Translating Timing into an Architecture: the Synergy of COTSon and HLS

(Domain Expertise: Designing a Computer Architecture via HLS)

Roberto Giorgi[1], Farnam Khalili[1,2], and Marco Procaccini[1]

[1]Department of Information Engineering and Mathematics, University of Siena, Italy.
[2]Department of Information Engineering, University of Florence, Italy

(Correspondence should be addressed to Roberto Giorgi; giorgi@diism.unisi.it)

Abstract

Translating a system requirement into a low-level representation (e.g., register transfer level or RTL) is the typical goal of the design of FPGA based systems. However, the Design Space Exploration (DSE) needed to identify the final architecture may be time consuming, even when using High Level Synthesis (HLS) tools.

In this paper, we illustrate our hybrid methodology, which uses a frontend to HLS so that the DSE is performed more rapidly by using a higher-level abstraction, but without loosing accuracy, thanks to the HP-Labs COTSon simulation infrastructure in combination with our DSE tools (MYDSE tools). In particular, this proposed methodology proved useful to achieve appropriate design of a whole system in a shorter time than trying to design everything directly in HLS.

Our motivating problem was to deploy a novel execution model called Data-Flow Threads (DF-Threads) running on yet to be designed hardware. For that goal, directly using the HLS was too premature in the design cycle. Therefore, a key point of our methodology consists in defining the first prototype in our simulation framework and gradually migrating the design into the Xilinx HLS after validating the key performance metrics of our novel system in the simulator.
To explain this workflow, we first use a simple driving example consisting in the modelling of a two-way associative cache. Then, we explain how we generalized this methodology and describe the types of results that we were able to analyze in the AXIOM project, that helped us reduce the development time from months/weeks to days/hours.

1. Introduction

In recent decades, applications are becoming more and more sophisticated and that trend may continue in the future [1]–[3]. To cope with the consequent system design complexity and offer better performance, the design community has moved towards design tools that are more powerful. Today many designs rely on FPGAs [4], [5] in order to achieve higher throughput and better energy efficiency, since they offer spatial parallelism on the portion of application characterized by dataflow concurrent execution. FPGAs are becoming more capable to integrate quite large designs and can implement digital algorithms or other architectures such as soft-processors or specific accelerators [5]. For the efficient use of FPGAs, it is essential to have an appropriate toolchain. The toolchain provides an environment, in which the user can define, optimize and modify the components of the design, by taking into account the power, performance, and cost requirements of a particular system and eventually synthesize and configure the FPGA.

The conventional method to implement application code on FPGAs is to write the code in Hardware Description Language (HDL) (e.g., VHDL or Verilog). Although working with HDL languages still is the most reliable and detailed way of designing the underlying hardware for accelerators, their use requires advanced expertise in hardware design as well as remarkable time. The Design Space Exploration (DSE) and debugging time of FPGAs and the bitstream generation may reach many hours or days even with powerful workstations. As such, moving an already-validated architecture to the FPGA's tool-flow may save significant time and effort and, as a result, facilitates the design development.

This situation is exacerbated by the interaction with the Operating Systems and by the presence of multicore. Therefore, the use of full-system simulators in combination with HLS tools permits a more structured design flow. In such case, a simulator can preliminary validate an architecture and the HLS-to-RTL time is repeated less times.

There are parameters, which make simulators preferable to reach a certain level of performance, scalability, and accuracy as well as reproducibility and observability. Based on the experience of the previous projects like TERAFLUX [6], [7], ERA [8], [9], AXIOM [4], [10]–[13], SARC [14], [15] we choose to rely on the HP-Labs COTSon simulation infrastructure [16]. The key feature of COTSon that is useful in HLS design is its "functional-directed" approach, which separates the functional simulation from the timing one. We can define custom timing models for any component of an architecture (e.g., FPGA, CPU, Caches) and validate them through the functional execution: however, the actual architecture has to be specified by a separate "timing model" (see Section 2 for more details). The latter is what can be migrated in a straightforward way to HLS. Moreover, COTSon is a full-system simulator, hence it permits to study the OS impact on the execution and choose the best OS configuration based on the application requirements [17]; the OS modelling is sometimes not available in other tools (reviewed in Section 2).

In this paper, we illustrate the importance of the simulation in synergistic combination with the Xilinx HLS tool [18], in order to permit a faster design environment, while providing a full-system Design Space Exploration (DSE).

Additionally, thanks to our DSE toolset (MYDSE) [17][19], we facilitate the extraction of important metrics like the execution time but also more detailed ones like cache miss rates, bus traffics and so on, which help investigate the appropriate system design. In order to illustrate our methodology, we start from a driving example related to design a simple two-way associative cache system. The methodology is then generalized by considering the case of the AXIOM project, in which this methodology was actually used to design and implement a novel Dataflow architecture [20]–[22] through the development of our custom AXIOM-Board [11].

The contributions of this work are:

- Presenting our methodology for designing FPGA-based architectures, which consists in the direct mapping of COTSon "timing models" into HLS, where such models are pre-verified via our MYDSE tools: the DSE is performed before using the HLS tools, thus saving much design time;
- Illustrating a simple driving example based on the modelling and synthesis of a simple two-way set-associative cache in order to grasp the details of our methodology;
- Presenting the bigger picture of using our proposed methodology to design a whole software/hardware platform (called AXIOM).

The rest of the paper is organized as follows: in Section 2, we analyze related work; in Section 3, we illustrate our methodology and tools; in Section 4, we provide a simple driving case-study; in Section 5, we show the possibilities of our tools in the more general context of the AXIOM project.

2. Related work

Our design and evaluation methodology aims at integrating simulation tools and HLS tools to ease the hardware acceleration of applications, via custom programmable logic. HLS tools improve design productivity as they may provide a high level of abstraction for developing high-performance computing systems. Most typically, these tools allow users to generate a RTL representation of a specific algorithm usually written in C/C++ or SystemC. Several options and features are included in these tools in order to provide an environment with a set of directives and optimizations that help the designer meet the overall requirements. In our case, we realized that more design productivity could be achieved by identifying in the early stages a candidate architecture through the use of a simulator: however, *the use of a generic simulator may not help identify the architecture, since often the simulation model is too distant from the actual architecture or is too much intertwined with the modeling tool* [23][24][25][26]. On the other hand, the COTSon simulator is using a different approach, called "functional directed" simulation, in which the functional and timing models are neatly separated and the first one drives the latter[1]. The similarity of our "timing model" specification to an actual architecture is an important feature and it is the basis for our mapping to a HLS specifcation.

In our research, we used Xilinx Vivado HLS, but other important HLS frameworks are available and are briefly illustrated in the following; their main features are summarized in Table I. LegUp [27] supports C/C++, Pthreads and OpenMP as programming models for HLS [28] by leveraging the LLVM compiler framework [29], and permits parallel software threads to run onto parallel hardware units. LegUp can generate customized heterogeneous architectures based on the MIPS soft processor. Bambu [30] is a modular open-source HLS tool, which aims at the design of complex heterogeneous platforms with a focus on several trade-offs (like latency versus resource utilization) as well as partitioning on either hardware or software. GAUT [31] is devoted to real-time digital signal processing (DSP) applications. It uses SystemC for automatic generation of testbenches for more convenient prototyping and design space exploration. DWARV [32] supports a wide range of applications like DSP, multimedia, encryption. The compiler used in DWARV is the CoSy commercial infrastructure [33], which provides a robust and modular foundation extensible to new optimization directives. Stratus HLS of Cadence [34] is a powerful commercial tool accepting C/C++ and SystemC and targeting a variety of platforms including FPGAs, ASICs, and SoCs. Thanks to low power optimization directives, the user can achieve a consistent power reduction. It gives support for both control flow and data flow designs, and actively applies constraints to trade-off speed, area, and power consumption. Intel HLS Compiler [35] accepts ANSI C/C++ and generates RTL for Intel FPGAs, which is integrated into the Intel Quartus Prime Design Software. Xilinx Vivado HLS tool targets Xilinx FPGAs [18], which offers a subset of optimization techniques including loop unrolling, pipelining, dataflow, data packing, function inline, bit-width reduction for improving the performance and the resource utilization.

[1] It is important to note that the "timing model" implicitly defines an architecture, which is functionally equivalent to the "functional model", but it is a totally separated code with different simulation speeds [16].

Xilinx SDSoC is a comprehensive automated development environment for accelerating embedded applications [36]. The tool can generate both RTL level and the software running on SoC cores for the "bare-metal" libraries, Linux, FreeRTOS. Xilinx SDAccel [37] aims at accelerating functionalities in data-centers through FPGA resources. We summarize the key features of aforementioned HLS tools in Table 1.

