On-board non-regression test of HLS tools targeting FPGA

Arief Wicaksana, Adrien Prost-Boucle, Olivier Muller, Frédéric Rousseau, Arif Sasongko

On-Board Non-Regression Test of HLS Tools Targeting FPGA

Arief Wicaksana, Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau
TIMA Laboratory
Université Grenoble Alpes
Grenoble, France
{firstname.lastname}@imag.fr

Arif Sasongko
School of Electrical Engineering & Informatics
Institut Teknologi Bandung
Bandung, Indonesia
asasongko@stei.itb.ac.id

ABSTRACT

High-Level Synthesis (HLS) has opened an opportunity for software programmers to target FPGA more rapidly. When developing HLS tools, tests are desirable to ensure their function, reliability and performance. When modifications are applied to a tool, Non-Regression Test (NRT) asserts that the changes have intended effect while Regression Test (RT) verifies that the tool still performs correctly without unwanted behaviour.

The work presented in this paper is focused on a method to automatically perform Non-Regression Test in HLS tool developments, although it can also be used as a Regression Testing technique. This method relies on a framework which allows HLS tool developers to verify the circuits generated from the tool directly on FPGA, instead of using simulations. The verification flow is automatic, so that knowing the details of the system is unnecessary for developers. The framework has been tested successfully over several applications from HLS benchmark and it gives more promising results than its simulation counterpart.

CCS Concepts

•**Hardware** → **Reconfigurable logic applications; Simulation and emulation;** *Best practices for EDA;*

Keywords

High-Level Synthesis; Non-Regression Test; FPGA

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) used to be reserved for digital hardware engineers in ASICs prototyping. The goals were centered on design optimization as much as possible. Given the usual very long design time in that domain, it was then appropriate for High-Level Synthesis (HLS) tools to take a long time (hours to days) exploring solutions, running simulations and even repeatedly launching back-end synthesis operations.

Nowadays, FPGAs are significantly bigger and more powerful than their earliest version and a huge interest comes from software programmers for general-purpose hardware acceleration. Their main objective is to rapidly obtain a significant acceleration from software applications. However, the considered applications are often big and very complex from Design Space Exploration (DSE) point of view, which make the search for the most optimized solutions less practical.

Due to these reasons, current HLS tools and DSE methods that suit digital hardware engineers are not appropriate for software engineers. Automatic and fast HLS tools are desirable to generate promising design suitable for software programmers. In the development of such tools, tests are being performed to ensure their function, reliability, and performance. These tests should be able to easily verify the development without additional complexity and works for developers.

When modifications are applied in an HLS tool being developed, Non-Regression Test (NRT) is normally carried out to verify whether the changes have intended effect. It can be performed by verifying the circuits generated by the tool. So far, behavioral simulation of the generated circuits is the simplest approach in NRT of HLS tools due to the rapidly obtained result, but the physical parameters are not considered in this type of simulation, e.g. gate and interconnection delays. On the other hand, simulating the circuits with additional timing annotations often takes unreasonably long time, especially when the designs are big and complex. Verification of the circuits on actual FPGA can be interesting since the physical parameters are naturally involved and the execution time is much shorter than its simulation counterparts but it requires huge efforts in design implementation.

The contribution of this work is focused on a method to perform NRT of HLS tools by directly verifying the generated circuits on FPGA. In [6], it is explained how automating NRT in software development can decrease the efforts of developers. We developed an automatic and transparent framework so that even HLS tool developers with a weak background in hardware design are able to access the framework. Without knowing the details of the system, developers are able to perform NRT of the HLS tool they are working on. In this work, we took HLS tool AUGH [8] as an example case of HLS tool under development although our proposed framework

is generic and applicable for any HLS tools.

This paper is organized as follows: Section 2 describes the HLS tool AUGH which is the main part of our specific case. Section 3 presents the architecture overview and the design flow. Section 4 shows the implementation and the results. Finally, in Section 5, we state the conclusion and the potential future improvement of this work.

2. THE HLS TOOL AUGH

AUGH [8] is a free and open-source HLS tool for FPGA devices and boards. It is designed to be accessible to people with a weak background in hardware design, e.g. to software programmers.

