is used as a scheduler to run the runtime scheduler and manage the dynamic execution of the application, while the second one is in charge of executing software tasks (i.e., reading/writing the image) and perfoming the reconfiguration procedures.

The designer is thus guided by the GUI through the different phases, such as adding the implementations, mapping the tasks, selecting the parameters for the architecture (e.g., memory addresses). In our first prototype implementation, each stage is composed of only one block. GS and ED phases were then mapped onto the first region, while GB and TH ones were mapped onto the second one. This allows to alternate computation and reconfiguration between the two regions. The framework automatically generates the hardware and software specifications fully compatible with the Xilinx ISE Design Suite for the subsequent synthesis. In details, the hardware specification contains the description of the hardware cores, along with the interfaces, and the platform specification file to describe the architecture elements and the interconnections. On the other side, the software specification contains the code that have to be executed by each of the software processors. Note that, minor modifications still need to be applied to this code.

In fact, original functions for reading and writing the image from file need to be updated to use directives for accessing the memory card where we store input/output files.

In addition we have developed a desktop system running CentOS linux which houses a XUPV5-LX110T platform plugged onto a PCIe 1x [14]. This is used to demonstrate the basic operations of a run-time system manager. For demonstration purposes we have designed three simple kernels and one complex 3D Stereo Vision kernel as IP cores for execution in the FPGA; the corresponding bitstreams are stored in the HDD of the host PC. Software running above the host OS awaits input from the user in order to make a selection through a command line interface. Once the user enters a choice, the user level program triggers reconfiguration of the FPGA by loading the corresponding bitstream; if the choice matches the kernel already loaded into the FPGA, reconfiguration is not triggered. Software components of the host PC include the user application and a kernel driver. The user application issues an IOCTL call to send/receive data to/from kernel driver. The driver is responsible for low-level data transfer. Practically, the software is a runtime system that selects a precompiled circuit, loads it to configure the FPGA, initiates the execution and then delivers the results back to the user. The user has no control either on FPGA reconfiguration, communication or execution on the FPGA. In the present system data transactions are performed through DMA achieving a throughput of 1.5 Gbps, which is close to the theoretical bandwidth of PCIe (2Gbps for PCIe lane x1), while reconfiguration is conducted through the JTAG interface using vendor's USB programmer. This is slow and we will move to faster reconfiguration interfaces, e.g. SelectMAP.

In our future activities, we are planning to use more complex platforms targeting the high-performance computing domain, deploy a Global Illumination and Image Analysis application in a desktop system and a Network Intrusion Detection application in an embedded system using the FASTER tool flow.

VII. Conclusions

The FASTER project attempts to enhance several aspects in designing modern computing systems. The main challenge is the inclusion of reconfiguration as an explicit design concept. In order to do so we are proposing new design methods and a tool flow for efficient and transparent use of reconfiguration. We intend to provide seamless integration of parallelism and reconfigurability in the system specification. The tool flow will interface with a run-time system which will be responsible for handling partial and dynamic reconfiguration in an effective manner so as to exhibit better performance over static implementations.

Acknowledgment

This work was supported by the European Commission in the context of FP7 FASTER project (#287804).

References

[1] S. Pontarelli, G. Bianchi, and S. Teofili, "Traffic-aware Design of a High Speed FPGA Network Intrusion Detection System," *IEEE Transactions on Computers*, vol. 1, no. 99, 2012.

[2] I. Sourdis, D. Pnevmatikatos, and S. Vassiliadis, "Scalable Multi-Gigabit Pattern Matching for Packet Inspection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 156–166, February 2008.

[3] http://hartes.org/hArtes/, [Online; accessed October 2012].

[4] http://www.alma-project.eu/, [Online; accessed October 2012].

[5] http://http://www.2parma.eu/, [Online; accessed October 2012].

[6] http://www.reflect-project.eu/, [Online; accessed October 2012].

[7] http://www.madnessproject.org/, [Online; accessed October 2012].

[8] http://www.era-project.eu/, [Online; accessed October 2012].

[9] http://flextiles.eu/, [Online; accessed October 2012].

[10] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs (Invited Paper)," in *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, August 2006, pp. 1–6.

[11] K. Bruneel and D. Stroobandt, "Automatic Generation of Run-Time Parameterizable Configurations," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, August 2008, pp. 361–366.

[12] T. Davidson, K. Bruneel, and D. Stroobandt, "Identifying Opportunities for Dynamic Circuit Specialization," in *Proceedings of the Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS)*, August 2012, pp. 19–21.

[13] 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, and T. Todman, "FASTER: Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration," in *Euromicro Conference on Digital System Design (DSD)*, September 2012.

[14] K. Papadimitriou, C. Vatsolakis, and D. Pnevmatikatos, "Acceleration of Computationally-Intensive Kernels in the Reconfigurable Era," in *IEEE International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, July 2012.

[15] G. Durelli, C. Pilato, A. Cazzaniga, D. Sciuto, and M. D. Santambrogio, "Automatic Run-Time Manager Generation for Reconfigurable MPSoC Architectures," in *IEEE International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, July 2012.

[16] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, "The Molen Programming Paradigm," in *International Workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, 2003, pp. 1–10.

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

[18] C. Pilato, A. Cazzaniga, G. Durelli, A. Otero, D. Sciuto, and M. D. Santambrogio, "On The Automatic Integration of Hardware Accelerators into FPGA-based Embedded Systems," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, August 2012.

[19] A. Bonetto, A. Cazzaniga, G. Durelli, C. Pilato, D. Sciuto, and M. D. Santambrogio, "An Open-source Design and Validation Platform for Reconfigurable Systems," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, August 2012.