Table 1 – Key features of discussed HLS tools. For the non-obvious columns, *Testbench* means the capability of automatic testbench generation. *SW/HW* means the support for the Software/Hardware co-design environment. *Floating Point* and *Fixed Point* are the supported data types for the arithmetic operations.

Tool	*Owner*	*License*	*Input*	*Output*	*Domain*	*Testbench*	*SW/HW*	*Simulation*	*Floating Point*	*Fixed Point*
LegUp [27]	LegUp Computing	Commercial	C, C++	Verilog	All	Yes	Yes	HW	Yes	No
Bambu [30]	Politecnico di Milano	Academic	C	VHDL, Verilog	All	Yes	Yes	SW, HW	Yes	No
GAUT [31]	U. Bretagne Sud	Academic	C, C++	VHDL, SystemC	DSP	Yes	No	HW	No	Yes
DWARV [32]	TU Delft	Academic	C	VHDL	All	Yes	Yes	HW	Yes	Yes
Stratus HLS [34]	Cadence	Commercial	C, C++, SystemC	C, C++, SystemC	All	Yes	Yes	SW, HW	Yes	Yes
Intel HLS Compiler [35]	Intel	Commercial	C, C++,	Verilog	All	No	No	SW, HW	Yes	Yes
Vivado HLS [18]	Xilinx	Commercial	C, C++, OpenCL, SystemC	VHDL, Verilog, SystemC	All	Yes	No	SW, HW	Yes	Yes
SDSoC [36]	Xilinx	Commercial	C, C++	VHDL, Verilog	All	No	Yes	SW, HW	Yes	Yes
SDAccel [37]	Xilinx	Commercial	C, C++, OpenCL	VHDL, Verilog, SystemVerilog	All	Yes	Yes	SW, HW	Yes	Yes

Although some of the HLS tools provide a general Software/Hardware simulation framework, the possibility of easily evaluating a complex architecture oriented design (e.g., computer organization: level and size of caches, number of cores/nodes, memory hierarchy) is still missing. Moreover, before reaching a bug free physical design, which meets all the design specifications, the debug and development of such designs by using aforementioned HSL tools may require a significant time and effort despite all benefits that HLS tools provide to the design community. Consequently, powerful design frameworks that simplify the verification of the design and provide an easy design space exploration are welcome. In this respect, many design frameworks have emerged to implement efficient hardware in less time and effort. Authors in [38] propose a framework relying on Vivado HLS to efficiently map processing specifications expressed in PolyMageDSL to FPGA. Their framework support optimizations for the memory throughput and parallelization. ReHLS [39] is a framework with automated source-to-source resource-aware transformation leveraging Vivado HLS tool. Their framework improves the resource utilization and throughput by identifying the program inherent regularities that are invisible by HLS tool. FROST [40] is a framework that generates an optimized design for HLS tool. This framework is mainly appropriate for applications based on streaming dataflow architectures like image processing kernels.

However, these tools focus on optimizing the whole application performance, while we are proposing instead an architecture oriented approach, where the designer can manipulate and explore the architecture itself, before passing it to the HLS toolchain. By using our proposed framework (see Section 4 for more details), we can validate the design in terms of the functional and timing models, and then define a specific architecture, while constantly monitoring the selected key performance metrics. The architecture model is specified in C/C++ and, thanks to the decoupling from the simulation details and functional model, it can be easily migrated into the HLS description. This is illustrated in Sections 4 and 5. In particular, we leverage the Vivado HLS tool and on top of it, we build our design space exploration tools relying on

COTSon simulator, which is one of the key components of our framework. In the following, we highlight relevant features and compare several simulators (Table 2), and we contrast them with our chosen simulator (i.e., COTSon).

SlackSim [23] is a parallel simulator to model single-core processors. SimpleScalar [24] is a sequential simulator, which supports single-core architectures at user-level. GEMS [25] is a virtual-machine based full-system multi-core simulator built on top of the Intel's Simics virtual-machine. GEMS relies on timing-first simulation approach, where its timing model drives one single instruction at a time. Even though GEMS provides a complete simulation environment, we found that COTSon simulator provides better performance as we increase the number of modelled cores and nodes. MPTLsim [26] is a full-system x86-64 multi-core cycle-accurate simulator. In terms of simulation rate, MPTLsim is significantly faster than GEMS. MPTLsim takes advantage of a real-time hypervisor scheduling technique [41] to build hardware abstractions and fast-forward execution. However, during the execution of hypervisor, the simulator components like memory, instructions or I/O are opaque to the user (no statistics is available). On contrary, for example, COTSon provides an easily configurable and extensible environment to the users [42] with full detailed statistics. Graphite [43] is an open-source distributed parallel simulator leveraged the PIN package [44], with the trace-driven functionalities. COTSon permits full-system simulation from multi-core to multi-node and the capability of network simulation, which makes COTSon a complete simulation environment. Both COTSon and Graphite permit a large core numbers (e.g., 1000 cores) with reasonable speed, but COTSon provides also the modelling of peripherals like disk and Ethernet card as well. Compared to COTSon, the above simulators do not express a timing model in a way that can be easily ported to HLS: COTSon is based on the "functional directed" simulation [16], which means that the functional part drives the timing part and the two parts are completely separated both in the coding and during the simulation. The functional model is very fast but does not include any architectural detail, whilst the timing model is an architectural-complete description of the system (and, as such, includes also the actual functional behaviour, of course). In this way, once the timing model is defined and the desired level of the key performance metric (e.g., power or performance) has been reached, the design can be easily transported to an HLS description as it will be illustrated in the next Sections.

Table 2 – Interesting features of simulators for high performance computing architectures. For the non-obvious columns, *Parallel/Sequential* means the simulator core can be executed either in parallel or sequential by the host processor. *Full System* means taking into account all events, including the OS.

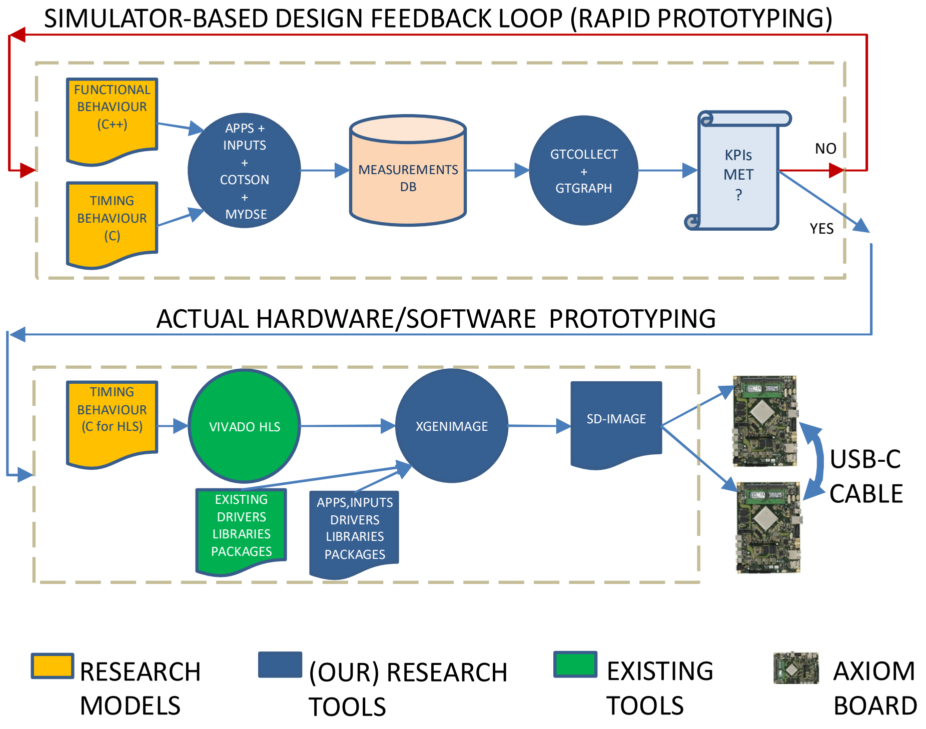
Simulator	**Parallel/ Sequential**	**Single-core/ Multi-core**	**Full System**	**Simulation methodology**
COTSon [16]	***Parallel***	***Multi-core***	***yes***	***Decoupled- functional first***
GEMS [25]	Sequential	Multi-core	yes	Decoupled – timing first
Graphite [43]	Parallel	Multi-core	no	Not-Decoupled – trace driven
SimpleScalar [24]	Sequential	Single-core	no	Not-decoupled – execution driven
MPTLsim [26]	Sequential	Multi-core	no	Not-decoupled – timing first
SlackSim [23]	Parallel	Single-core	no	Not- decoupled – timing first