For that purpose, AUGH takes as input ANSI-C programs that represent the algorithms to transform into FPGA hardware accelerators. All relevant specificities of the target device or FPGA board are automatically handled, including the amount of available FPGA primitives (in LUTs, FFs, RAM blocks, DSP blocks) and the target clock frequency. It autonomously performs fast Design Space Exploration (DSE) and generates optimized circuits under the strictly given hardware constraints (resource and clock frequency).

2.1 Handling of FPGA technologies

AUGH exploits an internal library of calibrated component models to transparently handle hardware target constraints such as resource and clock frequency. Each component model (e.g. adder, register, multiplexer, etc) is associated with a set of dedicated functions that calculates the component's resource usage (in LUTs, FFs, RAM blocks, DSP blocks) and delays (in sequential and combinational logic). Using these features, AUGH is able to precisely estimate the size of the entire circuit and its maximum clock frequency.

2.2 Estimation of the circuit execution time

The search for an efficient hardware implementation usually involves generating several implementation versions and retaining the fastest one. Due to this reason, it is needed to precisely know the execution time of each version. When the input algorithm contains only statically-known control, such as loops with literal bounds, the execution time is known perfectly.

However, when there is any control dependency, the input algorithm alone is often not enough to estimate the execution time. AUGH can cope with that situation by handling dedicated user annotations inserted in the code. These annotations, entirely optional, indicate the relevant branch probabilities and an average number of iterations of most relevant loops. More details can be found in [8].

This property enables AUGH to very rapidly estimate the execution time of any hardware implementation solution. In particular, there is no need to launch RTL simulation of all solutions, which is a very significant time saving. This enables AUGH to perform Design Space Exploration (DSE) and to rapidly obtain an optimized solution.

2.3 Generation flow

The input C code given to AUGH by the user normally represents a program, which means it is supposed to be executed instruction by instruction. This is what is expected when targeting a microprocessor, but on FPGA this would lead to a very slow circuit.

In order to obtain a rapid circuit, AUGH has to detect parallelism opportunities between the instructions of the C code, or create parallelism by applying modifications to the given instructions. To that purpose, AUGH is equipped with modules dedicated to detecting appropriate transformations and applying them to its internal representation of the circuit.

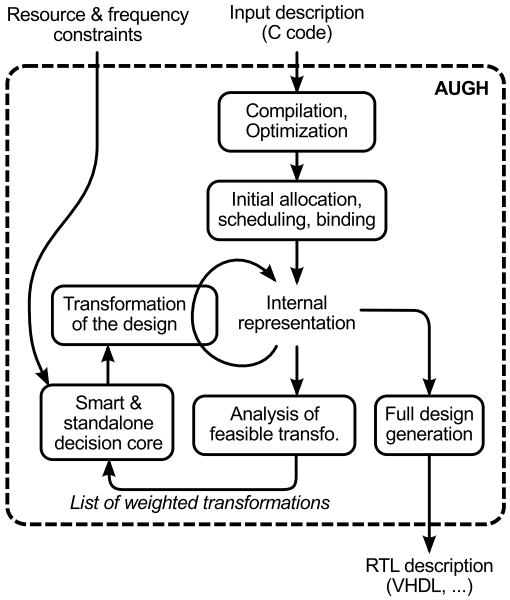

Figure 1: The AUGH generation flow

Figure 1 illustrates the AUGH generation flow. The input C code is initially loaded to AUGH and a first version of the circuit is generated by applying the usual operations allocation, scheduling and binding, which are known to be well suited to answer resource constraints [4]. And then, AUGH will execute DSE iteration with the initial solution as the starting point until the optimized solution is found. The detail will be explained in the next section.

2.4 Design space exploration

Searching for the theoretically optimal solution under resource constraints is known to be an unreasonably complex and long exploration task. For this reason, most HLS tools do not perform DSE. Instead, the user must indicate, through special directives, how to transform the input design (e.g. unrolling loops, adding computing operators). This is the case for the tools Catapult, PICO, Cynthesizer (now Cadence), GAUT, Xilinx Vivado HLS, ROCCC, and LegUp which are presented in [4] [9].

Some automatic DSE techniques have been proposed. In CyberWorkBench [4], genetic algorithms are used. However, the flow requires user expertise in order to intuitively initialize the freedom degrees of the design. In Altera OpenCL [9], the tool follows the OpenCL standard, where a computation *kernel* is synthesized and then duplicated as many times as possible. This is well suited for hardware acceleration on a PC with a PCIExpress-connected FPGA board but is not appropriate for embedded systems.

