A2B: an Integrated Framework for Designing Heterogeneous and Reconfigurable Systems

Christian Pilato, Riccardo Cattaneo, Gianluca Durelli,
Alessandro Antonio Nacci, Marco Domenico Santambrogio and Donatella Sciuto
Politecnico di Milano - DEIB
Milano, Italy
{pilato,rcattaneo,durelli,nacci,santambr,sciuto}@elet.polimi.it

***Abstract*—The constantly growing complexity of heterogeneous systems requires effective methods for supporting the designer both during the development of the application and the implementation of the architecture. Unfortunately, existing tools still require that the designer develops large parts by hand, especially when hardware accelerators and partial dynamic reconfiguration are taken into account.**

This paper presents $A2B$, an ongoing project at Politecnico di Milano, about a semi-automatic framework to assist the designer during the development of heterogeneous and reconfigurable applications for both embedded and high-performance systems. It allows to start from a C-based description of the behavioral specification to be implemented and to perform a progressive refinement of both the designed application and the hardware architecture. It offers the possibility to specify decisions either by an interactive environment or by automatic algorithms, hiding most of the implementation details to the designer.

I. Introduction

Nowadays, embedded systems are very widespread and the continuous growing request of computational power is supported by technology scaling that allows to create larger and larger devices at lower cost. Moreover, devices based on *Field Programmable Logic Array* (FPGA) are becoming very popular and integrated with general purpose processors (e.g., Intel Stellarton, Apple Macbook Pro and Maxeler MaxWorkstation [1], AVNET ZedBoard [2]), but they are also used to create custom architectures themselves, both for embedded systems and high-performance computing.

Moreover, in recent years, there is also a renovated interest for *High-Level Synthesis* (HLS): both academic (e.g., LegUp [3]) and commercial (e.g., Xilinx Vivado HLS, Maxeler MaxCompiler) solutions have been proposed to easily create more and more dedicated IP cores directly from their high-level descriptions (e.g., C/C++/SystemC and also Java) to accelerate the computation. On the other hand, in such a scenario, the proper use of *Partial Dynamic Reconfiguration* (PDR) [4] is becoming crucial to reduce the area occupation by assigning more than one core to the same FPGA region in a time multiplexing fashion. However, to obtain efficient systems, it is necessary to properly hide the reconfiguration overhead and, in general, to take these aspects into account from the early stages of the design process. Unfortunately, PDR is still lacking of complete and integrated design solutions, usually requiring complex manual steps.

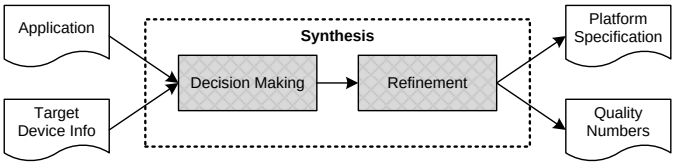

Fig. 1: System-level design methodology, as proposed in [7].

This opens new challenges in *Electronic System Level* (ESL) design [5] for a systematic and concurrent development of the hardware and software components, providing exploration of alternatives along the whole design process. Platform-based design and orthogonalization of concerns [6] are thus very common in system-level design since they allow to decouple the development of the hardware/software application from the implementation of the architecture. Following this principle, ESL design flow is basically composed of two steps: *decision making* and *refinement* [7], as shown in Figure 1. In particular, decision making computes an allocation of resources, a spatial binding and a temporal scheduling, while refinement automatically generates an implementation, consisting of structural models (e.g., RTL descriptions) and quality numbers (timing, area and power consumption). Even if these steps can be separately approached, they require to share several information to converge to high-quality solutions.

In this scenario, Daedalus [8] is a well-established example of modular framework for hardware/software co-design, but limited to loop affine applications and dataflow graphs. It includes tools for automatic partitioning, system-level exploration and automatic generation of the architecture, sharing information through XML files. However, integration of hardware cores and exploitation of PDR are not properly addressed. Different techniques (e.g., [9], [10]) have been proposed instead to support multiple hardware implementations (i.e., trade-offs between resource requirements and execution time), but without proposing an automated flow up to the final system. In the same way, [11] is able to automatically partition the application in hardware cores and software processes, but most of the system generation is still done by hand. On the other hand, existing tools can assist during refinement and efficiently perform logic and physical synthesis (like in Xilinx Design Suite), where decision making is still left to the designer and it requires high expertise.