3. Methodology

In this section, we present our methodology (Figure 1) for developing hardware components for a reconfigurable platform, as developed in the context of the AXIOM project.
First, we define the functional and the timing model of a desired architectural component (e.g., a cache system, as described in the next section 4). Such models are described by using C/C++

(two orange blocks in the top left part of Figure 1). These models are then embedded in the COTSon simulator, which is managed in turn by the MYDSE tools in order to perform the design space exploration [16], [17], [19]. The latter are a collection of different tools, which provide a fast and convenient environment to simulate, debug, optimize and analyse the functional and timing model of a specific architecture and to select the candidate design to be migrated to the HLS (top part of Figure 1).
Afterwards, we manually migrate a validated architecture specification from COTSon to Vivado HLS tool (bottom part of Figure 1), where the user can apply the specific directives defined in the timing model of COTSon into the Vivado HLS. This is possible due to the close syntax of the architecture specification in COTSon and Vivado HLS.
Our framework has the purpose of reducing the total DSE time to define an architecture (as input to Vivado HLS itself). We do not aim to define a precise RTL, but simply to select an architecture suitable as input to Vivado HLS (see Figure 4).

Finally, we pass the generated bitstream by Vivado to the XGENIMAGE, which is a tool that assembles all needed software including drivers, applications, libraries and packages in order to generate the operating system full image to be booted on the AXIOM board. In Figure 1, we highlight *in green* the existing (untouched) tools, *in blue* we highlight the research tools that we developed from scratch or that we modified (like COTSon). In our case, part of the process involves the design of the FPGA board (the AXIOM board). An important capability of the board is also to provide fast and inexpensive clusterization. The simulator allowed us to model exactly this situation, in which the threads are distributed across several boards, through a specific execution model (called DF-Threads). To that extent, the AXIOM board [11] has been designed to include a soft-IP for the routing of data (via RDMA custom messages) and the FPGA transceivers are directly connected to USB-C receptacles, so that four channels at about 18Gbps are available for simple and inexpensive connection of up to 255 boards, without the need of an external switch [12].


SIMULATOR-BASED DESIGN FEEDBACK LOOP (RAPID PROTOTYPING)
FUNCTIONAL BEHAVIOUR (C++)
TIMING BEHAVIOUR (C)
APPS + INPUTS + COTSON + MYDSE
MEASUREMENTS DB
GTCOLLECT + GTGRAPH
KPIs MET ?
NO
YES
ACTUAL HARDWARE/SOFTWARE PROTOTYPING
TIMING BEHAVIOUR (C for HLS)
VIVADO HLS
XGENIMAGE
SD-IMAGE
EXISTING DRIVERS LIBRARIES PACKAGES
APPS,INPUTS DRIVERS LIBRARIES PACKAGES
USB-C CABLE
RESEARCH MODELS
(OUR) RESEARCH TOOLS
EXISTING TOOLS
AXIOM BOARD
Figure 1 – The design and test methodology of the AXIOM involved a mix of simulation (via the COTSon simulator and other custom tools) and FPGA-prototyping (via our custom AXIOM board and hardware synthesis tools (like Vivado HLS) [11].

3.1. DSE Toolset and COTSon Simulator

Based on our experience of the AXIOM project [4], [12], [13], the main motivations behind the choice of the COTSon simulation framework resides in the "functional directed" approach [16]. COTSon also permits to model a complete system like a Cyber-Physical System (CPS), i.e., including the possibility to run a real software performing Input/Output (I/O) and an off-the-shelf Linux distribution (or other operating systems). Since the performance of a CPS is affected also by the Operating System (OS) and libraries [17], it is important to model not only the memory hierarchy and cores but also all the devices of the system: this is possible in the COTSon framework.

In Section 5.1, we show that the OS influence can be detected earlier in the DSE by using our methodology. Moreover, COTSon permits building a complete distributed system with multi-cores and multiple nodes, where we can observe and analyse any aspect of the application and, e.g., the OS activity. In order to guarantee a proper scientific methodology for studying the experiment results that are coming from the framework, we designed a DSE tool-set (called "MYDSE") [19], through which is possible to easily set up a distributed simulation, as well as automatically extract, calculate the appropriate averages and examine the key metrics. MYDSE addresses the designer's needs mostly on the first part of the workflow represented on the top of Figure 1.

Moreover, MYDSE represents a higher abstraction level in the design (Figure 9), in which existing architectural blocks (e.g., caches) can be combined and parameterized for a preliminary design exploration. The MYDSE phase permits us to answer questions like: "how large should be the cache in the target platform?", "how many cores I need in my design?", "what would be the overhead of distributing the computation across several FPGAs?".

3.2. COTSon framework

In this subsection, we briefly summarize the features of the essential component of our toolchain -- the COTSon -- for the sake of a more self-contained illustration of our framework. More details can be found in [16], [17], [19], [42].

The COTSon framework has been initially developed by HP-Labs and its simulation core is based on the AMD SimNow virtualization tool, which is an x86_64 virtual machine provided by AMD to test and develop their processors and platforms [16]. COTSon relies on the so-called "functional-directed" simulation approach, where the functional execution (top part of Figure 2) runs in the SimNow Virtual Machine (VM) and the detailed timing (bottom part of Figure 2) is totally decoupled and reconstructed dynamically based on the events coming from the functional execution.

COTSon can also model a distributed machine composed of several nodes: each SimNow VM models a complete multi-core node with all its peripherals, and an additional component (called "Mediator"), which models a network switch. The virtual machines can run in parallel, thus speeding up a simulation consisting of several nodes. Moreover, we can use different available simulation acceleration techniques, such as dynamic sampling or SMARTS [45], and perform other accounting activities such as tracing, profiling and (raw) statistic collection. The instruction stream coming out from each SimNow functional core is interleaved for a correct time ordering. The COTSon control interface extracts the instruction stream, passing it to the timing simulation (Figure 2).

In the "Timing Simulation" portion of the COTSon (see the bottom part of Figure 2), we can model any architectural components (i.e., CPU, L1 Cache, network switch, accelerator, etc) with a few lines of C++ code. The architecture of the modelled system is customizable by setting all the relevant information in a configuration file (written in the Lua scripting language) [46] as illustrated in the bottom part of Figure 9). Other aspects of the simulation can be customizable as well in the configuration file: for example, the sampling method, how to log statistics, which kind of Operating System (OS) image.

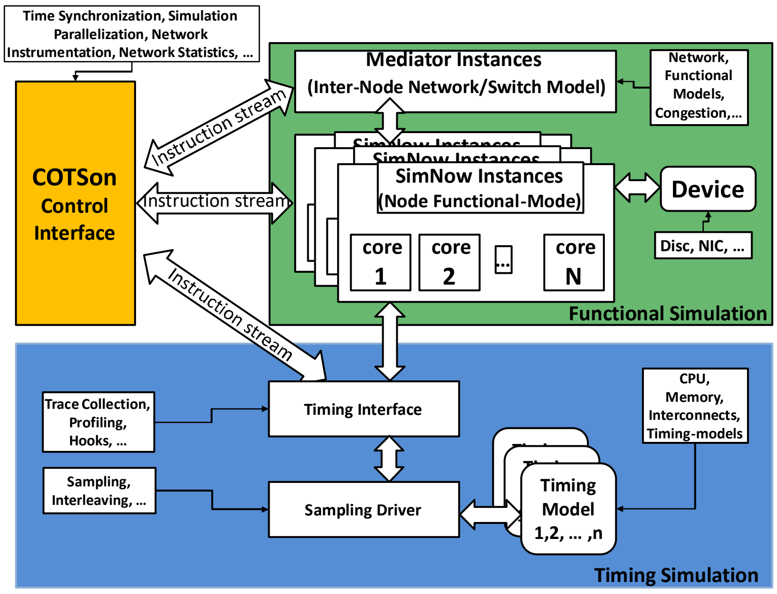

Time Synchronization, Simulation Parallelization, Network Instrumentation, Network Statistics, ...
COTSon Control Interface
Instruction stream
Instruction stream
Instruction stream
Mediator Instances (Inter-Node Network/Switch Model)
Network, Functional Models, Congestion,...
SimNow Instances (Node Functional-Mode)
core 1
core 2
...
core N
Device
Disc, NIC, ...
Functional Simulation
Trace Collection, Profiling, Hooks, ...
Sampling, Interleaving, ...
Timing Interface
Sampling Driver
CPU, Memory, Interconnects, Timing-models
Timing Model 1,2, ... ,n
Timing Simulation

Figure 2 - The COTSon simulation framework architecture [19], [47].