Generally, software programmers are not looking for the optimal solution. Instead, they would prefer a significant acceleration of their applications compared to a software version. Also, similarly to a compilation flow, they want the circuits to be generated automatically and in a reasonable time. The DSE algorithm implemented in AUGH is specifically designed to fit these goals.

Figure 2 illustrates a typical progression of the AUGH DSE algorithm. It is a greedy algorithm: from the initial solution, it iteratively applies transformations that make the circuit faster. This process generally increases the circuit size.

The possible transformation types are the addition of operators (adders, multipliers, etc), the addition of read ports to memory banks, replacement of memory banks by separate registers, unrolling of loops and wiring of conditions. Each transformation type is associ-

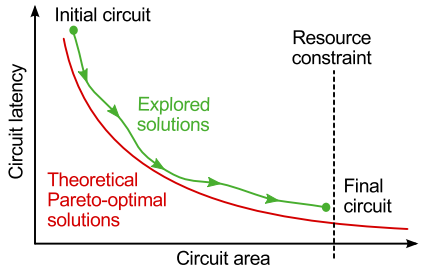

Figure 2: AUGH DSE progression

ated with a set of functions that provide estimations of design speed up and usage of additional hardware resources.

As illustrated in Figure 1, at each DSE iteration, AUGH lists all feasible transformations. Each one is weighted in resource cost and design speedup with the estimators. Then one or several transformations are selected among the ones that generate the lowest circuit latency while having the smallest impact on its size (circuit area) and applied to the circuit. The DSE is stopped when no feasible transformation remains or when all transformations would make the circuit exceed the resource constraint. Detailed examples of DSE on actual benchmark applications can be found in [8].

Due to its DSE process, AUGH rapidly generates optimized circuits for the target resources and clock frequency constraints. Within seconds to minutes, software programmers are able to obtain VHDL implementations of FPGA accelerators, with the corresponding estimations of their resource usage and execution time.

2.5 Interest for on-board Non-Regression Testing

Like many other HLS tools under development, tests are desirable to ensure that the latest modifications of AUGH are correct. Besides unit tests and integration tests, AUGH undergoes NRT as a method to make sure that the additional functions work properly. One method to perform NRT in HLS tools is by verification of the generated circuits.

The basic strategy in circuit verification usually includes behavioral simulation which does not consider physical parameters of targeted device, e.g. gate delay, interconnection delay, etc. Although this simulation is considered enough to check the functionality of circuits, it is not reliable enough for NRT of HLS tools for the reason of insufficient parameters. Most digital hardware designers will perform post-implementation (after placement and routing) simulation with additional timing annotations in circuit verification, although it is extremely time-consuming especially for big and complex circuits.

Due to the conditions stated above, on-board NRT offers more interesting advantages compared to simulation-based NRT. The verification is considered reliable since it is performed on actual FPGA. Executing circuits on real FPGAs will be much less time-consuming. The major drawback of on-board execution is the long placement and routing process, but this is also the case for post-implementation simulation. Currently, HLS tool developers are not interested in doing NRT directly on FPGA due to the necessary efforts in system design.

An automated framework that uses the Zynq System-on-Chip (SoC) [5] to perform remotely reconfigurable platform has been proposed in [7]. It is able to program the FPGA part and run applications with a pre-built software environment. However, the generation of FPGA configuration files is not included in the design flow. The user has to generate them on its own, including synthesizing the designs and adding the appropriate communication interface wrappers,

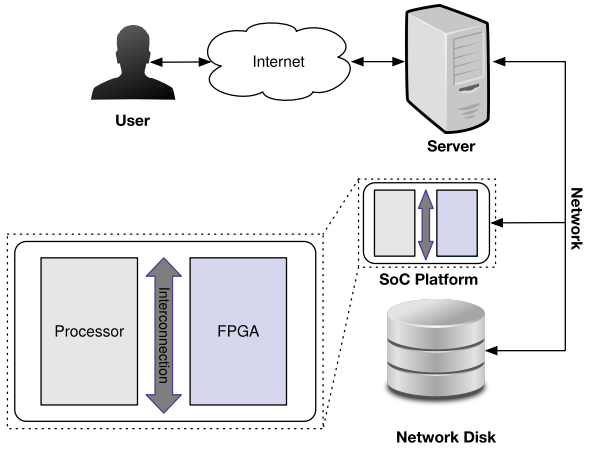

Figure 3: System Overview

which is highly inconvenient for a user with a weak background in hardware designs.