In summary, most of the existing solutions cover only specific aspects of the problem. We thus strongly believe that a great improvement in the design of heterogeneous and reconfigurable systems can be obtained when the designer will mainly focus on the high-level development of the application and the architecture in an integrated environment, while the low-level details (e.g., generation of hardware and software artifacts) should be hidden to the designer and assisted by automatic and integrated design flows.

This paper proposes $A2B$, a modular and semi-automatic framework for the development of reconfigurable systems. It starts from the application description (currently in C language with OpenMP pragmas [12] to suggest the kernels to be potentially implemented in hardware) and a minimal description of the target architecture, along with information about the target device. Such information is used to properly develop the application by efficiently taking into account the characteristics of the target platform (e.g., communication bandwidth/delay, resources available for hardware implementations, ...). Different phases have been identified to cover the different aspects of the design. Moreover, since we defined a common exchange format based on XML and the framework is organized in a modular way, it results very simple to integrate and compare alternative solutions (either automated algorithms or commercial tools) for each phase without affecting the others. The output is a system specification (both hardware and software) that can be directly used with commercial toolchains (e.g., Xilinx Design Suite or Maxeler MaxCompiler) to actually generate the bitstream and program the target device.

The main contributions of this paper are:

- it presents a semi-automatic framework to design heterogeneous and reconfigurable systems on the top of existing commercial tools;
- it shows how to integrate different solutions for each of the phases, hiding most of the low-level implementation details to the designer;
- it shows how to perform co-exploration of application and architecture to easily generate customized reconfigurable systems.

At the time being, we are working to support both the Xilinx XUPV5 device (for FPGA-based embedded systems) and the Maxeler MaxWorkstation (for High-Performance Computing). We performed different experiments to validate the proposed framework by generating alternative solutions for the same application up the actual execution on the target device and the results look promising. Moreover, we also developed a practical Graphical User Interface (GUI) to assist the designer during the development, suggesting the proper decisions and disabling the ones that would lead to unfeasible solutions.

The rest of this paper is organized as follows. Section II describes the proposed framework and its organization. Then, Section III and Section IV detail the different phases of the design, that are decision making and refinement, respectively. Preliminary results are presented in Section V, while Section VI concludes the paper and outlines future directions of work.

II. A2B: Framework Overview

The aim of the proposed framework is to support the design for three different classes of users: the *application designer*, the *platform architect* and the *system designer*. The *application designer* needs to optimize the application and, thus, to evaluate different solutions of partitioning, mapping and scheduling, including the evaluation of the reconfiguration overhead. The *platform architect* needs to explore the architecture and its parameters in order to evaluate different architectural solutions with respect to the input application. Finally, the *system designer* covers both the phases, needing to concurrently develop the application and the architecture. In this case, an integrated environment is definitively required to proper share the information between the two phases of the design.

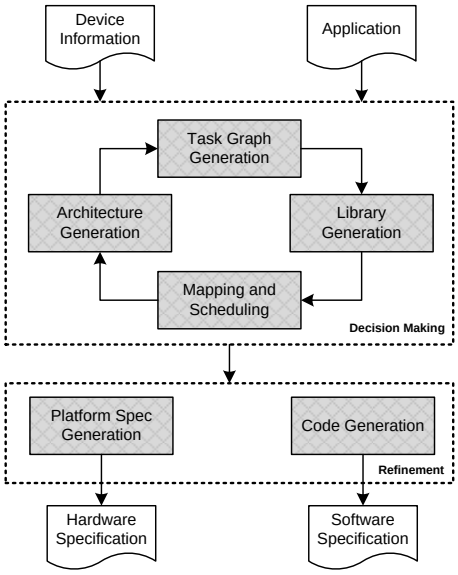

Fig. 2: Overview of the proposed framework, namely $A2B$. Gray boxes represent the identified steps.

The proposed framework, namely $A2B$, follows the idea in [7] and it is thus composed of two different parts, shown in Figure 2: the *decision making* and the *refinement*. The former includes all the phases that are needed for determining the organization of the system and the partitioned application, including its mapping and scheduling with respect to the available components of the architecture. The latter includes the generation of the artifacts for the actual synthesis of the architecture (i.e., both hardware and software specifications).

In details, $A2B$ starts from the application specification (in C annotated with pragmas) and a minimal description of the target architecture (in XML), along with information about the target device. As output, it produces the specification of both the hardware and software specifications. The former includes the definition of the architecture and the hardware descriptions of the cores and their interfaces. The latter includes the software that runs on the top of each general purpose processor. The produced system is fully compatible with commercial tools for the actual synthesis and generation of the bitstream.

