2016 IEEE International Parallel and Distributed Processing Symposium Workshops

Scala-based Domain-Specific Language for Creating Accelerator-based SoCs

Gianluca C. Durelli[1], Fabrizio Spada[1], Christian Pilato[2], Marco D. Santambrogio[1]

[1]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy
[2]Columbia University, New York NY, USA
{gianlucacarlo.durelli, marco.santambrogio}@polimi.it
fabrizio.spada@mail.polimi.it, *pilato@cs.columbia.edu*

***Abstract*—Nowadays, thanks to technology miniaturization and industrial standards, it is possible to create System-on-Chip (SoC) architectures featuring a combination of many components, like processor cores and specialized hardware accelerators. However, designing an SoC to accelerate an embedded application is particularly complex. After decomposing this application into tasks and assigning each of them to a processing element, the designer must create the required hardware components and integrate them into the final system. Currently, this process is not well supported by commercial tool flows and has to be manually performed. This is time consuming and error prone.**

This paper proposes a Domain-Specific Language (DSL) based on Scala to specify the architecture of accelerator-based SoCs. We leverage this DSL to coordinate commercial High-Level Synthesis (HLS) tools in order to create the corresponding accelerators with proper standard interfaces for system-level integration.

I. Introduction

Commercial System-on-Chip (SoC) architectures are required to process more and more data and, at the same time, achieve low power consumption. Due to the end of Dennard scaling [1], this problem is particularly critical for mobile devices, where all transistors cannot be powered on at the same time. So, thanks to the continuous transistors miniaturization [2] and the progress of system-level design methodologies, these systems are becoming a complex mix of General-Purpose Processors (GPPs) (to perform application's control flow) and *hardware accelerators* (to implement application kernels at energy-efficient high performance) [3]. For example, the SoC of the latest Apple iPhone 6s includes the M9 co-processor that allows the phone to collect data even if the device is asleep. As a result, the "Hey Siri!" functionality[1] is now implemented in hardware and the functionality is always active without draining the battery.

Reconfigurable hardware, like Field Programmable Gate Array (FPGA) devices, is playing a key role in SoC design [4]. On one hand, thanks to the possibility of being configured multiple times, FPGAs can be used in the design phase to prototype many alternative solutions. On the other hand, they are increasingly used also in final products since it is possible of updating an already deployed solution without changing the physical hardware. For this reason, FPGA vendors are now offering SoC boards that are composed of processor cores and reconfigurable logic in the same chip (e.g. Xilinx Zynq [5]). They are also extending their synthesis toolchains to support the generation of complex hardware accelerators by means of High-Level Synthesis (HLS).

The design flow to create an System-on-Chip (SoC) for accelerating an application can be summarized as follows: 1) the application is decomposed into tasks and hardware/software partitioning is performed to identify hardware tasks [6]; 2) HLS is used to generate a hardware accelerator for each of the hardware tasks; 3) hardware interfaces are generated for each hardware accelerator based on a standard protocol; 4) the resulting components are integrated into the final system and interconnected to the rest of the different components; 5) the application that runs on the GPP is updated to take advantage of the new hardware accelerators. In this context, commercial HLS tools help the designer in automatically translating high-level language descriptions (coming from the input application) into the corresponding Hardware Description Language (HDL) implementations (required to synthesize the accelerators) [4], [7]. On the other hand, there are plenty of alternative solutions to perform Design Space Exploration (DSE) phase in order to decide how to implement each task of the application [6], [8], [9]. Specifically, in order to optimize some design metrics (e.g. performance or the power consumption), these solutions determine which tasks are better to be implemented as hardware accelerators and which ones must to be executed in software by the GPP. Even if all these steps are often supported by tools that can be easily configured by scripts, there is still a significant gap with respect to the application description and the process is not completely automated. As a result, the designer must manually execute all the steps and this process is tedious and error-prone.

Contributions. In this paper we propose a Domain-Specific Language (DSL) based on Scala to specify an SoC architecture with multiple accelerators. We use this DSL to coordinate the generation and the integration of hardware accelerators in an complete system, along with the generation of device drivers. As a proof of concept, we integrate the support for Xilinx commercial tools in order to target the Xilinx Zynq platform, i.e. a modern reconfigurable SoC with an ARM processor and a reconfigurable logic, interconnected through an AXI/AMBA bus.

[1]This refers to the possibility of activating *Siri* (i.e. Apple virtual assistant) just by saying "Hey Siri" instead of pressing a button.

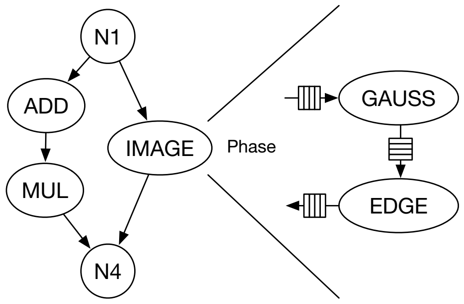


Fig. 1. An example of HTG, which is the representation that we use to describe the input application to be processed.