3.3. DSE tool-set

In this subsection, we describe the tools that we have designed for the DSE. A detailed overview of these tools has been introduced in previous work [19]. Here we recall the main features.

Design Space Exploration (DSE) and its automation is a significant part of modern performance evaluation and estimation methodologies to find an optimal solution among the many design options, while respecting several constraints on the system (e.g., a certain level of performance and energy-efficiency).
In order to facilitate and speed up the DSE, we developed a set of tools (called "MYDSE"), through which is possible to easily configure the relevant aspects of our simulation framework and automate the routine work. Thanks to MYINSTALL, a tool included in the MYDSE, the installation and validation phase of the overall environment (which was previously taking a lot of human effort and many hours of work) now takes less than 10 minutes minimizing the human interaction and giving us the possibility to set up several host machines in a fast and easy way. At the end of the installation phase, a set of regression tests is performed to verify if the software is correctly patched, compiled and installed (see Figure 3 - left). This permits a fast deployment of different machines with possibly different characteristics and at the same time have a monitoring of the actual resources that are available for an optimal utilization of them.

Another critical aspect of the simulation is the automatic management of experiments, mostly in the case of a large number of design points to be explored: this is managed by the MYDSE tool. Using a small configuration file, we can define the Design Space of an experiment by using a simple scripting syntax[2]: <key>=<value>. For example, it is possible not only to define a modelled architecture (e.g., number and types of cores, cache parameters, multiple levels of caches and so on), the Operating System image and other parameters of the COTSon simulator, but also to define other higher-level parameters related to the applications, their inputs, and the standard libraries to be used. The MYDSE configuration file also permits listing a set of values for each parameter so that the design points are automatically generated. Once the design points are generated, the tools manage the execution of the experiments by scheduling and distributing the simulations on, e.g., a cluster of simulation hosts, by collecting the results of each simulation and inserting them in a database, where off-line data mining can be performed afterwards. Moreover, the tools constantly monitor the simulations: if one of them is failing then it is automatically retried (thresholds are applied to limit the re-trials).

A large number of output statistics are produced during the simulations, thus a database is necessary to store such data. Statistical processing can also be selected to give a quantification of the goodness of the collected numbers (e.g., the Coefficient of Variation, the presence of outliers and so on). Other tools in Figure 3 (GENIMAGE, ADD-IMAGE, GTCOLLECT, GTGRAPH) are described in more details in [19].

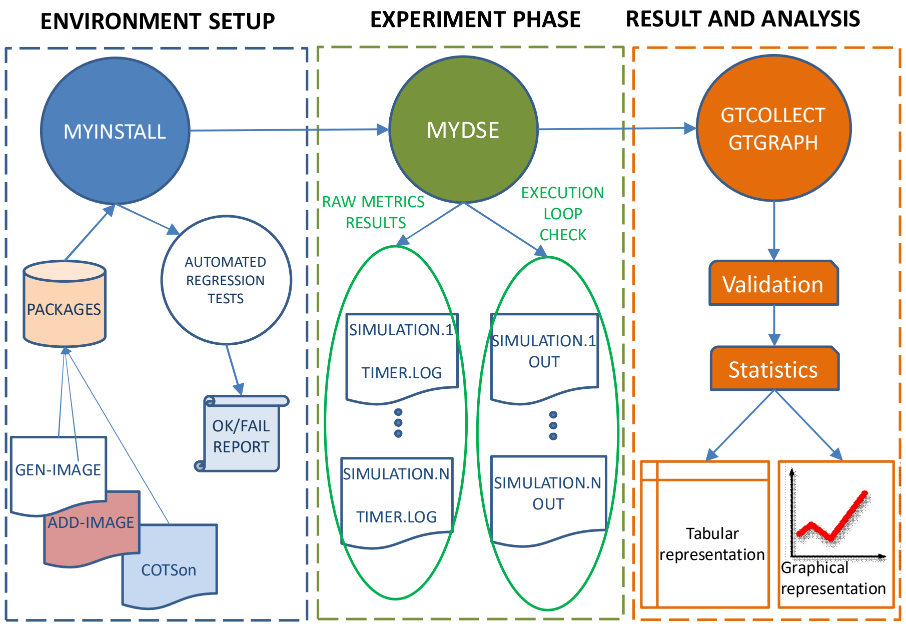


Figure 3 – Tool-flow of the Design Space Exploration tool. The MYINSTALL tool prepares the whole environment and performs automated regression tests in the end. The MYDSE tool takes care of the experiments loop and the re-ordering of the several output files generated by each simulation. Finally, the GTCOLLECTS and GTGRAPH tools collect the results, perform validation and statistical operations on the results, and plot the data in a tabular or graphical format [19].

3.4. Mapping an Architecture to HLS

High-level Synthesis (HLS) aims at enhancing design productivity via facilitating the translation from the algorithmic level to RTL (Register Transfer Level) [48], [49]. In current state-of-the-art, given an application written in a language like C/C++ or SystemC, an HLS tool particularly performs a set of successive tasks to generate the corresponding Register

[2] In our case, we refer to “bash”. “bash” is a popular scripting language for Linux.

Transfer Level (RTL, for example, VHDL or Verilog) description suitable for a reconfigurable platform, such as an FPGA [49] (Figure 4 - left). This workflow typically involves the following steps:

- Compiling the C/C++/SystemC code to formal models, which are intermediate representations based on control flow graph and data flow graph.
- Scheduling each operation in the generated graph to the appropriate clock-cycles. Operations without data dependencies could be performed in parallel, if there are enough hardware resources during the desired cycle.
- Allocating available resources (LUTs, BRAMS, FFs, DSPs and so on) in regards to the design constraints. For instance to enhance the parallelism, different resources could be statistically allocated at the same cycle without any resource contention.
- Binding each operation to the corresponding functional units, binding the variables and constants to the available storage units as well as data paths to data buses.
- Generating the RTL (i.e., VHDL or Verilog).

All these operations continue to be performed in our proposed framework, but the designer would like to avoid excessive iterations through them, since they may require many hours of computing processing or even more, depending on the complexity of the design, even on powerful workstations and with not so big designs. However, COTSon and MYDSE tools (illustrated above) act like a “front-end” to the HLS tool, as outlined in Figure 4. We use HLS also for defining a specific architecture to accelerate the application. Our tools allow the designer to explore possible options for the architecture, without going to the synthesis step: only when the simulation phase has successfully selected an architecture (output of the blue block in Figure 4), the model will be manually translated by the programmer as an input to the HLS synthesis tools. Doing this step automatically is out of the scope of this work.