To easily share the information among the different phases, $A2B$ adopts an XML file based on the following structure:

- **architecture**: the description (at different levels of abstraction) of processors, hardware logic, memories, IP blocks and their interconnections;
- **application**: the description (e.g., reference to the source code files) and the characterization (e.g., profiling data) of the application;
- **library**: list of hardware and software implementations that are available for each of the application's tasks;
- **partitions**: the description of the partitioned solution (task graph) and its mapping and scheduling with respect to the components of the architecture.

$A2B$ is thus highly integrated and it allows co-exploring architecture and application, if needed. In fact, each algorithm can get information and/or modify each of these parts.

Architecture: As shown in Figure 4, in *A2B*, an architectural template is composed of one or more systems (i.e., a portion of the architecture implementing a given functionality), adopting *communication elements* for their interconnection, (e.g., shared bus, NoC, point-to-point). Since a generic functionality can consist of more than one part, a system can be represented as a set of sub-systems (this is the reason of the self-loop in the Figure 3). Then, in order to implement a generic functionality on a piece of hardware, *processing elements*, *memory elements* and *communication elements* are needed. As it might be notice this schema is really flexible: in fact, using this hierarchical representation, it is easy to represent in *A2B* a large set of architectures, such as Maxeler MaxWorkstation [1] (i.e, Intel i7 CPU connected via PCI-E to a Xilinx Virtex-6) and the AVNET ZedBoard [2] (i.e., a dual-core ARM Cortex-9 and a Xilinx Artix-7 within the same chip) and also FPGA-based embedded systems built on the top of existing commercial devices.

In details, a *processing element* is a component that can implement an application task and it can thus represent a processor, a static IP-core or a reconfigurable region. In order to specify the differences between the different kinds of processing elements, we adopted a set of attributes (i.e., "specs"). For instance, if the designer wants to specify a software processor, the designer can specify parameters like the number of concurrent threads, the ISA and the working frequency. Differently, if the design wants to describe a static IP core, the designer can specify the implemented functionality and the area occupation, if available.

A *memory element* is a generic storage unit that can represent a cache, a read-only ROM, on-chip BRAMs, an off-chip DDR, etc. Again, in order to specify the characteristics of the memory elements, we adopted the "specs" attributes, such as, for example, the number of clock cycles needed to read or write a data. It is clear that changing these two parameters, it is possible to model different memories and thus explore the resulting effects on the program execution. Another important attributes are the size of the memories and the number of read/write ports to model concurrent accesses. Then, based on the adopted communication elements and the specified connections, the memories can be local to some processing elements or shared among them.

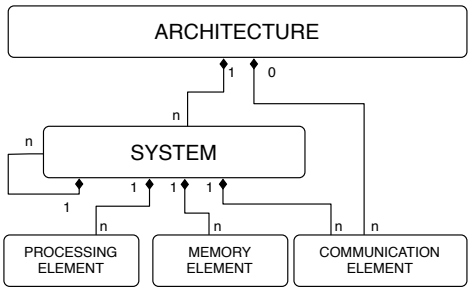

Fig. 3: Hierarchical representation of an architectural template in the *A2B* representation.

Note that the number of the specified parameters highly depends on the level of abstraction that is currently adopted for the exploration. For example, if we are in the early stage of the design, most of the low-level details can be omitted.

Application: The application to be implemented is represented in the A2B-XML as the list of the C files describing the specification, annotated with OpenMP pragmas to identify the tasks to be potentially hardware accelerated and the parallelism inside the specification. Further custom pragmas are adopted to specify additional information (e.g., memory accesses, data bit-width, ...). Then, additional information can be obtained through profiling, performed with custom plugins based on the LLVM compiler[1]. This profiling determines the number of executions for each task and the accessed memory location for each of them. Based on these data, it will be then possible to better understand if a task is a good candidate to be hardware accelerated: indeed, if a function is executed many times, probably it is good to implement it in hardware. Moreover, for streaming application, we also evaluate if the analyzed memory access pattern is compliant with this model of computation. In this way, it is also possible to generate custom pipelines with dedicated memory elements between the different cores to exchange the data.

Library: The library contains all the implementations (software and hardware) for each task of the application under analysis. Each implementation can be considered as a mode for executing the functionality and it specifies details about its execution time and requirements of resources (i.e., number of LUTs, BRAMs, DSPs). Note that an implementation can implement different tasks of the same application. In this case, we can easily model the *hardware reuse* of the logic to implement different functions.