II. Methodology

This section describes the models that we used for the application and the target architecture, along with the design flow that we use. This design flow leverages a Scala-based tool that coordinates commercial tools for hardware generation and system-level integration.

A. Application Model

The partitioned application used as input to our flow is described as a task graph, where the nodes represent the tasks to be executed and the edges describe the precedences among them. However, in order to generate the desired architecture, we need to provide information about how the data have to be transferred (e.g. parameter copy or data streams). For this reason, we use a two-level Hierarchical Task Graph (HTG) [10], as the one represented in Figure 1.

At the higher level, nodes can represent *simple entities* directly mapped on the tasks of the target applications or to more *complex systems*, called *phases*. In our case, a *phase* represents a set of application tasks which are entirely mapped either in hardware or software. In fact, hardware/software partitioning is performed only at the top level. Each *phase* is then represented by a dataflow graph at the lower level of the hierarchy, where the actors exchange data with each other without the need of communicating with the GPP (except for the initial input and the final output). Specifically, the tasks within a *phase* are connected stream interfaces. These interfaces allow the actors to fire their execution as soon as the minimum amount of data is available, and then their execution is repeated to process all data. Data exchange among nodes is instead performed through shared memory (e.g. DRAM), and each node executes only when the previous nodes have completed their execution and stored the results.

B. Target Architecture

In this work we target reconfigurable hardware devices commonly used in both prototyping stages and commercial products. A renowned example of such architectures (which will be the one used throughout the whole paper) is the AVNET Zedboard, which is a development board featuring the Xilinx Zynq SoC [5] and shown in Figure 2. This SoC is composed of a GPP (i.e. a dual-core ARM Cortex A9) to execute

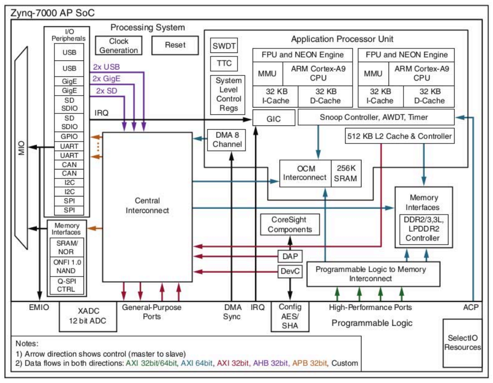


Fig. 2. Block diagram of the AVNET Zedboard, which is the target board for this work.

software tasks and a reconfigurable logic fabric to implement hardware accelerators. The entire system is interconnected by means of a standard protocol (e.g. AMBA/AXI). In particular, in this work, we use two AXI-based interconnections: the AXI-Lite and AXI-Stream protocols [11]. The AXI-Lite protocol supports simple memory-mapped data transfer and is well suited for small chunks of data or single data transfers, like sending commands or parameter values to an accelerator. The AXI-Stream protocol, instead, supports a continuous stream of data, thus reducing the transfer overhead, and is used to move larger amounts of data between the processing elements.

C. Proposed Flow

Figure 3 illustrates the design flow that we follow to port an application onto the target platform. We can identify the following phases in this design flow:

- The initial application is firstly decomposed into tasks and represented by a HTG. This HTG is then manually or automatically partitioned into hardware and software tasks in order to determine the tasks to be implemented as hardware accelerators and the ones to be executed by the GPP, respectively.
- At this point, the generation of hardware accelerators and software tasks can run in parallel: the hardware-oriented flow proceeds with the creation of the HDL descriptions for the hardware components by means of HLS, while the software-oriented one integrates the proper directives in the original source code to interact with the newly created hardware accelerators.
- Finally, hardware and software artifacts are merged and integrated into the final system, and the final bitstream (for the hardware platform) and elf files (for the software binaries) are generated for the target board with vendor-specific synthesis tools.

There are many approaches to automate the hardware/software partitioning and many commercial tools are available to generate the hardware accelerators or to perform logic synthesis of the final architecture. However, drivers generation and system integration are usually not fully automated and the designer has to manually coordinate and execute these steps.

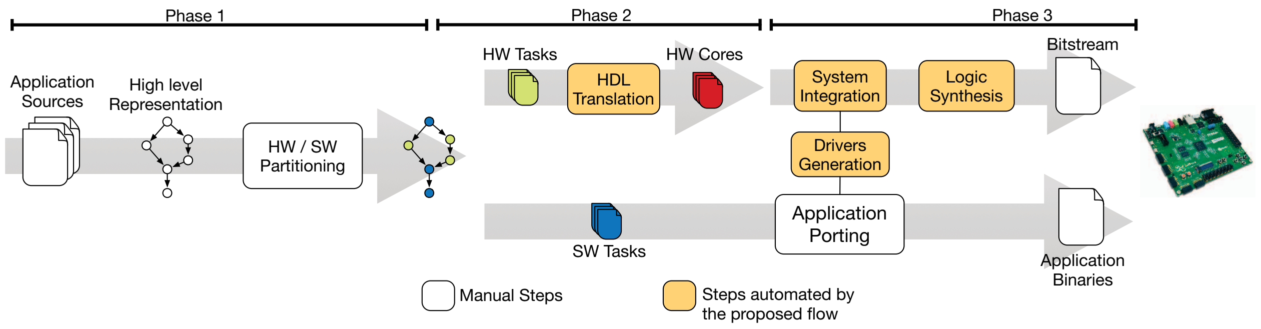


Fig. 3. Design flow used to port a partitioned application onto the target platgorm.