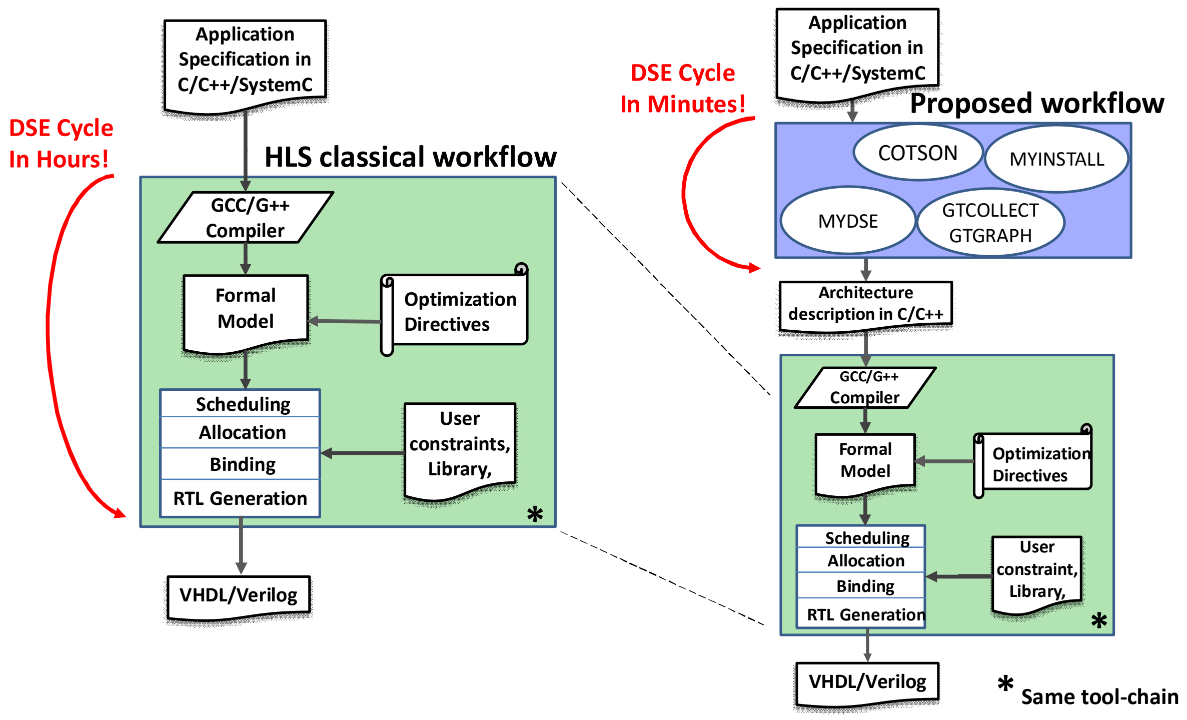


Figure 4 – Differences between classical and proposed architecture modelling framework. Workflows to generate VHDL/Verilog hardware description language from the application specification written in C/C++. On the left, a typical workflow of existing HLS tools. On the right, we leverage the HLS tool, and on top of it, we build our framework to simulate and validate the design specification.

Table 3 – Comparison of different total DSE time of the classical design workflow for FPGAs (Figure 4 - left) and our proposed methodology (Figure 4 - right).

Application	HLS+Synthesis (Figure 4- left)	Our Framework (Figure 4 – right)
2-Way Cache	3:50 Hours	5 Seconds
Blocked Matrix Multiplication (DF-Threads, matrix size = 864, block size=8, integer)	4:25 Hours	8 Seconds
Fibonacci (DF-threads, N=35)	1:40 Hours	8 Seconds

A comparison of the total time of the DSE loops between our framework (Figure 4 - right) and HLS (Figure 4 - left) is reported here for different benchmarks (Table 3). For example, a Blocked Matrix Multiplication benchmark (matrix size 864, and block size 8), and a Fibonacci benchmark (order of up to 35) are executed based on our DF-Thread execution model (Data-flow model). As a result, thanks to our framework, we were able to reduce the required time in validating and developing the architecture compared with solely HLS workflow, through which applying any changes in the source codes may require many hours for the synthesis process.

4. Case Study

In this section, first we want to explain our workflow by using a simple and well-known driving example, i.e., the design of a two-way set associative cache in a reconfigurable hardware platform through our methodology. Afterwards, we will illustrate the more powerful capabilities of our framework for a more complex example, which is the design of the AXIOM hardware/software platform. In both cases, first we design the architecture in the COTSon simulator, then we test its correct functioning and achieve the desired design goals. Finally, we migrate the timing description of the desired architecture into the Xilinx HLS tools.

4.1. From COTSon to Vivado HLS – a simple example

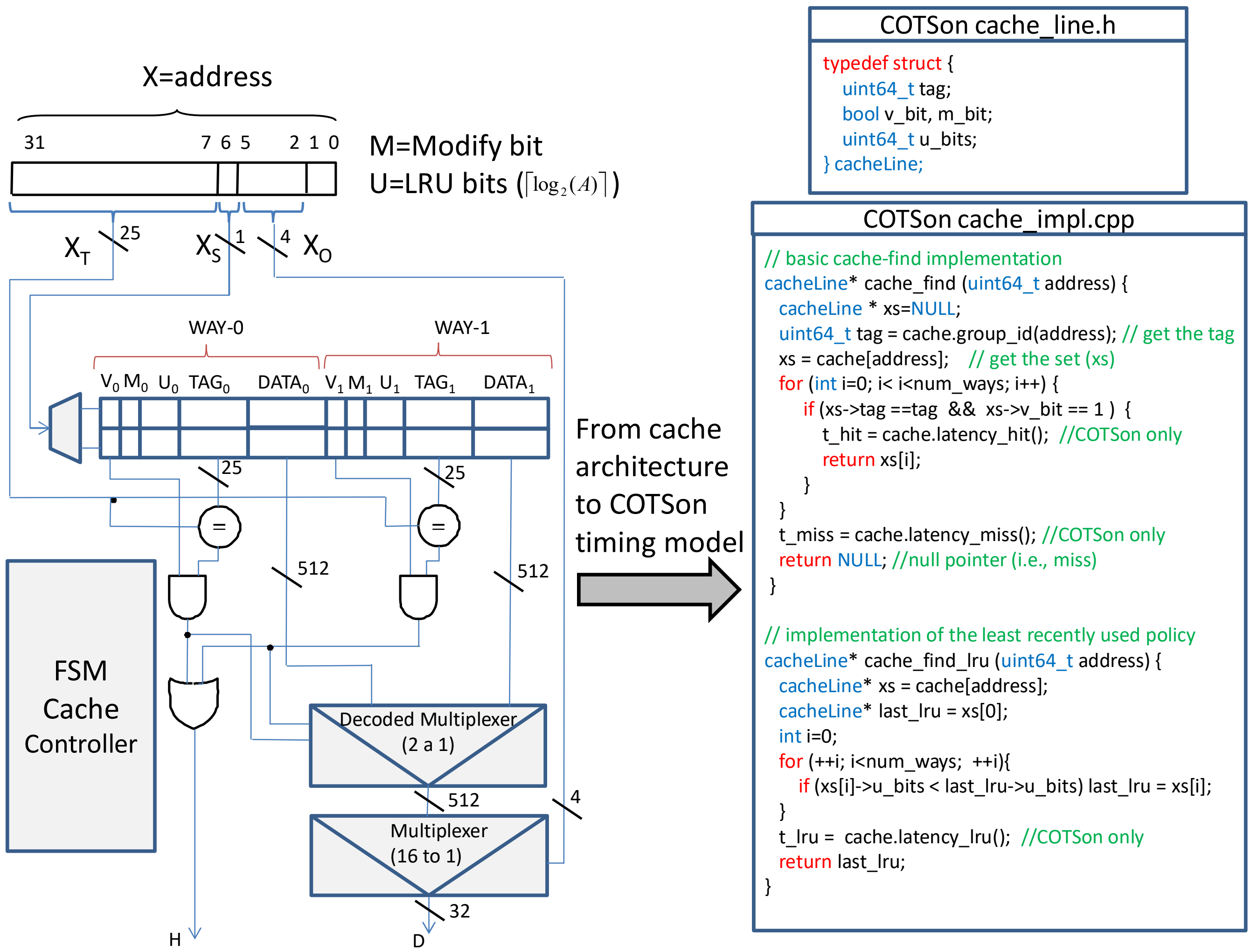
In COTSon, the architecture is defined by detailing its "timing model". A timing model is a formal specification that defines a custom behaviour of a specific architectural or micro-architectural component, in other terms the timing model defines the architecture itself [16], [19]. The timing model in the COTSon simulator is specified by using C/C++. The designer defines the *storage* by using C/C++ variables (more often structured variables). The timing model *behaviour* is specified by *explicating* into C/C++ statements the steps performed by the control part and *associating them with the estimated latency*, which can be defined through our DSE configuration files (see Figure 9) easily. After defining the model, we can simulate and measure the performance of it. This is illustrated in Figure 5, and discussed in the following.

Let us assume here that we wish to design a simple two-way set-associative cache: we show how it is possible to define the timing model of a simple implementation of it in COTSon and then how we can map it in HLS. We start here from a conceptual description of such cache, as shown in Figure 5. In particular, for each way of the cache, we need to store the "line" of the cache, i.e., the following information:

1. **Valid bit or V-bit (1 bit)**: used to check the validity of the indexed data;
2. **Modify bit or M-bit (1 bit)**: used to track if data has been modified;
3. **LRU bits or U-bits (e.g., 1 bit in this case)**: used to identify the Least Recently Used data between the two cache ways;
4. **Tag (e.g., 25 bits)**: used to validate the selected data of the cache;
5. **Data (e.g., 512 bits, 64 bytes, or 16 words):** contains the (useful) data.

The data structures to store this information in COTSon is given by the “Line” structure, which is shown in Figure 6 (left side).

When we want to read or write data, which are stored in a byte address (X in Figure 5), we check if the data are already present into the cache. The cache controller implements the algorithm to find the data in the cache. Although not visible in the left part of Figure 5, there is a control part also for identifying the LRU block. We can implement this control in COTSon by using the two functions (shown in the right part): one named “find” (Figure 6), which is a simple linear search, and the other one named “find_lru” (Figure 7).