The tool LegUp [3] generates designs where the hardware accelerators are tightly coupled to a soft processor. This alleviates the user's work, but the processor part is much less powerful than hard processors integrated into SoC platform, such as Cortex-A9 ARM cores, while consuming an important amount of FPGA resources.

In the following section, a novel and automated framework is described. It is completely autonomous, starting from the HLS flow with AUGH until the results of NRT from the execution on a SoC platform.

3. ARCHITECTURE OVERVIEW AND SYSTEM DESCRIPTION

This section describes the architecture of our proposed framework for on-board NRT in HLS tool developments. The framework relies on a SoC platform which enables efficient software-controlled environment. The proposed design flow to perform autonomous and transparent NRT is also presented in this section.

3.1 Specification

Our main objective is to provide a framework for on-board verification of circuits generated by HLS tools under development, which is AUGH in our case. We designed a framework which receives as input the VHDL description of the circuits, automatically advances the process to synthesis, placement, and routing, and generates the configuration files for targeted FPGA. It will then reconfigure the FPGA and run the execution autonomously. Note that test vectors are prepared in a binary format for circuit verification purpose.

3.2 System Overview

The core of our framework is a programmable SoC platform which integrates an FPGA and a hard-processor in the same device, e.g. Xilinx Zynq [5] and Altera SoC [1]. This type of platforms allows very efficient control of the FPGA from the processor since they are able to communicate internally using AXI interconnect.

In our proposed framework, the SoC platform is connected to a host server for the generation of configuration files (bitstreams) and a network storage to store the bitstreams as well as the test vectors and golden reference. Knowing the details of the system is unnecessary for HLS tool developers who want to perform NRT using this framework. Figure 3 illustrates the overview of the proposed system.

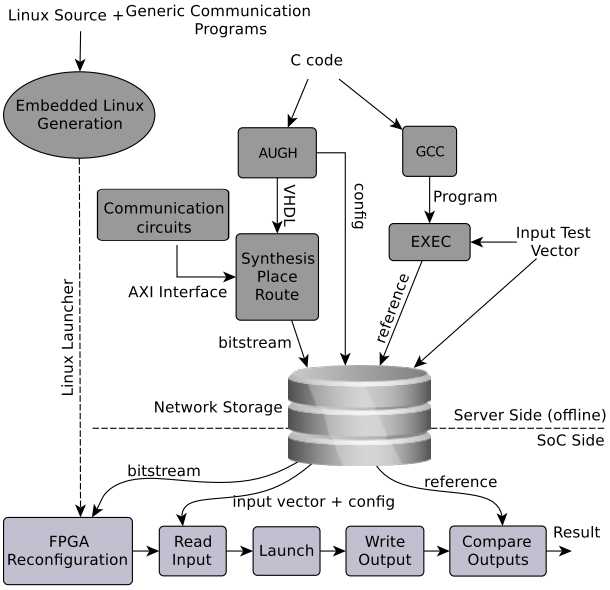

Figure 4: Design Flow of The System

The HLS tool to be tested is installed on the host server. Meanwhile, any back-end synthesis tools related to the SoC platform used must also be available on the server as well for bitstream generation. A common network protocol is used so that the network storage is accessible to both the host server and the SoC platform.

3.3 Proposed Design Flow

Our design flow consists of two main steps. The first step is the generation of the bitstream for targeted FPGA on the server. The second step is the circuit verification on SoC platform. Both steps are illustrated in Figure 4.

On the server, the flow starts from C code written by the user. Using AUGH, VHDL representation of the circuits is generated. The process continues to back-end synthesis, placement and routing, and bitstream generation for targeted FPGA. We prepared *Communication Circuits* (ref. Figure 4) with AXI interfaces in advance which will be automatically integrated into the design to facilitate the communication between FPGA and processor using AXI interconnect. Meanwhile, the same C code is compiled with GCC. The objective of this process is to obtain a golden reference that is necessary in the circuit verification.

In the end of the flow on the server, the bitstream, the input test vectors, and the golden reference are stored in *Network Storage* together with a file which contains the information of I/O port widths (*config* in Figure 4).