For this reason, we propose a tool-flow shown on the right-hand side of Figure 3. Our flow is based on a DSL that coordinates and automates the generation of the hardware accelerators, the system integration, and the driver generation. partitioning. The starting point of our flow is a high-level representation of an application by means of the HTG model described above. Moreover, the hardware/software partitioning is provided as input and can be manually obtained by the user or with the help of DSE tools [6], [8], [12]. In this work we performed the partitioning manually and we left the integration with DSE tools as a future work.

Furthermore, we provide a synthesizable C/C++ description of each task to be implemented in hardware. Our tool-flow then automatically generates the necessary drivers and interfaces with tools for system integration, and the directives for logic and physical synthesis. Note that, in this work, we use Xilinx Vivado HLS for HLS and Xilinx Vivado Design Suite for system-level integration and synthesis. However, this can be easily extended to support other tools (e.g. Altera Quartus) provided that they support command-line directives or scripts.

III. DSL Definition

As introduced in Section II, the starting point of our flow is the HTG of the target application, where the application nodes have already been partitioned into software and hardware tasks. At this point, to automate the following steps, we need to describe the *single nodes* (i.e. the ones directly representing accelerators), the *phases* (i.e. the dataflow graphs), and the dataflow actors in a representation easy to process. There are plenty of languages to describe a task graph, like XML [13]. However, this requires to implement the CAD tools in another language on the top of an existing library for parsing XML representations. Hence, we use Scala as a DSL to represent the partitioned application, where we can assign complex actions directly to language keywords to be *executed* during the parsing. In this way we can describe our application with the devised DSL and we can then implement the actions to automatically perform the design steps in the same language.

In order to introduce the proposed DSL, we need to have in mind the mapping between the high-level HTG presented before and the final expected output of our Scala-based tool. In fact the actual DSL will reflect more the expected output than the HTG with the part of the DSL describing the links

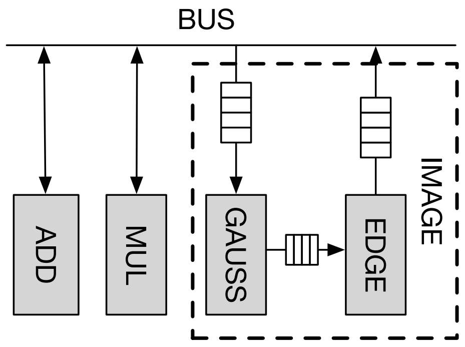


Fig. 4. Example of target architecture. ADD and MULT are connected through AXI-Lite and GAUSS and EDGE filters compose an image-processing pipeline and are interconnected through AXI-Stream.

being related to the actual interconnections. Let us consider the HTG of Figure 1, where nodes N1 and N4 have been selected to run in software, while ADD, MUL, and IMAGE have to be implemented as hardware accelerators. Figure 4 illustrates a high-level overview of the reconfigurable logic part of the corresponding final system. For this reason, nodes N1 and N4 do not appear in the figure. Furthermore we can see that IMAGE does not appear as a core either since it has been substituted by the accelerators corresponding to the actor of the related *phase*. This allows us to specify their interconnection with the rest of the system. Concerning the edges present in the HTG, we specify exactly the ones described in the low-level dataflow graph, while the simple nodes are connected to the main memory through the system bus. Specifically, a task graph $G = \{N, E\}$ for our DSL is described the set of nodes N and the set of communication edges set E. The corresponding DSL will then describe these two lists of elements, one for N (i.e. the hardware cores) and one for E (i.e. the connections). The nodes will integrate information regarding the available input/output ports, while the edges will encode information regarding the type of interconnection (i.e. AXI-Lite or AXI-Stream). In particular, when an edge connects two nodes at the top level, it represents a simple AXI-Lite communication performed by the GPP, while when it connects two nodes in the low-level dataflow graph, the communication is implemented with an AXI-Stream interconnection.