X=address
31 7 6 5 2 1 0
M=Modify bit
U=LRU bits ($\lceil \log_2(A) \rceil$)
X_T 25
X_S 1
4 X_O
WAY-0
WAY-1
V_0 M_0 U_0 TAG_0 $DATA_0$ V_1 M_1 U_1 TAG_1 $DATA_1$
25
25
=
=
512
512
FSM Cache Controller
Decoded Multiplexer (2 a 1)
512
4
Multiplexer (16 to 1)
32
H
D
From cache architecture to COTSon timing model

COTSon cache_line.h

```
typedef struct {
   uint64_t tag;
   bool v_bit, m_bit;
   uint64_t u_bits;
} cacheLine;
```

COTSon cache_impl.cpp

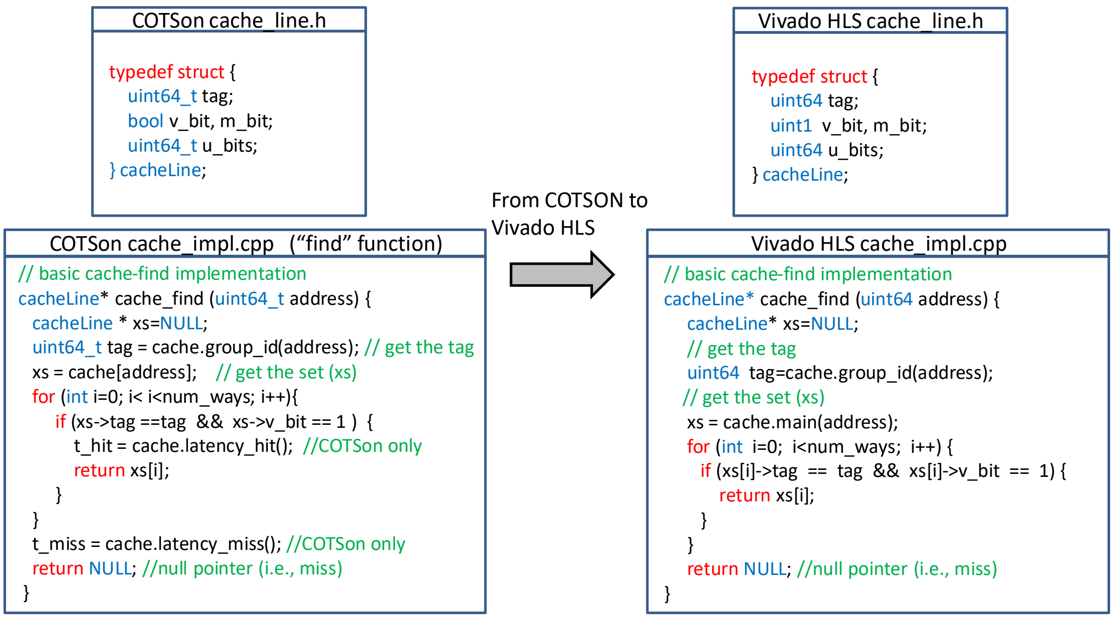
```
// basic cache-find implementation
cacheLine* cache_find (uint64_t address) {
  cacheLine * xs=NULL;
  uint64_t tag = cache.group_id(address); // get the tag
  xs = cache[address];    // get the set (xs)
  for (int i=0; i< i<num_ways; i++) {
      if (xs->tag ==tag  &&  xs->v_bit == 1 )  {
         t_hit = cache.latency_hit();  //COTSon only
         return xs[i];
      }
  }
  t_miss = cache.latency_miss(); //COTSon only
  return NULL; //null pointer (i.e., miss)
 }

// implementation of the least recently used policy
cacheLine* cache_find_lru (uint64_t address) {
  cacheLine* xs = cache[address];
  cacheLine* last_lru = xs[0];
  int i=0;
  for (++i; i<num_ways;  ++i){
     if (xs[i]->u_bits < last_lru->u_bits) last_lru = xs[i];
  }
  t_lru =  cache.latency_lru();  //COTSon only
  return last_lru;
}
```

Figure 5 – Example of logic scheme of a two-way set associative cache. Given the byte address X on 32 bits, in this example, the cache indexes four 64-byte blocks (2 words in 2 sets). This implies that the last 6 bits are needed to select a byte inside the block, the first 25 bits of the address (X_T) is used for the tag comparison and the remaining 1 bit (Xs) is used for the cache set indexing. The cache hit (signal H) is set if the tag of the X is present in the cache at the specified index and if the valid bit is equal to one.

From the timing model of the implemented cache in COTSon, we migrate the design into the Xilinx HLS tools. One minor restriction in Vivado HLS is to use fixed size arrays instead of dynamic data structures, due to the direct transformation of the structures to the available hardware resources.

The advantage of using our hybrid methodology is that the DSE (see Figure 4 and Table 3) of the architecture of this small cache takes a few seconds in the COTSon, while it takes approximately four hours on a powerful workstation to synthesize and perform the DSE with the HLS version of the same architecture (right side of Figure 6 an dFigure 7).

In the next Section, we will illustrate how, thanks to our methodology, we were able to reduce significantly the DSE cycles and development time of a relatively large project like the AXIOM project, and produce reliable specification to be implemented on the AXIOM board.


COTSon cache_line.h
Vivado HLS cache_line.h
From COTSON to Vivado HLS
COTSon cache_impl.cpp (“find” function)
Vivado HLS cache_impl.cpp

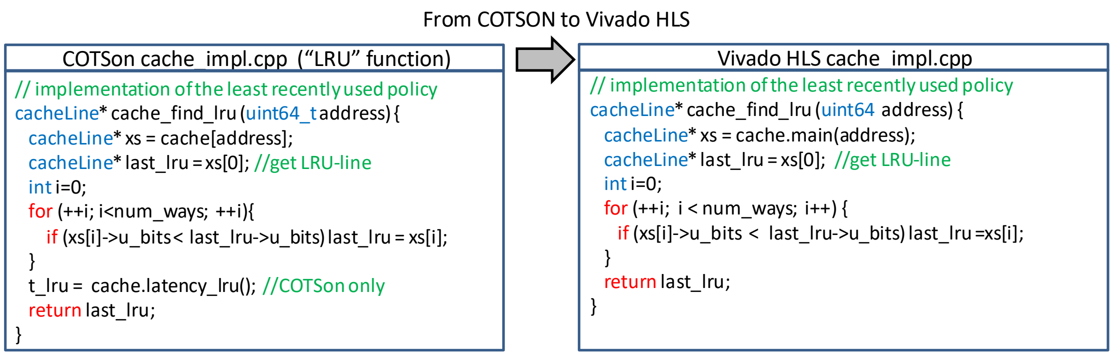
```
typedef struct {
   uint64_t tag;
   bool v_bit, m_bit;
   uint64_t u_bits;
} cacheLine;
```

```
typedef struct {
   uint64 tag;
   uint1  v_bit, m_bit;
   uint64 u_bits;
} cacheLine;
```

```
// basic cache-find implementation
cacheLine* cache_find (uint64_t address) {
  cacheLine * xs=NULL;
  uint64_t tag = cache.group_id(address); // get the tag
  xs = cache[address];    // get the set (xs)
  for (int i=0; i< i<num_ways; i++){
     if (xs->tag ==tag  &&  xs->v_bit == 1 )  {
        t_hit = cache.latency_hit();  //COTSon only
        return xs[i];
     }
  }
  t_miss = cache.latency_miss(); //COTSon only
  return NULL; //null pointer (i.e., miss)
}
```

```
// basic cache-find implementation
cacheLine* cache_find (uint64 address) {
   cacheLine* xs=NULL;
   // get the tag
   uint64  tag=cache.group_id(address);
  // get the set (xs)
   xs = cache.main(address);
   for (int  i=0;  i<num_ways;  i++) {
     if (xs[i]->tag  ==  tag  &&  xs[i]->v_bit  ==  1) {
        return xs[i];
     }
   }
   return NULL; //null pointer (i.e., miss)
}
```

Figure 6 – Example of timing model of the cache “find” function, which is translated from the COTSon to the Vivado HLS. The implementation of this function for the both COTSon (left) and Vivado HLS (right) environments is shown in the bottom part of the figure.