Partitions: This part of the A2B-XML describes the solution for the application. In particular, it describes the partitioned solution (i.e., the task graph) in terms of tasks and their dependencies. Then, it also describes the mapping of these tasks with respect to the implementations and the processing elements, along with a order of execution.

[1] http://llvm.org

III. Decision Making

According to the framework proposed in Figure 2, the decision making step starts from the description of the input application and the information about the target device. As output, it produces an XML file that contains the information about the target architecture and the decisions about the application. Then, it is composed of the following phases:

- **Task Graph Generation**: it extracts and optimizes the task graph for the input application.
- **Library Generation**: it generates admissible implementations (both hardware and software) for each of the tasks.
- **Mapping and Scheduling**: it assigns an implementation and a component for the execution of each task.
- **Architecture Generation**: it determines the topology and the high-level parameters of the target architecture (if the architecture exploration has to be performed);

Note that, if the architecture has been already generated and characterized, the design making phase coincides with the classical platform-based design [6]. On the contrary, it is also possible to perform the design of a custom architecture on the basis of the application to be deployed.

Task Graph Generation: In our current implementation, this part starts from the code of the input application and it extracts the task graph based on the OpenMP pragma annotations, as defined in [13], with a compiler-based approach. Automatic partitioning methodologies ([11], [14]) can be also adopted to generate the task graph, provided to generate the corresponding description in the *partitions* section. In this case, this part may include further steps, as high-level analysis or profiling of the application behavior, to annotate each of the extracted kernels, represented as *functions* in the rest of the framework. Additionally, different transformations (e.g., kernel splitting/merging) are performed to determine the proper granularity of the extracted tasks and trade-off performance and requirement of resources based on the *library* implementations. The tasks and the task graph are then reported into the XML, associated with the corresponding C-based description to be further processed by methods for library generation. Meta-information, such as the number of repetitions for each of the tasks, is also stored along with the required communications. This will be also necessary to properly generate the module interconnections and the run-time system.

Library Generation: For each task , it is possible to generate multiple implementations for the available processing elements. Moreover, considering hardware implementations, it is also possible to derive multiple variants of the same task with different execution times and requirement of resources.

In this phase, we interface with HLS tools (e.g., Xilinx Vivado HLS) for the automatic generation and synthesis of the hardware implementations on the target device. This will allow to determine the latency and the requirements of resources. An alternative possibility is to adopt high-level estimation methods. In this way, depending on the stage of the exploration phase, one can trade-off elaboration time and accuracy by relying, at the beginning, on a high-level analysis tool to obtain a rough estimation of the resource consumption of a hardware task. Then, after this first high-level exploration phase one can synthesize the actual hardware components and obtain final the characterization. The characterization of software implementations can be obtained instead by estimation or profiling methods [15].

All the results are then reported into the *library* section for further use and they are fundamental for the mapping phase in order to effectively explore the design space.

Mapping and Scheduling: To deploy an application on the target architecture, we need to select, for each task, an implementation and a processing element for its execution. Automatic methodologies (e.g., [9], [10]) can be easily integrated into $A2B$: the implementations are contained into the *library* and this phase updates the *partitions* section with the information about the mapping decisions.

For each of the hardware components, the framework is able to determine the proper level of reconfiguration. In particular, if only one core is assigned to only one hardware module (i.e., a portion of the FPGA), it is implemented as a static IP since no reconfiguration is required. Otherwise, if multiple components are assigned to the same FPGA portion, PDR will be adopted to switch between the different implementations. Finally, if two or more tasks share the same implementation on a component, a reconfiguration is not required to switch between the tasks and *hardware reuse* can be exploited. In such a way, the designer (or the corresponding automated methodology) simply determines the mapping of the tasks, without taking care of the reconfiguration details that are automatically managed by the framework.

The mapping methodology uses the information contained in the architectural template that specifies, among the other components, the number of generic processors available in the system as well as an upper bound to the number of reconfigurable regions that the mapper may effectively use in the resulting architecture solution. At the time being, while exploring, the adopted algorithm increasingly instantiates any number of reconfigurable regions deemed appropriate in order to reduce the overall execution time of the application, finding the best trade off between the time saved executing tasks in hardware and the time spent reconfiguring these tasks onto the corresponding reconfigurable regions. Moreover, depending on the nature of the application and the adopted mapping and scheduling, it might even be possible to *mask* the reconfiguration time, as it will be shown in the adopted case study (see Section V).

While finding the trade-off for the best execution time, the adopted algorithm enforces that no resource consumption (for example, BRAMs or LUTs) exceeds the corresponding availability of that resource on the target platform; the computed solution is then admissible by construction. Moreover, if the resulting solution assigns only one task to a specific reconfigurable region, that region is immediately converted to a static hardware module (i.e. non reconfigurable) to save the overhead both in term of area and reconfiguration time.