Listing 1. DSL described using EBNF form.

$$\begin{array}{rcl}
\langle\text{DSL}\rangle & \models & \text{object } \langle\text{Project}\rangle \text{ extends App } \langle\text{Graph}\rangle \\
\langle\text{Graph}\rangle & \models & \{\ \langle\text{Nodes}\rangle\ \langle\text{Edges}\rangle\ \} \\
\langle\text{Nodes}\rangle & \models & \text{tg nodes; } \langle\text{Node}\rangle^{+} \text{ tg end\_nodes;} \\
\langle\text{Edges}\rangle & \models & \text{tg edges; } \langle\text{Edge}\rangle^{+} \text{ tg end\_edges;} \\
\langle\text{Node}\rangle & \models & \text{tg node } \langle\text{NodeName}\rangle\ \langle\text{Interface}\rangle^{+} \text{ end;} \\
\langle\text{Interface}\rangle & \models & \text{i } \langle\text{PortName}\rangle \ \mid\ \text{is } \langle\text{PortName}\rangle \\
\langle\text{Edge}\rangle & \models & \langle\text{AXI-Lite}\rangle \ \mid\ \langle\text{AXI-Stream}\rangle \\
\langle\text{AXI-Lite}\rangle & \models & \text{tg connect } \langle\text{PortName}\rangle \\
\langle\text{AXI-Stream}\rangle & \models & \text{tg link } \langle\text{Port}\rangle \text{ to } \langle\text{Port}\rangle \text{ end;} \\
\langle\text{Port}\rangle & \models & \text{'soc} \ \mid\ (\ \langle\text{NodeName}\rangle,\langle\text{PortName}\rangle\ ) \\
\langle\text{Project}\rangle & \models & \langle\text{Letter}\rangle^{+} \\
\langle\text{NodeName}\rangle & \models & \text{"}\langle\text{Letter}\rangle^{+}\text{"} \\
\langle\text{PortName}\rangle & \models & \text{"}\langle\text{Letter}\rangle^{+}\text{"} \\
\langle\text{Letter}\rangle & \models & \text{A} \ldots \text{Z}
\end{array}$$

Listing 2. Nodes list representing using devised DSL for the example of Figure 4

```
tg nodes;
    tg node "MUL" i "A" i "B" i "return"
        end;
    tg node "ADD" i "A" i "B" i "return"
        end;
    tg node "GAUSS" is "in" is "out" end;
    tg node "EDGE" is "in" is "out" end;
tg end_nodes;
```

The EBNF grammar of the proposed DSL is shown in Listing 1. To better describe the syntax of this DSL, we will use the same example of the architecture shown in Figure 4 and described above. This architecture is composed of four cores in hardware (i.e. ADD, MUL, GAUSS, and EDGE): cores *ADD* and *MULT* are interconnected to the system bus with AXI-Lite interfaces. These interfaces are used by the GPP to configure the accelerators. It prepares the data in the shared memory and starts the execution. Then, it waits until the cores complete their execution. The other two cores, a gauss filter (*GAUSS*) and an edge-detection filter (*EDGE*), are instead connected via AXI-Stream and they can start the computation when the minimal amount of data arrives, allowing us to overlap data transfers and computation.

A. Nodes List

The list of nodes for the example is represented in Listing 2. As we can see, the list of nodes is specified with the code between keywords `tg nodes` and `tg end_nodes`. There is one element for each hardware accelerator to be generated, which is described between keywords `tg node` and `end`. It contains information about the node name (between quotes) and the list of ports. Each port is then specified with the type and the port name: `i` stands for AXI-Lite, while `is` stands for AXI-Stream.

Listing 3. Edges list representing using devised DSL for the example of Figure 4

```
tg edges;
    tg link 'soc to ("GAUSS","in") end;
    tg link ("GAUSS","out") to ("EDGE","in
        ") end;
    tg link ("EDGE","out") to 'soc end;
    tg connect "MUL"
    tg connect "ADD"
tg end_edges;
```

B. Edges List

Listing 3 represents the edges list described using the devised DSL. Similarly to the nodes, the edges specification is specified between keywords `tg edges` and `tg end_edges`. For each connection of the desired target architecture in Figure 4, an edge description is specified in the code. Three AXI-Stream links are thus present (see keyword `link`) since there are three streams in the example. For each of them, we specify source and destination ports. Note that the system bus is either source or destination, we use the keyword `'soc` instead. Finally two AXI-Lite connections are specified for the accelerators *MULT* and *ADD* and identified by the keyword `connect`.

IV. System Integration

After the description of the DSL, in this section, we describe the implementation of the flow presented in Section II, delving into the details of the automated steps carried out to perform the system integration phase which produces a valid bitstream for the target device and the elf files for the software part.

A. Overall Automation Flow

The proposed Scala-based executable representation requires that we provide as input a file compliant with the DSL described in Section III and a synthesizable C/C++ file compliant with Vivado HLS for each node. Starting from these elements, we "execute" Scala program associated with the task graph description. This program coordinates Vivado HLS and Vivado Design Suite to generate the accelerators with the proper interfaces and integrate them in order to generate to complete bitstream for the specified architecture.

