Speeding Up Robot Control Software Through Seamless Integration With FPGA

Xuan Sang LE[1,2], Luc Fabresse[1], Jannik Laval[3], Jean-Christophe Le Lann[2], Loic Lagadec[2] and Noury Bouraqadi [1]

[1]Mines Douai-DIA, Univ. Lille
[2]ENSTA Bretagne
[3]DISP Laboratory, University Lumière Lyon 2

Context

Today robotic computing systems are usually implemented using general purpose processors because of their accessibility and simplicity which do not require specific knowledge. However, this approach restricts several optimization opportunities and may not always satisfy performance, cost, and energy requirements [5]. FPGA infrastructures can be considered as a good solution for these issues, especially in complex robotic systems that require time consuming tasks. Advantages are many to use FPGAs along with general purpose processors. On the one hand, FPGAs provide hardware acceleration and on the other hand, CPUs allow developers to use flexible software development environments. However, programming hardware devices requires a specific knowledge which remains a challenge for developers and usually results in a loss of productivity [3]. Moreover, interfacing FPGAs and high-level software remains problematic. This problem varies depending on projects due to the change of requirements and takes an important amount of development time. These difficulties are a barrier for a widespread adoption of FPGA in systems such as robotics ones.

In this work, we present a generic architecture along with a toolset and libraries to ease the integration of FPGAs into existing robotic system. We provide a mean to connect the FPGA to the system and automatically deploy VHDL code that can be either legacy VHDL code, or generated VHDL code from a HLS tool. Developers do not have to worry about the interface between the FPGA and the processor. This interface is automatically generated according to the VHDL code. On the high-level software side, our software platform allows direct access to circuit registers on FPGA as memory addresses (memory mapping). To demonstrate this proposal, we describes a complete follower robot as a case study. The image processing to achieve a colour-based object detection is deployed on a FPGA. The rest of the robot controller runs on a CPU and has been developed using an object-oriented programming language (Pharo Smalltalk).

Our approach

We consider the following scenario: Developers have a robotic system that embeds a FPGA and they have some VHDL legacy code that performs some time-consuming tasks. The aim is to provide an easy way to (1) deploy this code to the FPGA, and (2) to directly access the result on the FPGA from a high-level robotic software framework (such as the Robotic Operating System, ROS) without worrying about the communication between the FPGA and the software.

The proposed work abstracted the communication between the low-level hardware and high level software using an auto-generation code approach. The goal is to reduce the development time by limiting as much as possible the work on hardware (FPGA) and focusing more on the functionalities of the application.

Figure 1 shows the general architecture of our approach. The VHDL legacy code is first imported to the system with the help of a dedicated VHDL Parser. The parser processes any valid VHDL code and automatically generates a corresponding circuit model of that design. This circuit model is described using the meta-model presented in [8]. This meta-model captures the subset of synthesizable VHDL structures [6]. From this meta-model, developers can express descriptions i.e. models of digital circuits as plain objects that can automatically be processed and exported to VHDL code if needed.

The meta-model allows to compose arbitrary circuit models (imported from VHDL) via predefined interfacing specifications. Concretely, a generic interfacing specification (Wishbone) is built using the meta-model, this specification takes the user circuit model as input and generates a corresponding interface model based on its structure. The circuit model is then integrated with the generated interface model to form the final model. This latter can be automatically exported to VHDL and deployed on FPGA. Meanwhile, on the software side, the meta-model generates also the wrapper classes based on user input design. These classes grant user software access to circuit registers (on FPGA) as normal objects.

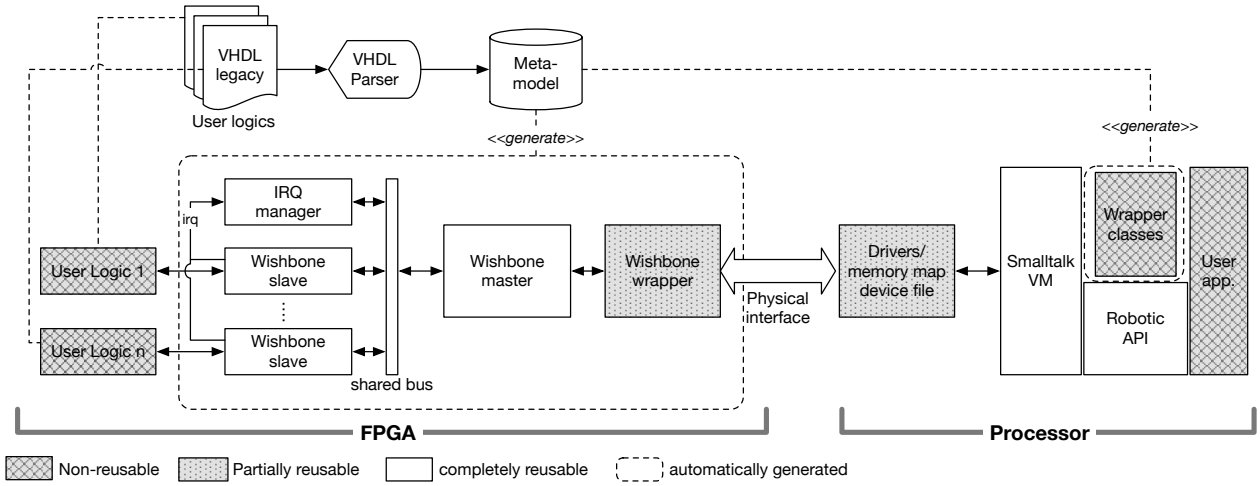

Figure 1: Auto-generation code is the heart of our work with the help of the meta-model [8]. The hardware interface is generated based on input VHDL; software applications can directly access to circuit registers via generated wrapper classes.