From COTSON to Vivado HLS
COTSon cache_impl.cpp (“LRU” function)
Vivado HLS cache_impl.cpp

```
// implementation of the least recently used policy
cacheLine* cache_find_lru (uint64_t address) {
  cacheLine* xs = cache[address];
  cacheLine* last_lru = xs[0]; //get LRU-line
  int i=0;
  for (++i; i<num_ways; ++i){
     if (xs[i]->u_bits< last_lru->u_bits) last_lru = xs[i];
  }
  t_lru = cache.latency_lru(); //COTSon only
  return last_lru;
}
```

```
// implementation of the least recently used policy
cacheLine* cache_find_lru (uint64 address) {
  cacheLine* xs = cache.main(address);
  cacheLine* last_lru = xs[0];  //get LRU-line
  int i=0;
  for (++i;  i < num_ways;  i++) {
    if (xs[i]->u_bits  <  last_lru->u_bits) last_lru =xs[i];
  }
  return last_lru;
}
```

Figure 7 - Example of translation of the timing model of the LRU (Least Recently Used) function from the COTSon (left side) to the Vivado HLS (right side).

4.2. COTSon configuration and Timing

One more feature of our environment, based on COTSon+MYDSE, is the capability of easily integrating the modelled components (i.e., the simple two-way set associative cache of the previous subsection). As we can see in Figure 8, we can build the overall architecture by specifying how to integrate the component in a higher-level configuration file ("Level-2" in Figure 8, the "MYDSE" configuration file).
In particular, we define the following simple syntax: the character "-" is the link between two architectural COTSon blocks and the "+" character separates different links between such blocks. The architectural blocks are implicitly defined, since they appear in the link specification. The "." character serves to replicate a set of architectural blocks, which follow the ".", for m times, where – by default – "m" is the number of cores. This is shown in the "listarch" variable of Figure 8: i.e., the part "l2-ic+l2-dc+ic-cpu+dc-cpu" will be instantiated "m" times.

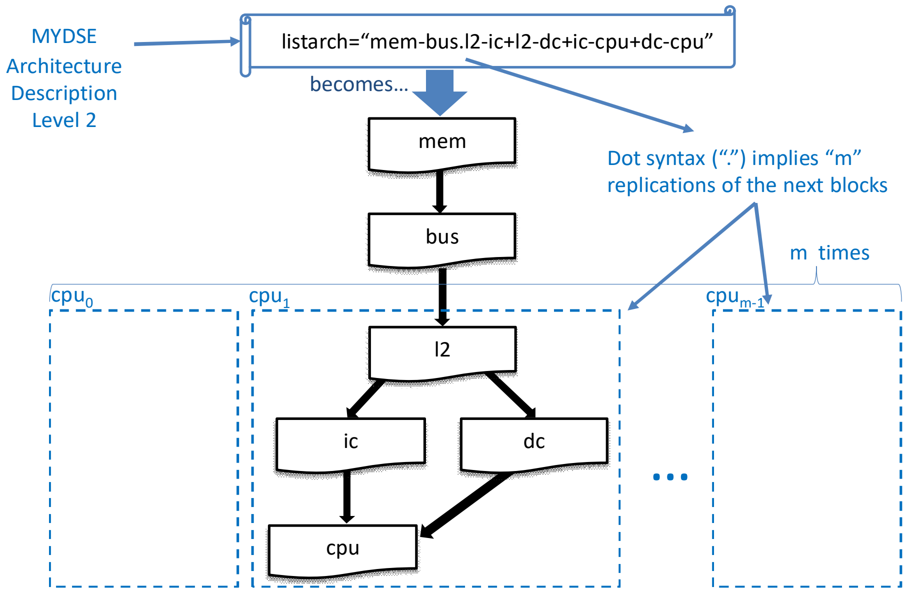
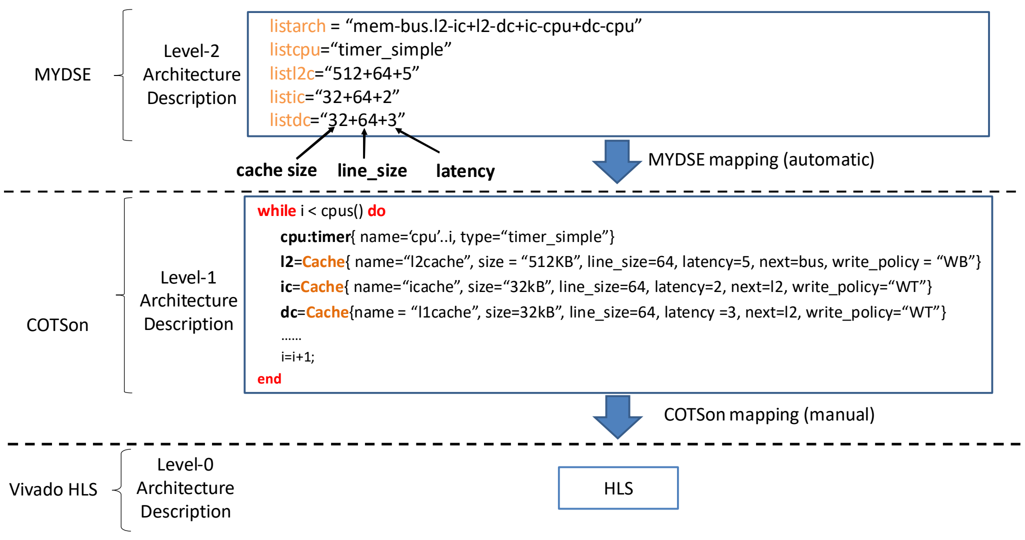

MYDSE Architecture Description Level 2
listarch="mem-bus.l2-ic+l2-dc+ic-cpu+dc-cpu"
becomes…
mem
bus
Dot syntax (".") implies "m" replications of the next blocks
m times
cpu_0
cpu_1
cpu_{m-1}
l2
ic
dc
cpu
…

Figure 8 – Level 2 architecture description of the cache model in COTSon by using MYDSE toolset. In this design, the CPU is directly connected to both an Instruction Cache ("ic") and a Data Cache ("dc"). The "ic" and "dc" caches are then connected to another level of caching, the L2-Cache ("l2"), which is connected to the main memory ("mem") through the "bus".

As depicted in Figure 9, at the higher level, we specify the parameters in an even compact way, and we can indicate several instances of such parameters so that MYDSE can generate the design space points to be explored. In the COTSon configuration, the MYDSE points will be assigned to the parameter of the corresponding architectural element. Moreover, we can specify the latencies of an architectural block, which are used by its timing model for the execution time estimation.



```
listarch = "mem-bus.l2-ic+l2-dc+ic-cpu+dc-cpu"
listcpu="timer_simple"
listl2c="512+64+5"
listic="32+64+2"
listdc="32+64+3"
```

```
while i < cpus() do
    cpu:timer{ name='cpu'..i, type="timer_simple"}
    l2=Cache{ name="l2cache", size = "512KB", line_size=64, latency=5, next=bus, write_policy = "WB"}
    ic=Cache{ name="icache", size="32kB", line_size=64, latency=2, next=l2, write_policy="WT"}
    dc=Cache{name = "l1cache", size=32kB", line_size=64, latency =3, next=l2, write_policy="WT"}
    ......
    i=i+1;
end
```

MYDSE
Level-2 Architecture Description
cache size
line_size
latency
MYDSE mapping (automatic)
COTSon
Level-1 Architecture Description
COTSon mapping (manual)
Vivado HLS
Level-0 Architecture Description
HLS

Figure 9 – The relation between the higher level MYDSE description, the COTSon configuration file and the final HLS translation: At the higher level, we specify the parameters in a compact way (level-2 architecture description), and we can indicate several instances of such parameters so that MYDSE can generate the design space points to be explored. In the COTSon configuration (level-1), the MYDSE points will be automatically mapped to the parameter of the corresponding architectural element (bottom part of the figure: the "next" field specifies the position in the architecture tree, WB means write-back and, WT is the write-through policy). Finally, the architecture description is mapped manually from COTSon description to HLS (level-0).

5. Generalization to the AXIOM project and evaluation

The aim of the AXIOM project was to define a software/hardware architecture configuration, to build scalable embedded systems, which could allow a distributed computation across several boards by using a transparent scalable method like the DF-Threads [20]–[22].

In order to achieve this goal, we rely on RDMA capabilities and a full operating system to interact with the OS scheduler, memory management and other system resources. Following our methodology, we included the effects of all these features thanks to the COTSon+MYDSE full-system simulation framework. We will present in the next subsection the results that we were able to obtain through this preliminary DSE phase reasonably quickly.
After the desired software and hardware architecture was selected in the simulation framework, we started the migration to the physical hardware: we had clear that we needed at least the following features:

i) Possibility to exchange rapidly data frames via RDMA across several boards: this could be implemented in hardware thanks to the FPGA high-speed transceivers;
ii) Possibility to accelerate portions of the application on the Programmable Logic (PL), not only on one board but also on multiple FPGA boards: this could be implemented by providing appropriate network-interface IPs in the FPGA.