Figure 5 illustrates the steps performed by the tool. Starting from the input described above, each of the application functions is translated by means of HLS into the corresponding VHDL representation of the hardware accelerator. After creating the hardware accelerators, the final system is generated integrating each one of these cores into the system. Finally the commercial tool-flow is invoked to execute *synthesis*, *map*, *place and route*, and *bitstream generation*.

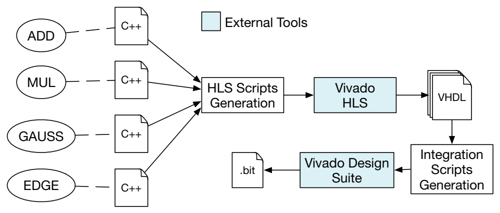


Fig. 5. Integration of the devised tool with Xilinx toolchain.

Since our current implementation targets the AVNET Zedboard through the Xilinx Vivado Design Suite, our tool automatically performs the following steps to have a complete working system:

- it adds a Zynq Processing System (PS),
- it configures the Zynq PS and enables the High Performance (HP) I/O ports to transfer data via Direct Memory Access (DMA).
- it adds a DMA core for managing I/O via AXI-Stream.

During the rest of the execution, the Zynq Programmable Logic (PL) is customized by adding all the generated hardware accelerators and their interfaces with the PS.

B. DSL Execution Steps

After describing the steps at high level, we now present how such steps are carried out when the DSL representation is executed; Figure 6 illustrates these steps. Note that each one of the keywords defined in the DSL is an executable function. The nodes section of the program will generate the commands for the integration within Vivado HLS; while the edges part perform the automation of the system integration step. The execution of the program evolves as follows:

1) the function `nodes` creates a new Vivado project;
2) the function `node` creates a new instance of the Node object for the specific node and adds it to the list of available nodes; this function also generates the *tcl* code to create a new Vivado HLS project;
3) for each of the interfaces the function associated with either `i` or `is` is executed to generate the proper interface and add it to the list of interfaces; furthermore it adds the proper specifications for the interface under analysis to the *directives* file;
4) upon the execution of the keyword `end`, the tool invokes the synthesis of the hardware core through Vivado HLS; after the synthesis is completed, the control returns to the TaskGraph class to process the next node;
5) the `connect` function generates the code to connect the AXI-Lite interfaces to the system bus;
6) upon the execution of the *link* function, a new instance of the Link class is created;
7) the `to` function inside the Link class creates the *tcl* script to connect the AXI-Stream interface either to another AXI-Stream interface or to the DMA core;
8) the function associated with `end_edges` executes the *tcl* for the Vivado project, executing all the steps up to the bitstream generation; then it invokes the part of the tool responsible for API generation.

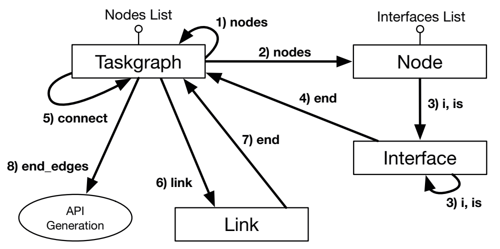


Fig. 6. Flow carried out while executing the DSL.

V. Software Generation

After generating the bitstream, we also generate the API to configure and invoke the accelerators from a software application. We also generate the files needed to boot the board using a pre-compiled version of the PetaLinux Operating System. Specifically, the boot file generation process first produces the files needed to start the board with Linux and then customizes the *device-tree* used by Linux. In this way, the Linux kernel automatically recognizes the new hardware accelerators and the corresponding DMA cores; the resulting *device* file is thus placed into the `/dev` directory.

For AXI-Stream connections, we do not provide a complete API wrapping the execution of the function. Instead, we provide a device driver that we developed internally and pre-compiled inside the Linux image[2]. This device driver can be used to perform data transfers between the ARM processor and the reconfigurable logic via DMA. In fact, we provide two simple APIs (*readDMA* and *writeDMA*) to move data after opening the corresponding device in the `/dev` directory.

VI. Case Study

This section presents a simple case study to show how we used the devised environment to seamlessly generate alternative implementations of a given application. Specifically, we performed design space exploration by testing the different solutions on the target board.

A. Application Description

The application used as the case study implements the Otsu filter for binary image segmentation. This filter is usually used as a preprocessing step for further image analysis and allows for automatically performing clustering-based image thresholding. As an example, Figure 7b reports the results of applying the Otsu fiter to the grayscale input image shown in Figure 7a. This application is composed of six tasks:

- *readImage* loads an image from file,
- *grayScale* converts the image into gray scale format,
- *histogram* computes the histogram of the image,
- *otsuMethod* finds the threshold used to filter the image,
- *binarization* constructs the binary image starting from the original gray scale image and the threshold;
- *writeImage*: writes the resulting image into a file.