In our work, we used embedded Linux system in the processor of the SoC platform. The main reason is because it integrates FPGA reconfiguration driver as native, remote access SSH protocol and other packages in Linux distribution which can be configured according to our needs.

On the SoC platform, the flow is executed under Linux environment. It searches all the files related to the verification in Network Storage : (1) bitstream, (2) input test vectors and *config*, and (3) golden reference. The outputs of the circuits will be compared to the golden reference and the comparison results are the results of NRT.

4. IMPLEMENTATION AND RESULTS

We used ZedBoard Zynq-7000 ARM/FPGA SoC as the main platform in our implementation. The ZedBoard integrates a Xilinx

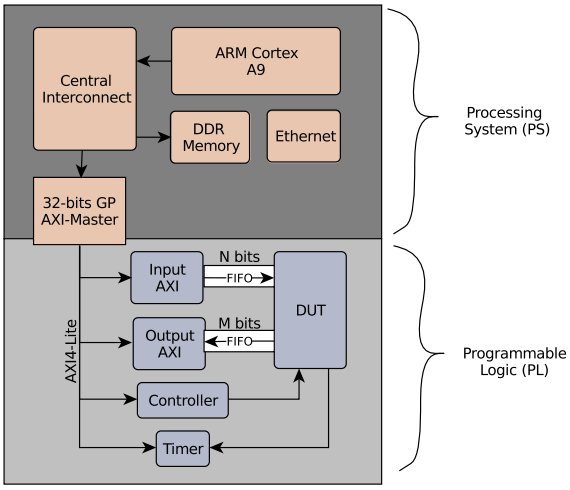

Figure 5: Hardware Architecture Layout

SoC XC7Z020, 512 MB DDR3, and other peripheral devices. In our framework, the ZedBoard is connected to a server and a network storage via ethernet connection.

The server used is NUMA-based with 4 nodes and 8 cores Intel Xeon CPU @ 1.87GHz in each node. It is equipped with 32GB RAM and 18 MB L3 Cache. The server is running on Debian 4.3. Bitstream generation is performed using Xilinx Vivado Design Suite 2015.3.

4.1 Hardware Architecture

The Xilinx SoC XC7Z020 consists of a dual-core ARM Cortex-A9 based Processing System (PS) and a Xilinx Programmable Logic (PL) in a single device. The working frequency of the circuits in PL is adjustable following the constraints set in HLS tool AUGH. In our experiments, 100 MHz is arbitrarily chosen. The PS and PL communicate via AXI interconnect. This work used AXI low performance (AXI-lite) interconnect for early implementation.

The AXI-lite interconnect uses master/slave communication model. In the framework, the processor in PS is the master for software-controlled execution. The circuit we want to verify (Design Under Test, DUT) is programmed in PL as the slave. The communication between DUT and processor is done via *Communication Circuits* which integrate AXI interfaces.

Figure 5 shows the hardware architecture layout in Xilinx SoC XC7Z020.

The *Communication Circuits* which hold the I/O data are *Input AXI* and *Output AXI*. The DUT is linked to *Input AXI* and *Output AXI* through FIFOs. These FIFOs are used to ease the communication protocol conversion between AXI interconnect and DUT since they are different. The DUT uses handshake protocol while AXI interconnect implements AXI protocol. While performing the conversion between the protocols, the data should be stored in the FIFOs. In our experiments, the DUT only has single I/O port. In case of DUT with multiple I/O ports, the number of *Input AXI* and *Output AXI* should follow the number of I/O ports.

Besides the protocol conversion between DUT and AXI interconnect, the data width often has to be adjusted. AXI lite interconnect has 32-bit data width while the data width in DUT depends on its application, shown by N and M in Figure 5. Our framework automatically adjusts the I/O port widths of *Input AXI* and *Output AXI* before the synthesis in Vivado.

During communication between the processor and the DUT, the FIFOs in *Input AXI* and *Output AXI* store the data and serialize/dese-

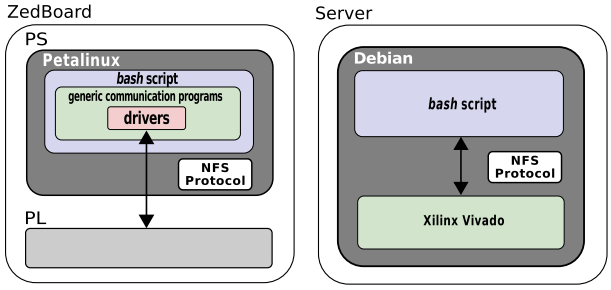

Figure 6: Software Architecture Layout

rialize the input/output according to the destination. In *Input AXI*, the FIFO holds the 32-bit data from AXI interconnect and transforms it into inputs for DUT while in *Output AXI*, it transforms outputs from DUT to the data which will be sent to AXI interconnect. For example, if the input of DUT is the 8-bit width, each word of 32-bit from AXI interface in *Input AXI* is converted to 4 inputs which are released sequentially by *Input AXI*. A similar method is used in *Output AXI*.