The solution is highly reusable and generic since the interfacing specification is made at the meta-model level. The only modification needed is when the physical interface between the FPGA and the processor changes. In this case, both hardware wrapper and the low-level driver needs to be modified correspondingly. These changes are made once, and then can be reused for different projects using the same physical interface.

Hardware interface

The software framework provides access to the FPGA circuit registers as a virtual memory region using a memory mapping technique. Each FPGA appears as a device file on the processor. The software framework maps this file to a segment of virtual memory. This permits the user applications to treat the FPGA as if it is part of the primary memory. An addressing mechanism is therefore needed on FPGA to map each circuit register to the corresponding address [9].

Accessing circuit registers by address requires an address/-data IO interface between the FPGA and the software framework. On the processor, this relies on the dedicated driver. On the FPGA, we use the Wishbone bus to provide such mechanism. Each user logic is mapped to a wishbone slave. These slaves are controlled by a wishbone master that takes an address from software and decides which slave to be activated. This master is connected to a wrapper close to the physical interface which is used to adapt the physical interface to Wishbone interface.

An IRQ (interrupt) manager is added to the interface to allow software to react to the changes raised on hardware side.

This Wishbone interface is a part of the meta-model's interface specification. It will be generated automatically based on the input target circuit.

Software framework

The software framework views each circuit register as a memory segment which value can be accessed by an address. As shown in the figure 1, this is achieved using the low lever hardware driver. This latter is the user space I/O (UIO) driver dedicated to the physical interface. On the first hand, it handles the communication between FPGA and the processor. On the other hand, it provides a generic device special file to the high-level software that can be mapped as a virtual memory region. This driver is hardware specific and must be updated accordingly when we adopt a new physical interface.

Thanks to the low level driver, the higher software layer, now considers the FPGA as a virtual memory region. Every FPGA register can therefore be read/written via its corresponding virtual address. The generated wrapper classes ease this access by: (1) automatically addressing the FPGA registers and (2) providing an abstract way to read/write these registers.

Multiple circuits can run in parallel on FPGA and each is assigned to an independent virtual memory segment. Our software framework supports multi-process at language level, it's easy to map each circuit on FPGA to a equivalent process.

Apart from the memory mapping, the framework also offers different features: (1) a integrated ROS client that allows software to easily communicate with a ROS network; (2) a mini integrated web-server that allows users to instantly develop, inspect or even execute their applications from browser. This latter is especially helpful since, for most robotic system, the software framework is often run in headless. Writing a dynamic code and execute it from distance is a good solution for rapid robotic prototyping and experiment.

Controllability and debugging

Our platform offers software-like debug capabilities on hardware using the concept of dynamic hardware breakpoint presented in [8] into the framework. The idea is that, when the meta-model generates the slave for a target circuit, if needed, a debug specific sub-circuit will be injected automatically into that slave. This sub-circuit allows the software to control the execution flow of the target circuit by setting a dynamic breakpoint.

The execution of the circuit will be stopped when the breakpoint condition becomes true. In this state, software can inspect the registers value as well as the execution time (in clock cycles).

Software can resume the halted execution at anytime by writing a true boolean to the *resume* register on the debug subcircuit.

Verification & demonstration

The VHDL parser has been verified using some standard benchmarks: ANTRL [7], IWLS 2015 [2] and ITC'99 [4]. The verification of the meta-model is much more challenge since, at the moment, only 60% of VHDL structures is implemented. However, in the purpose of automated configuration and integration, it is not necessary that all the VHDL code should be imported and modelled. The interface specification needs only the external interface (entity) of the topmost design entry. Our interface specification is similar to IP-XACT [1], a well-know standard for this kind of problem.

To demonstrate our platform in action, we have built a robot follower application that use a camera and FPGA for object detection. The FPGA takes the image from the camera and filters each pixel (using a hardware HSV filter) by a specific colour pattern. The filtered pixels are then used to calculate the barycenter of the detected region which finally provides the position of object. The VHDL design (reused from another project) has been imported to out framework for interface generation and registers addressing. The wrapper software classes for FPGA accessing has been are also generated. These tasks are completely automatic. The only task that we have needed to manually do is to create a simple configuration class for IOs specification of the target circuit.

Via the wrapper classes, software can easily collect the object position from the FPGA and stream it to the ROS network. The controller software (compliant to ROS) uses this information to follow the object.

From the developers perspective, generated wrappers classes are all they need to access to FPGA (as an oriented-object manner). The system abstracts all the communication and hardware accessing that are performed transparently.

References

[1] Ieee standard for ip-xact, standard structure for packaging, integrating, and reusing ip within tool flows. *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009)*, pages 1–510, Sept 2014.

[2] C. Albrecht. Iwls 2005 benchmarks. Technical report, June 2005.

[3] D. F. Bacon, R. Rabbah, and S. Shukla. Fpga programming for the masses. *Commun. ACM*, 56(4):56–63, Apr. 2013.

[4] F. Corno, M. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *Design Test of Computers, IEEE*, 17(3):44–53, Jul 2000.

[5] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang. Computing Performance Benchmarks among CPU , GPU , and FPGA.

[6] IEEE. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. Technical Report October, 2004.

[7] T. Parr. ANTLR v4.0.

[8] L. X. Sang, L. Lagadec, L. Fabresse, J. Laval, and N. Bouraqadi. A meta model supporting both hardware and smalltalk-based execution of fpga circuits. *IWST'15*, 2015.

[9] B. Wile. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems White Paper. Technical Report September, IBM Systems and Technology Group, 2014.