More specifically, the mapping process is divided into two sequential steps, iterated as many times as deemed required for an extensive design space exploration. During the first phase, it uses a constructive approach to iteratively build a *mapping trace*, i.e. a sequence of *mapping choices* where each application's task is present exactly once. A mapping choice is a 3-tuple composed of a task (from the task graph), a processing element (among software processors, IPs, and reconfigurable regions) and an implementation (among the one available in the *library* section of the XML file). The solution is iteratively built by analyzing the set of currently ready tasks in the task graph (given the currently executed tasks) and computing all the possible mapping choices at each step. This choice represents the execution of a specific task onto a specific processor using a specific implementation of that task onto that (kind of) processor. Afterwards, one of these mapping choices becomes part of the mapping trace by means of a biased random selection process. The choice of a mapping choice implies that the corresponding task is no longer ready (because it has been executed) and the corresponding task dependencies are satisfied, potentially increasing the set of ready tasks. As already specified, in order to guarantee the admissibility of the solution, a constraint on resource consumption is in place such that no choice is considered at any step of the algorithm if it would violate any resource limit.

After all the tasks have been mapped once, the mapping trace is complete. During the second step, the mapping trace is evaluated against any number of performance metrics, which can be easily customized in the framework to target different objectives (e.g., execution time or power consumption). In this work we focused on reducing the overall execution time: for this reason, our metric is the make-span of the application. It takes into account also the impact of reconfigurations. To do this, we implemented a fast, approximate reconfiguration aware scheduler. This component firstly scans the mapping trace to obtain the set of tasks mapped onto every reconfigurable region, then remodels the original task graph adding all the necessary reconfiguration tasks, i.e. between successive tasks scheduled onto the same reconfigurable region. Communication tasks are introduced, too, and considered during tasks' start and end times assignments. After this, it analyzes the dependencies between tasks and simulates their execution on the target architecture. Communications-related considerations (i.e. bandwidth of the bus and total amount of data to be transferred) can be also taken into account. This information is used to guide the exploration towards an improved solution during successive iterations of the optimization algorithm by adding more information to the "biased" random selection.

Note that this part is tightly connected with the floorplanning phase [16], implemented as an inner loop in our methodology. In fact, it is necessary to determine the actual physical regions and evaluate the corresponding task assignment to early identify any violation of the constraints.

Architecture Generation: This part includes all the necessary steps to determine the high-level description of the target architecture, that is the description only with components, interconnections and memories, along with their most relevant parameters (i.e., number of instances, size, ...). Different algorithms can be integrated for the exploration of the architecture based on the application's characteristics. In particular, considering the requirements of the application, it is possible to determine the processors number and area of the FPGA dedicated to hardware cores. Then, it is also possible to explore the interconnection topology (e.g., bus-based, NoC-based or point-to-point) and the memories size as in [17]. This phase thus generates an updated version of the *architecture* section in the XML file to be taken into account by the rest of the flow, allowing a progressive refinement of both application and architecture.

IV. Refinement

The refinement step starts from the solution generated in the previous phase (represented into the XML file) and generates, as output, the hardware and software specifications. This part is thus composed of the following phases:

- **Platform Specification Generation**: it generates the specifications of the hardware part (i.e., component instances and their interconnections), compliant with the backend tools and thus ready for the synthesis;
- **Code Generation**: it generates the specifications of the software part, that is the software code that has to run on each of the general purpose processor.

Note that this part is highly target-dependent since it is based not only on the target device, but also on the tools that will adopted for the synthesis.

Platform Specification Generation: It generates the specification of the hardware part to interface with commercial tools for the synthesis. This part is highly target-dependent since it requires to have information also about the synthesis tools that will be adopted. We thus include a database of components classified by the software tool and the corresponding version. This phase then requires to map the high-level components of the architecture that have been selected in the previous phase to the physical components (e.g., IP cores) in the database. At the time being, we support the generation of the project files for Xilinx ISE Design Suite 14.3 to target Xilinx XUPV5 and for Maxeler MaxCompiler to target Maxeler MaxWorkstation. We are also working to support AVNET ZedBoard with the newer Xilinx Vivado Design Suite. This step is based on the refinement of an architectural template with the specification of the target application obtained through the previous steps.