These customized interfaces are designed to consecutively send data each clock cycle to guarantee that the I/O communication will not create bottleneck even though the data width of the circuit is different.

The *Controller* is designed to receive commands from the processor and translate them into control signals to the DUT.

For experiment purpose, the *Timer* counts the execution time (in clock cycles) and saves it. It will be sent to the processor after execution is finished.

Note that *Input AXI*, *Output AXI*, *Controller*, and *Timer* were generated using pre-built interfaces in Vivado. They are customized to allow the communication between DUT and processor and are added to the design before synthesis. In terms of FPGA resources, the four *Communication Circuits* consume 386 LUT slices in logic and 798 FF registers. The FIFOs are implemented in BRAM and they consume one BRAM for each 1K x 32-bit FIFO.

4.2 Software Architecture

The software architecture of our implementation relies on the programs launched on the server and ZedBoard. On the server, a script performs the bitstream generation using TCL commands in Vivado for non-graphical execution. The generated bitstreams are stored in Network Storage using Network File System (NFS) protocol.

On the ZedBoard, software is executed by the ARM processor. We developed generic communication programs to find the bitstreams in Network Storage and access the drivers for AXI interfaces. The verification process on FPGA is launched by a script. Petalinux is chosen as the embedded operating system on the ZedBoard.

Figure 6 illustrates the software architecture layout in our implementation.

4.2.1 Petalinux

In this work, we run Petalinux version 2014.4 on the ARM. It is a Linux distribution built and deployed for Xilinx SoC. This approach gives the advantage in FPGA reconfiguration as well as in communication via AXI interconnect since all the drivers are natively installed in Petalinux.

Every time the ZedBoard is turned on, it boots the Petalinux from pre-installed SD card. This SD card contains the Petalinux bootloader and kernel. The kernel was generated one time and does not have to be included in the design flow. The Petalinux bootloader and kernel generation is the only manual step we did in this work.

4.2.2 Generic Communication Programs

This section describes the generic communication programs integrated into Petalinux. These programs are developed to access the drivers in Petalinux in an abstract way since we would like to control the execution using custom scripts. The programs are integrated into Petalinux during kernel generation. They are written in C code, generic for all applications tested, and executable from the command line.

- *com_axi*: This function is a modified version of ready-to-use function from Xilinx tutorial to write and read an integer from Linux shell to the AXI4-Lite interface. It sends 32-bit values to *Controller*, or reads number of cycles from *Timer*.

- *write_vectors*: This function reads input test vectors from the Network Storage and writes them to *Input AXI*.

- *read_vectors*: This function reads from *Output AXI* and writes in a file and send it to Network Storage.

- *compare*: This function is used to compare the output with golden reference and list the error parts of output if there is any.

4.2.3 Scripts

Automated NRT is performed by two scripts written in *bash*. The first script, *script_server*, is launched on the server to generate the configuration files. It manages the flow starting from the moment the input C code is received by AUGH until the bitstream is generated on the server. The second script, *script_zed* manages the verification on the ZedBoard by executing the *Generic Communication Programs* explained in 4.2.2. Both scripts represent the execution flows explained in 3.3. Note that the executions of both scripts are independent of each other.

- *script_server*: After circuit generation from HLS tool AUGH, the next processes are executed by *script_server*. The script reads all necessary parameters to adjust the communication interfaces and launches Vivado for bitstream generation in *batch* mode. The process advances to synthesis, place-and-route and bitstream generation in Vivado using Tool Command Language (TCL) commands. The generated bitstream is put in Network Storage together with the input test vectors, golden reference, and *config*.