[2]Available at: https://github.com/durellinux/ZedBoard_Linux_DMA_driver

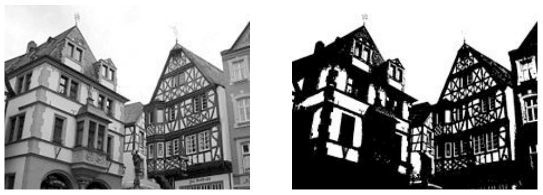

(a) Original image (b) Filtered image

Fig. 7. Example of the application of the Otsu filter.

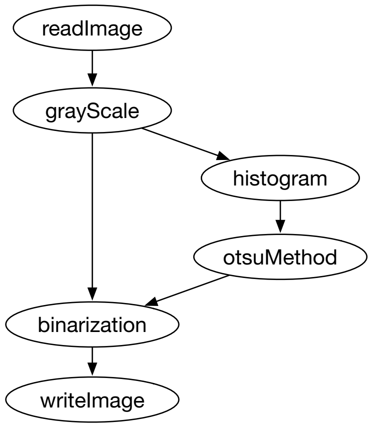


Fig. 8. Dependency graph of the Otsu filter used as a case study

Figure 8 represents the dependency graph of the application. Every task, except for *readImage* and *writeImage* that perform I/O, can be implemented in hardware. In the next subsection, we will perform the automatic creation of multiple architectures for this example varying the mix of cores to be used in hardware.

B. Results

We generated four descriptions of the application using our DSL. Each description corresponds to a different architecture to be synthesized and implements a different number of functions as hardware cores. Table I reports a summary of which functions have been implemented as hardware cores. All the hardware cores implementing the application function make use of AXI-Stream interface. As an example, we report in Listing 4 the description in the proposed DSL for *Arch4*, which corresponds to implement all the functions in hardware.

Starting from the four files describing the solutions in Table I and the corresponding C code compliant with Vivado HLS, we automatically generated the four hardware architectures. The total time needed to generate all the solutions is 42 minutes and Figure 9 reports the time breakdown of the different phases. Note that the generation of the hardware cores is done only once for each function and, for efficiency, we first generated *Arch4* that has all the functions implemented in hardware. The time needed to compile the Scala code is reported as SCALA.

All the architectures for the devised case study have been synthesized successfully. As a result, our design flow automatically created the four bitstream files and the corresponding boot files that can be used to test the application on the board. Table II reports the resource requirements for the generated solutions.

TABLE I
SUMMARY OF THE AUTOMATIC GENERATED IMPLEMENTATION FOR THE CASE STUDY.

Solution	grayScale	histogram	otsuMethod	binarization
Arch1		✓		
Arch2			✓	
Arch3		✓	✓	
Arch4	✓	✓	✓	✓

Listing 4. Description of *Arch4* using the DSL proposed in this work.

```
object otsu extends App {
  tg nodes;
    tg node "grayScale" is "imageIn" is "
        imageOutCH" is "imageOutSEG" end;
    tg node "computeHistogram" is "
        grayScaleImage" is "histogram" end;
    tg node "halfProbability" is "
        histogram" is "probability" end;
    tg node "segment" is "grayScaleImage"
        is "otsuThreshold" is "
        segmentedGrayImage" end;
  tg end_nodes;

  tg edges;
    tg link 'soc to ("grayScale","imageIn
        ") end;
    tg link ("grayScale","imageOutCH") to
        ("computeHistogram","grayScaleImage
        ") end;
    tg link ("grayScale","imageOutSEG") to
         ("segment","grayScaleImage") end;
    tg link ("computeHistogram","histogram
        ") to ("halfProbability","histogram
        ") end;
    tg link ("halfProbability","
        probability") to ("segment","
        otsuThreshold") end;
    tg link ("segment","segmentedGrayImage
        ") to 'soc end;
  tg end_edges;

}
```

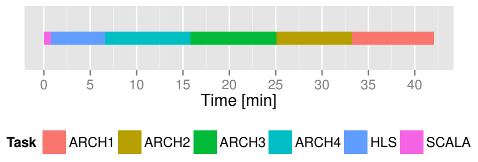
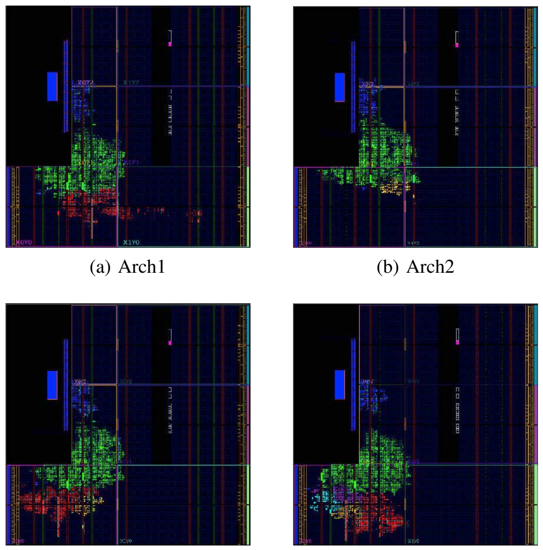


Fig. 9. Time breakdown of the different actions needed to generate the four architectures of the case study.