Concerning the Xilinx XUPV5 architecture the starting template consists in a bus-based system with a shared memory and one or two General Purpose Microblaze processors. The platform specification generation step creates for each of the tasks mapped on hardware the VHDL code is generated by means of HLS techniques and bus and memory (if needed) interfaces are generated. These accelerators are then connected to the base system used as a template.

Maxeler MaxWorkstation is a high-performance system for dataflow computation. In this case the platform generation

step still generates the core for each HW task exploiting Maxeler proprietary HLS tools based on Java descriptions; however these cores are not directly connected to a shared bus, but they are connected one other to form a pipeline. Maxeler proprietary tools permit the generation of the whole hardware system starting from a high-level description of the components without requiring to generate specific interfaces for interconnecting the components or for communicating with an external memory.

Regardless of which target platform is selected, this step analyzes the mapping directives to identify whether or not a hardware component must be implemented as a reconfigurable core or not depending on how many different implementations have been mapped on the core itself.

Code Generation: This step generates the software code to be executed by the general purpose processors, including the software tasks, the run-time manager, including the drivers to control the execution of the cores, the data transfers and the dynamic reconfigurations. Note that, if one task/core requires a dedicated communication protocol to transfer the data, it generates the proper software code according to the protocol description. This is possible since information is shared between the library generation (where the core is created), the mapping (where the implementation is selected) and this part (where the core is instantiated and connected to both the software and the hardware sides).

As for the previous step, also this part is target dependent. Indeed, even if the software code for a task can be generally directly extracted from the starting application, the set of drivers and OS functions and abstractions for scheduling the tasks and for managing the reconfiguration are target-dependent. For the XUPV5 architecture, this step generates the drivers to transfer data to and get data from the hardware tasks and also a scheduler to be executed on one of the Microblazes, along with the drivers to manage the reconfigurations exploiting Xilinx ICAP. On the other side, for the Maxeler MaxWorkstation, this step rewrites the original application code substituting the invocation of tasks starting the streaming computation with the proper API calls provided by MaxCompiler [1].

V. Case Study: Edge Detection

We are implementing $A2B$ in C++ as an extension of the framework proposed in [18]. At the moment, we support the specification for architectures to be synthesized with Xilinx ISE Design Suite ver 14.3 and Maxeler MaxCompiler 2012.1. Note that extending the framework to support different toolchains mainly requires to properly customize the refinement steps. This allows to decompose the problem, with different teams that can work in parallel.

In the following, we will show how the proposed framework can be adopted to easily design a reconfigurable system for a point-wise filtering algorithm to compute the edge detection, where the designer has only to take care of the application development and the toolchain hides some implementation details. 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).

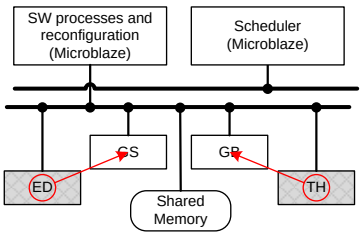

Fig. 4: High-level schema of the evaluated architecture: gray regions are automatically removed in the second experiment.

We initially designed a static architecture, working at 100MHz, implementing all the four computational steps in hardware (designed by hand with a specific protocol for interfacing the memory represented in the XML file as in [19], [20]), while reading and writing the image have been performed by software tasks (assigned to one of the Microblazes). For the sake of simplicity, the runtime execution is managed by another Microblaze processor. The resulting architecture is shown in Figure 4 and it occupies 5,641 slices of the target device, that is a Xilinx XUPV5 FPGA (Virtex5 LX110T – 17,280 slices). Then, we profiled only the execution of these four kernel steps that completed in 151,792,313 clock cycles for an image of 1,024x768 pixels.

After that, we aim at reducing the area of the resulting system by exploiting PDR. For this reason, we analyzed the execution time of each phase, reported in Table I, and we decided to move ED and TH to the regions where GS and GB were executed, respectively. In such a way, we aim at masking the reconfiguration of one region during the execution of the other one. For doing this through the GUI, we simply reduce the area dedicated to hardware acceleration and the mapping strategy automatically assigns different tasks to the same region to save resources. Thus, the *architecture generation* phase automatically infers that they have to be implemented as reconfigurable regions and determines the minimal size for each of these modules, to generate the proper synthesis constraints. Finally, it implements the necessary steps for reconfiguring the functionality at run-time (including the necessary descriptions in the output project file for partial bitstream generation). It also adds the proper elements to the architecture (e.g., ICAP) and it accordingly modifies the software run-time manager to properly issue both the execution and the reconfiguration. We thus generated the new system in few seconds (except for the time required for synthesis and bitstream generation) and it now occupies 5,321 slices since $A2B$ automatically removed the unused regions. However, some extra logic to manage the reconfiguration is required.