- *script_zed*: The *script_zed* searches the bitstream file from the Network Storage. It launches the reconfiguration of the FPGA with the bitstream and starts the execution on FPGA. From Petalinux, the script access the *devcfg* driver to reconfigure PL via Processor Configuration Access Port (PCAP). The verification is launched with input test vector given and the output is compared with the golden reference. All this process is done sequentially in *script_zed* and can be repeated with different sets of inputs (bitstream, input test vector, golden reference, and *config*).

Using the proposed framework, we are able to carry out Non-Regression Tests of HLS tool AUGH. The result of comparison between the output of the circuits and the golden reference on the ZedBoard shows whether any regression exists after modifications of AUGH.

4.3 Network File System

During NRT, file exchange is done between server, ZedBoard, and Network Storage. It is possible if a common protocol is used in communication among them. After considering the complexity level and performance, we decided to use Network File System (NFS) in our proposed framework.

The implementation of NFS requires two packages: *nfs-kernel-server* package in server side and *nfs-common* in client side. Meanwhile, in Petalinux, the external package of NFS client is not necessary since it has been integrated into its kernel. Enabling *portmap* in the kernel configuration will allow NFS disk mounting.

4.4 Examples of on-board NRT in AUGH development

This section describes some examples of the on-board NRT we have done using our proposed framework. In [8], several applications are taken from CHStone benchmark suite to test HLS tool AUGH. The same applications are used to confirm the reliability of our system as an NRT framework.

Table 1 details the time spent in Non-Regression Test of HLS too AUGH. The first column lists all the applications used in the test. The second column presents the time needed by AUGH to generate the VHDL description of corresponding applications, which was relatively short except for *mjpeg* which took some time in DSE due to its size. The last two columns present the time spent by our framework to perform NRT on the ZedBoard. The *Communication Circuits* and DUT are integrated into the design and compiled using Xilinx Vivado on the server. The time for synthesis, placement and routing, and bitstream generation is significantly higher compared to the execution time on FPGA.

For reference, the time spent when NRT is performed in a simulation is shown in the third, fourth, and fifth columns of the table. The third column presents the time spent in the behavioral simulation. It checks the functionality of DUT without considering the circuit latency. While behavioral simulation time is relatively short, it is insufficient in NRT since we would like also to verify the clock frequency estimation of the circuits from HLS tools.

The other type of simulation is the post-implementation simulation (fourth and fifth column of Table 1). This simulation is performed after synthesis, placement, and routing process (with FPGA technology consideration). Using the netlist generated for targeted FPGA, the simulation can be launched to check the functionality of the circuits with or without additional timing annotations. Functional simulation checks the behavior of the DUT without calculating the delay while timing simulation includes netlist and logic gate delays from the timing annotations. The time spent in post-implementation timing simulation can be significant depending on the size and complexity of the circuits.

With the data from Table 1, we are able to compare the NRT time using our proposed framework and simulations. The additional circuits in our hardware architecture introduce an overhead to the design, thus extend the bitstream generation time. However, for bigger and more complex applications, the overhead becomes less significant. Performing post-implementation simulation of a complex application may take more time than executing the application directly on FPGA. Some applications take too much time in our experiments and become irrelevant to our comparison, e.g. post-implementation timing simulation of *mjpeg* and *adpcm* shown in Table 1. Although the long bitstream generation in our framework can be a major drawback, it is actually still considered reasonable if we compare to post-implementation simulation time. Moreover, our framework gives the precise time of the on-board circuit execution.

With enough number of stimulus, our proposed framework can be used to validate any circuits generated by AUGH at a given working frequency. Depending on the application, a large number of stimulus might be needed to find the bug in the circuits. For circuit validation with large amounts of test vectors, on-board execution offers the same results but in a shorter time than simulation.

Table 2: Optimal working frequency of DUTs on ZedBoard

	Optimal Frequency (MHz)	# of iterations
idct	180	5
mjpeg	250	11
adpcm	250	11
aes	350	21
blowfish	400	26
gsm	350	21
motion	120	2

The fact that the NRT is performed on actual FPGA allows working frequency verification of the designs. With the given estimated frequency, the circuits generated by HLS tools can be verified on-board with more trusted results than simulation. In some cases, a circuit does not work properly at its desired working frequency. In other cases, it can achieve higher working frequency than targeted. Knowing optimal working frequency of a circuit generated by HLS tool helps developers discovering its full run-time potential. In [2], adjusting clock frequency to find its full potential was able to improve the performance in execution.