TABLE II
RESOURCE USAGES OF THE FOUR DIFFERENT SOLUTIONS IMPLEMENTED IN THE CASE STUDY.

Solution	LUT	FF	RAMB18	DSP
Arch1	3809	4562	5	0
Arch2	7834	9951	4	2
Arch3	8190	10234	5	2
Arch4	9312	11256	5	3


(a) Arch1 (b) Arch2

(c) Arch3 (d) Arch4

Fig. 10. Architectures generated by the tool for the analyzed case study: ARM processor and bus in blue, DMA related blocks in green, *otsuMethod* in red, *histogram* in orange, *grayScale* in light blue, and *binarization* in purple.

C. Discussion

The automatic generation of the bitstreams with our toolchain allowed us to quickly speed-up the development process. In fact, it avoided us the burden of using the Graphical User Interface (GUI) manually to perform the interconnections and the cores customization, or write the proper *tcl* scripts.

A first consideration that we can draw from the test we performed is that our tool is able to generate the whole Vivado project ready to be synthesized in less than one minute (worst case for the analyzed case study). Indeed, it takes about 6 seconds to compile the Scala task graph and then 50 seconds to generate the corresponding Vivado project. To compare this time with the time required to develop the Vivado project through the GUI. We started a timer when we invoked *vivado* through the command line and we confirmed all the default options without making any choice, except selecting the AVNET Zedboard as the target device. After 48 seconds, we were only able to instantiate the Zynq PS, and we still had to add the repository for the HLS cores, add all the generated cores, and perform the interconnections. Clearly, our tool bring advantages over using the GUI.

However, since the tool leverages *tcl* commands, it is also necessary to look at how the tool compares against using the commands directly in *tcl*. In this case, we compared the code that the designer has to write in order to generate the target system. In our case study, the *tcl* script generated by our tool, which is the one that the designer is supposed to write, has 4 times more lines of code than the starting Scala task graph representation. If we look deeper and analyze the actual characters that have to write, we realized that the *tcl* script requires 4 to 10 times more characters than the Scala-based representation.

Furthermore, we have also to keep in mind that, from an application designer viewpoint, specifying the application task graph in Scala is much simpler since it is at a higher level of abstraction and it corresponds only to the application functionality. On the contrary, the *tcl* script needs to delve deeper into hardware details and actual Xilinx tool-flow commands, which might be unknown to the application designer. Even a hardware designer requires some expertise and familiarity to use to the GUI.

Finally, it is also worth noting that the developed tool is easily maintainable. We started its development with Vivado Design Suite ver. 2014.2 and we easily ported the proposed DSL to support Vivado Design Suite ver. 2015.3 in less than a day. To perform this update, we had simply to change the backend that generates the *tcl* scripts by upgrading the versions of the cores used by Vivado Design Suite and updating a few commands that have been changed across the two versions. In general, since it is unlikely that Xilinx completely revolutions the *tcl* commands, the tool is very easy to maintain across multiple versions of the toolchain.

VII. RELATED WORK

The high complexity of the SoC design flow leaded to a continuous improvement of Computed-Aided Design (CAD) tools used to assist the designer in the porting of an embedded application onto the target SoC. Both commercial and academic tools have been developed for the different phases of the design process.

Generally the design flow for SoCs starts with partitioning the application to determine which tasks have to run in software (on one of the GPPs) and which ones have to be mapped on the reconfigurable logic. This step, called *hardware/software partitioning*, has been extensively explored by academic tools such as Daedalus [14], [15] and RAMPSoC [16]. After hardware/software partitioning, the designer must create the cores for the tasks to be executed in hardware. Few research frameworks integrate a HLS step in their flow and generally require manual intervention in this step. However research is also active in the study of new HLS solutions alternative to the commercial ones [4], [17]. After the partitioning, the hardware cores must be mapped onto the available computational resources and, for this reason, some tools focused on the definition of a template architecture (*base interconnection*) and then performed automatic mapping of hardware cores onto this template [8], [18], [19].

Finally vendor tools must be invoked to generate the bitstream to configure the target FPGA device. To this purpose vendor tools mainly focused on two of the steps which are the simplification of the system integration via a GUI, incorporating in the tool all the complexity of the bitstream generation steps. However, in many cases the actual integration is manually performed, even when a clear description of the system is available from the mapping step described above.

Compared to the works presented above, this paper proposes a tool which allows us to completely automate the integration and invocation of vendor-specific EDA tools. In fact, even we currently target only Xilinx tools, we have shown that the tool can be easily modified to support newer versions or different tool-flows. Furthermore, this paper does not target template architectures, which have to be maintained, but interconnection mechanisms based on standard protocols, such as the AX-I/AMBA bus. In fact, standard-based interconnections will be more and more used given the success of these SoC platforms. Finally, the software layer requires to correctly boot the board with a Linux OS capable of exploiting the new hardware devices. Its generation is usually manually performed, which is tedious and error-prone.