Concerning the performance, the two implementations are almost equivalent in terms of overall execution time since: with these mapping decisions we are effectively able to mask the reconfiguration time overhead.

In an analogous fashion, we implemented the same appli-

TABLE I: Execution time and area occupation of each kernel in the Xilinx XUPV5, apart from the core interface that does not require to be reconfigured.

Task	Execution Time [cycles]	Slices
GS	26,824,247	128
GB	58,399,544	276
ED	39,524,613	213
TH	26,896,272	134

cation targeting the Maxeler MaxWorkstation platform. We first modeled the architecture of the heterogeneous system as defined in the specification of the Maxeler MaxWorkstation with a generic processor linked via PCI-Express to a Xilinx Virtex-6 FPGA board. Afterwards, we implemented the same application used in the previous test case as both static and reconfigurable designs. The refinement phase automatically also generates the files (e.g., Java-based descriptions) for the HLS of the hardware cores with Maxeler MaxCompiler.

The application is partitioned so that all the computational tasks are moved to hardware, exploiting streaming computation only internally to each task. In fact, all data movements to the tasks are managed through PCI-Express, as synthesized by the Maxeler platform. For efficiency reasons, the communication to and from the host processor is managed through an additional Virtex-6 FPGA present on the same board, linked to the first through a dedicated channel. The static design features four static regions, one per hardware task. We also provided four different implementations (i.e., VHDL obtained through the Maxeler toolchain) per each task, differing among them only by the amount of pixels that may be processed at once. The more the number of pixels processed at once, the better the performance but the higher the resource usage count. We could provide larger implementations, but due to long synthesis time we preferred to scale the problem down, preferring to constrain the actual available area and synthesizing relatively smaller implementations. Even though the problem is scaled, the same conclusions apply when all the FPGA is employed and larger and more performing implementations are employed.

After running the mapper onto this instance problem for just a few seconds, the optimal mapping is found. This solution is built so that it never violates the maximum resource count of the FPGA. For this experiment, we fix the amount of available area to use to 10000 LUTs. For this reason, the maximally performing ad implementable pipeline is the one processing 4 pixels per clock cycle, with a resource usage count of 7892 LUTs. In fact, the even more performing implementation processing 8 pixels per clock cycle would occupy more than 15000 LUTs. Figure 5 represents the fraction of execution time spent by each applications' task, along with total execution time and time spent for potential reconfigurations (time actually spent only in the reconfigurable design). The bottom part of the figure shows how the total execution time of the static design (when run with an input data of 2500 frames) is 1637,5 ms.

On the other hand, the architectural template of the re-

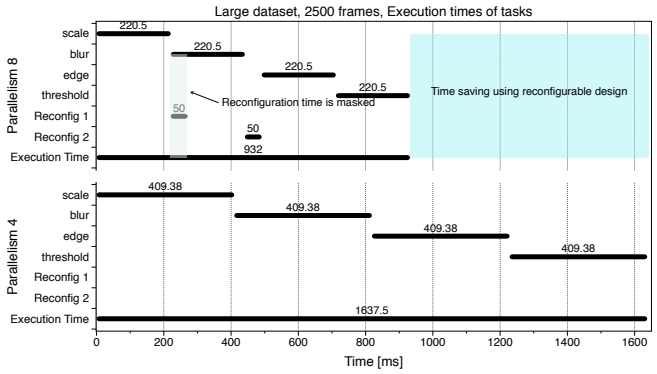

Fig. 5: Parallelism 8 takes less time to compute than parallelism 4, even though two reconfigurations occur (one of which is masked).

configurable system features only 2 reconfigurable regions (R0 and R1) available for tasks mapping. In this test case, the mapping strategy rapidly explores all possible mappings and determines that the best one is R0:$< GS, TH >$ R1:$< GB, ED >$. This mapping corresponds requires 664 LUTs for R0 ($max(LUTs(GS), LUTs(TH)) = max(664.64)$) and $7,680$ LUTs for R1 ($max(LUTs(GB), LUTs(ED)) = max(7,680, 7,376)$), totaling $8,344$ LUTs for the entire hardware system. This resource consumption is comparable to the static architecture. Considering the same inputs size of the case with the static architecture ($2,550$ frames) the corresponding execution time is 882ms, a speedup of 1.86x with an area consumption factor of 1.06x.