In further implementation, we added a register-controlled PLL to the framework to adjust the working frequency of DUT at runtime. It is called *Clocking Wizard* in Vivado and it can be controlled via AXI interface. This IP allows changing the frequency of the circuits on the fly without regenerating the bitstream. By iteratively increasing the frequency by 20 MHz and 10 MHz after the circuits achieved 200 MHz, we were able to find the maximum frequency at which the DUT still worked properly within few seconds. Table 2 presents the optimal frequency obtained for each circuit from Table 1 using our proposed framework and the number of iterations performed to get the results.

4.5 Adaptation of the framework to other platforms

Our method is not limited to the platform used in our implementation. The genericity of the proposed framework allows performing NRT of other HLS tools.

From the hardware point of view, the architecture is layout is defined in our proposed framework. The strategy is applicable to other SoC platforms from Xilinx Zynq or from other vendors with ARM processor and FPGA connected with AXI interconnect. The design is built in VHDL language which is technology-independent, although certain modifications are expected for different FPGA vendors with different tools.

From the software point of view, the programs to access the drivers are reusable in other SoC platforms from Xilinx Zynq since they implement the same drivers. In SoC platforms from different vendors, modifications of the programs are necessary to adapt the drivers used. The scripts should be adapted for back-end synthesis tool from other vendors.

5. CONCLUSION

In this paper, we presented a method to perform automated NRT in HLS tool developments directly on FPGA. Our method is well-suited

Table 1: Non-Regression Test Time (in seconds)

	AUGH Execution	DUT Only			DUT + Communication circuits	
		Behavioral Simulation	Post-Implementation Simulation		Synthesis, P&R and Bitstream Generation	Execution on FPGA
			Functional	Timing		
idct	1.29	55.60	449.88	889.32	738.46	0.73
mjpeg	31.58	137.31	1346.58	n/a[1]	864.28	0.79
adpcm	0.57	63.79	489.71	n/a[1]	783.87	0.75
aes	0.50	52.75	288.36	543.14	456.65	0.84
blowfish	0.52	67.19	504.09	2219.09	518.30	0.77
gsm	0.46	53.00	321.58	482.86	517.03	0.76
motion	0.18	46.65	266.93	1143.81	464.49	0.74

[1] The simulation did not successfully finish the elaboration process. The process took too much time and was stopped after 24 hours.

for developers with a weak background in hardware design since the flow is transparent and automatic. Even with relatively long placement and routing time as its major drawback, our proposed framework offers more promising results compared to simulation-based NRT, especially for post-implementation timing simulation. The proof-of-concept of our system relies on Zynq SoC platform which allows to automatically verify the circuits generated by AUGH. With relatively small modification, the method is applicable to other SoC platforms with similar specifications.

Mainly constructed to perform on-board NRT of HLS tools, our proposed framework can also be used to rapidly validate the maximum working frequency of the circuits generated by HLS tools. With sufficient test vectors, performing circuit validation is also possible using the proposed framework.

Acknowledgments

The authors would like to thank Cyril Brisebard, Jérémy Brun and Julien Flèche for their contributions in the project, both in terms of concept and development.

6. REFERENCES

[1] Altera SoCs: When Architecture Matters.

[2] J. A. Bower, W. Luk, O. Mencer, M. J. Flynn, and M. Morf. Dynamic clock-frequencies for FPGAs. *Microprocessors and Microsystems*, 30(6):388–397, Sept. 2006.

[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 the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 33–36. ACM, 2011.

[4] P. Coussy and A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 2008.

[5] L. Crockett, R. Elliot, M. Enderwitz, and B. Stewart. *The Zynq Book*. Strathclyde Academic Media, july 2014.

[6] D. Hoffman. Non-Regression Test Automation. In *PNSQC*, Portland, Oregon, 2008.

[7] O. Machidon, F. Sandu, C. Zaharia, P. Cotfas, and D. Cotfas. Remote SoC/FPGA platform configuration for cloud applications. In *Optimization of Electrical and Electronic Equipment (OPTIM), 2014 International Conference on*, pages 827–832, May 2014.

[8] A. Prost-Boucle, O. Muller, and F. Rousseau. Fast and standalone design space exploration for high-level synthesis under resource constraints. *Journal of Systems Architecture*, 60(1):79 – 93, 2014.

[9] S. Windh, X. Ma, R. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, March 2015.