Vendors are also very active in solving the issues tackled by this work; in fact at the same this paper was under publication, Xilinx commercialized a tool that addresses most of the issue presented here, called SDSoC. This tool is basically an IDE where the designer writes the application code in C/C++ and then tags which functions has to be implemented in the reconfigurable logic. With a series of pragmas, it is possible to control how the IP cores are synthesized through Vivado HLS directly from the IDE and how to generate the set of communication links and perform the data transfers in order to move the data between the ARM processor and the IP cores. Furthermore it is also possible to control, always by means of pragmas, how the runtime system is generated. For instance, it is possible to instruct how caches have to be managed during data transfers and if how to execute more instances of an IP core in an asynchronous way to exploit task parallelism. The SDSoC tool produces a set of files that can be put on the SD card to directly execute the application as specified in the IDE, without manually writing any specific code to target the HW architecture automatically generated by the system. However, our tool has still some advantages with respect to Xilinx SDSoC in many designs. In particular, given a function with N vectors as parameters, SDSoC instantiates a DMA component for each of them. This solution generally leads to unnecessarily increase the resource requirements, while it is generally possible to use a single channel to send input data to the core, store them locally, and then produce the output. At the time being, this feature is not supported by Xilinx SDSoC, while in our tool the designer simply specifies a single input channel in the Scala task graph and then writes the runtime code to perform the desired write pattern on this channel.

VIII. Concluding Remarks

In this paper we presented a DSL based on Scala to specify accelerator-based systems. In our representation, we can specify accelerators and interconnections (either memory-mapped or streaming), and automatically interface with vendor tools for HLS and system-level integration.

We used this DSL to specify a system for an image processing application. We showed that, with our approach, we are able to generate alternative architectures with very limited effort for the designer.

Acknowledgments. This work was partially funded by the European Commission in the context of the FP7 SAVE project (#610996-SAVE).

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proc. of ISCA*, pp. 365–376.

[2] R. R. Schaller, “Moore’s law: past, present and future,” *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[3] M. Horowitz, “Computing’s energy problem (and what we can do about it),” in *Proc. of ISSCC*, Feb 2014, pp. 10–14.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for FPGAs: From prototyping to deployment,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[5] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[6] 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 CAD of Integrated Circuits and Systems,*, vol. 29, no. 6, pp. 911–924, June 2010.

[7] F. Winterstein, S. Bayliss, and G. A. Constantinides, “High-level synthesis of dynamic data structures: A case study using vivado hls,” in *Proc. of FPT*, 2013, pp. 362–365.

[8] R. Cattaneo, C. Pilato, G. C. Durelli, M. D. Santambrogio, and D. Sciuto, “SMASH: A heuristic methodology for designing partially reconfigurable MPSoCs,” in *Proc. of RSP*, 2013, pp. 102–108.

[9] R. Piscitelli and A. D. Pimentel, “Design space pruning through hybrid analysis in system-level design space exploration,” in *Proc. of DATE*, 2012, pp. 781–786.

[10] M. Girkar and C. D. Polychronopoulos, “The hierarchical task graph as a universal intermediate representation,” *International Journal of Parallel Programming*, vol. 22, no. 5, pp. 519–551, 1994.

[11] A. ARM, “Axi protocol specification (rev 2.0),” *Available at http://www. arm. com*, 2010.

[12] F. Ferrandi, C. Pilato, D. Sciuto, and A. Tumeo, “Mapping and scheduling of parallel C applications with Ant Colony Optimization onto heterogeneous reconfigurable MPSoCs,” in *Proc. of ASP-DAC*, 2010, pp. 799–804.

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (xml),” *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, vol. 16, 1998.

[14] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, “System design using khan process networks: the compaan/laura approach,” in *Proceedings of DATE*, 2004, pp. 340–345.

[15] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, “Daedalus: toward composable multimedia mp-soc design,” in *Proc. of DAC*, 2008, pp. 574–579.

[16] D. Göhringer, M. Hübner, V. Schatz, and J. Becker, “Runtime adaptive multi-processor system-on-chip: RAMPSoC,” in *Proc. of IPDPS*, 2008, pp. 1–7.

[17] C. Pilato and F. Ferrandi, “Bambu: A modular framework for the high level synthesis of memory-intensive applications,” in *Proc. of FPL*, 2013, pp. 1–4.

[18] K. Sigdel, M. Thompson, A. D. Pimentel, T. Stefanov, and K. Bertels, “System-level design space exploration of dynamic reconfigurable architectures,” in *Proc. of SAMOS*, 2008, pp. 279–288.

[19] D. Göhringer, O. Oey, M. Hübner, and J. Becker, “Heterogeneous and runtime parameterizable star-wheels network-on-chip,” in *Proc. of SAMOS*, 2011, pp. 380–387.