The results show that reconfigurable designs do perform better than static designs in specific cases. In the test case we see that the reconfigurable design performs better than the static one when the execution time of a stage in the pipeline is long enough to mask the reconfiguration time of any reconfigurable region and when the adoption of a more resource hungry implementation leads to an improved execution time. In this case the adoption of PDR allows the FPGA to be virtually larger than what it really is, allowing for more performing implementations to be used.

Referring to Figure 5 it is easy to see how with parallelism 8, during the execution of *GB* and *ED*, the reconfiguration time is largely masked, given that tasks are sequential and other tasks cannot execute in parallel. Moreover, using a more resource hungry – and more performing – implementation, if we can execute the application on a device virtually larger than the actual one, which is the case when PR is used, we can potentially design a system more efficient that the static one – which is the case in Figure 5. The conclusion is that under the imposed area constraint of $10,000$ LUTs, the reconfigurable design is the most performing one.

The results show that $A2B$ can be efficiently used to design and prototype complex reconfigurable systems, hiding most of the details to the designer and opening new possibilities for collaborative research thanks to its modular organization.

VI. Conclusions and Future Work

This paper proposed an integrated framework that is currently under development at Politecnico di Milano for the design of reconfigurable systems. It is composed of different steps for the exploration of both architecture and application. It also includes different methodologies and tools to cover the two classical aspects of the system-level design: the decision making and the refinement in order to support the designer as much as possible in the generation of systems ready for the synthesis and the deployment onto the target device. We described how it is possible to integrate different existing methodologies for each of the steps, including the manual intervention of the designer through a practical GUI.

We are currently working on automated methodologies and exploration algorithms for each of the phases (e.g., task graph generation, mapping and scheduling, architecture generation) to develop high-performance systems customized with respect to the applications, along with the integration of commercial tools for HLS (i.e., library generation of the hardware implementations). Finally, we are also evaluating the proposed framework on more complex test cases.

Acknowledgments

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

References

[1] Maxeler Technologies Ltd., "http://www.maxeler.com/."

[2] Avnet Design Services, "http://www.zedboard.org."

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of FPGA '11*, 2011, pp. 33–36.

[4] M. Santambrogio and D. Sciuto, "Design methodology for partial dynamic reconfiguration: a new degree of freedom in the HW/SW codesign," in *Proceedings of IPDPS '08*, 2008, pp. 1–8.

[5] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, March 2007.

[6] A. Sangiovanni-Vincentelli, L. Carloni, F. D. Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proceedings of the 41st Design Automation Conference (DAC 2004)*, 2004, pp. 409–414.

[7] A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, oct. 2009.

[8] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs," in *Proceedings of CODES+ISSS '07*, 2007, pp. 9–14.

[9] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 11, pp. 1189–1202, Nov. 2006.

[10] F. Ferrandi, P. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 6, pp. 911–924, june 2010.

[11] D. Gohringer, M. Hubner, M. Benz, and J. Becker, "A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip," in *Proceedings of FCCM '10*, may 2010, pp. 259 –262.

[12] M. Sato, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in *Proceedings of ISSS '02*, 2002, pp. 109–111.

[13] K. Bertels, V. Sima, Y. Yankova, G. Kuzmanov, W. Luk, G. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti, "Hartes: Hardware-software codesign for heterogeneous multicore platforms," *IEEE Micro*, vol. 30, pp. 88–97, 2010.

[14] D. Cordes, P. Marwedel, and A. Mallik, "Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming," in *Proceedings of CODES/ISSS '10*, 2010, pp. 267–276.

[15] M. Lattuada, C. Pilato, A. Tumeo, and F. Ferrandi, "Performance modeling of parallel applications on MPSoCs," in *Proceedings of SOC'09*, 2009, pp. 64–67.

[16] 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 *Proceedings of FPL '12*, 2012, pp. 707–710.

[17] G. Beltrame, L. Fossati, and D. Sciuto, "Decision-theoretic design space exploration of multiprocessor platforms," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 29, no. 7, pp. 1083–1095, 2010.

[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 *Proceedings of FPL '12*, 2012, pp. 607–610.

[19] R. Nane, S. van Haastregt, T. Stefanov, B. Kienhuis, V. M. Sima, and K. Bertels, "IP-XACT extensions for Reconfigurable Computing," in *Proceedings of ASAP '11*, 2011, pp. 215–218.

[20] T. P. Perry, R. L. Walke, R. Payne, S. Petko, and K. Benkrid, "IP-XACT Extensions for IP Interoperability Guarantees and Software Model Generation," in *Proceedings of FPL '12*, 2012.