In this way, we preselected the basic features of the AXIOM board (Figure 10 - left) through the COTSon framework and the MYDSE toolset. Then, once the DSE was completed, we migrated the final architecture specification with the Vivado HLS tool into the AXIOM Distributed Environment (Figure 10 - right).

The DF-Threads execution model is a promising approach for achieving the full parallelism offered by a multi-core and multi-node systems, by introducing a new execution model, which internally represents an application as a direct graph named Data-Flow graph. Each node of the graph is an execution block of the application and a block can execute only when its inputs are available [20].

5.1. Designing the AXIOM Software/Hardware platform

During the AXIOM project, we analysed two main real-world applications, Smart Video Surveillance (SVS) and Smart Home Living (SHL) [50]. These applications are very computational demanding, since they require analysing a huge number of scenes coming from multiple cameras located, e.g., at airports, home, hotels or shopping malls.

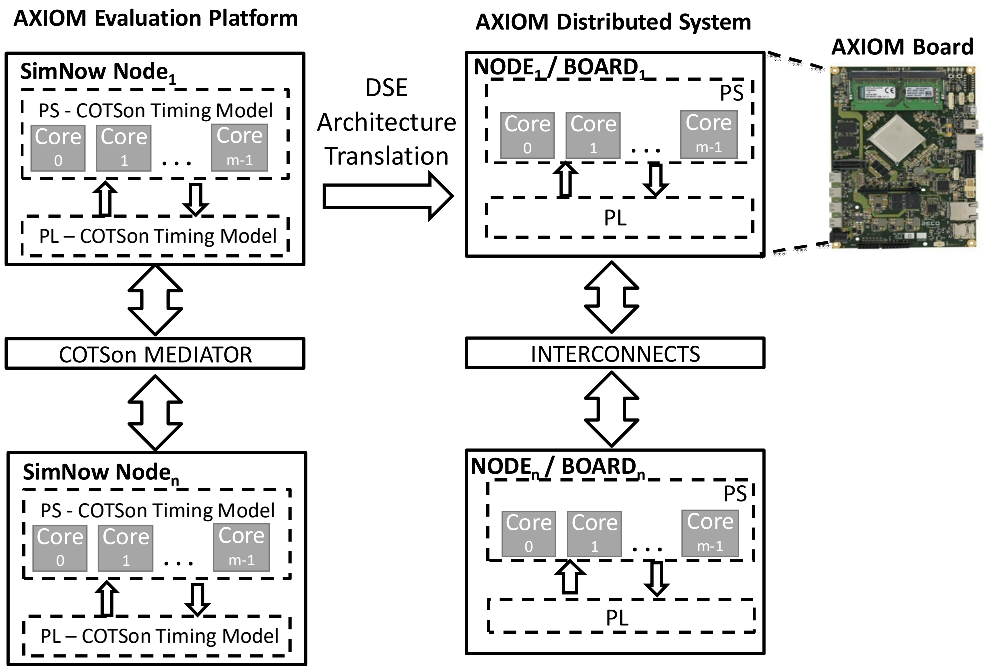


Figure 10 - From COTSon Distributed System definition to AXIOM Distributed System by using the DSE Tools. The Processing System (PS), the Programmable Logic (PL) and the Interconnects of the AXIOM Board are simulated and evaluated into the COTSon framework with the definition of the respective timing models.

In these scenarios, we figured out that one of the computationally intensive portions of those applications relies on the execution of the Matrix Multiplication kernel. For these reasons, the experiment results presented in this section are based on the execution of the Block Matrix Multiplication benchmark (BMM) using the DF-Threads execution model. The BMM algorithm is based on the classical three nested loops, where a matrix is partitioned into multiple sub-matrices, or blocks, according to the block size.

As we generalized the methodology described in the previous section to the AXIOM project [10]–[12], we were able to experiment on the simulator our DF-Threads execution model [15], [20], [21] before spending time consuming development on the reconfigurable hardware. We show here some evaluations that are possible within the MYDSE and COTSon framework once applied to the test case of the DF-Threads modelling. In such a test case, we aim to understand the impact of architectural and operating system choices on the execution time of our novel Dataflow execution model [21].

Thanks to the MYDSE, we were also able to easily explore different architecture parameters, e.g., for the L2 cache size (from 2 to 1024kB) and for the number of nodes/boards (ranging from one to four). Thus, in the case of deploying a soft-processor and its peripherals on the FPGA, the designer can choose safely a well-optimized configuration for, e.g., the L2 cache size.

Moreover, we choose different Operating System (OS) distributions to analyse the overhead produced by the OS in a target architecture: four different Ubuntu Linux distributions have been used: Karmic (or Ubuntu 9.10 – label "karmic64"), Maverick (or Ubuntu 10.10 – label "tfxv4"), Trusty (or Ubuntu 14.04 – label "trusty-axmv3"), and Xenial (or Ubuntu 16.04 – label "xenv0"). The different architecture configurations used in the experimental campaign are summarized in Table 4.

Table 4 – COTSon architectural parameter

Parameter	Description
SoC	1-core connected by a shared-bus, IO-bus, MC, high-speed transceivers
Core	3GHz, in-order super-scalar
Branch Predictor	Two-level (history length=14bits, pattern-history Table=16KiB, 8-cycle miss-prediction penalty)
L1 Cache	Private I-cache 32KiB, private D-cache 32 KiB, 2 ways, 3-cycle latency
L2 Cache	Private 2,8,32,64,256,1024 KiB, 4 ways, 5-cycle latency
L3 Cache	Shared 4MiB, 4 ways, 20-cycle latency
Coherence protocol	MOESI
Main Memory	1 GiB, 100 cycles latency
I-L1-TLB, D-L1-TLB	64 entries, direct-access, 1-cycle latency
L2-TLB	512 entries, direct access, 1-cycle latency
Write/Read queues	200 Bytes each, 1-cycle latency

The simulation framework permits exploring the execution of our benchmark easily, while we vary, e.g., the number of nodes (1, 2, 4), the OS. The input size of the shown example is fixed (matrix size=512 elements).

As can be seen in Figure 13, there is a large variation of the kernel cycles between "xenv0" (Linux Ubuntu 16.04) and the other three Linux distributions. This indicated us to put attention on the precise configuration of many daemons that run in the background and that may affect the activity of the system. While doing tests directly on the FPGA, it would not have been easy to understand that most of the time taken by the execution was actually absorbed by the OS activity: a designer could have taken it for granted or he/she could have not even had the possibility of changing the OS distribution for testing the differences, since the whole FPGA workflow is typically oriented to a fixed decision for the OS (e.g., Xilinx Petalinux). The situation is even worse for cache parameters or for the number of cores, since the designer might be forced to choose a specific configuration.

However, the important information for us was to confirm the scaling of the DF-Threads model, while we increase the number of nodes/boards. We can observe that the number of cycles is decreasing almost linearly - except the case of "xenv0", which is decreasing sublinearly - when we use two and four nodes compared to the case of a single board/node (Figure 15). Moreover, we were able to understand the size of the cache that we should use in the physical system in order to properly accommodate the working set of our applications.

We further explored the reasons why we can obtain a good scaling in the execution time with more nodes by analyzing the behavior of L2 cache miss rate (Figure 11, Figure 12).
Again, this type of measurement was conveniently done in the simulator, while it is more difficult to perform it on the Xilinx Vivado HLS model especially when it comes to design a

soft-processor and choose the best configuration (e.g., size of L2-cache, and OS). In particular, we varied again the number of nodes (1, 2, and 4), the OS distribution as before and the cache size for L2 with larger values (64KiB, 256KiB, 1024KiB, to allow a wider range of exploration of the L2 cache). As we can see from Figure 12, the L2-cache miss rate is decreasing for all OS distributions while we vary the number of nodes, thus confirming that this is one of the main factors of the improvement of the execution time. Moreover, we can analyze which OS distribution leads to best performance. For example, the "xenv0" produces a huge amount of kernel activity during the computation (Figure 13). However, the combined effect of the kernel activity (Figure 13) and the average data latency (Figure 14) - considering L1, L2 and L3 caches - may affect the total execution time (Figure 15) quite heavily. Thanks to this preliminary DSE, we found that the OS distribution with best trade-off between memory accesses and kernel utilization is the "trusty-axmv3".

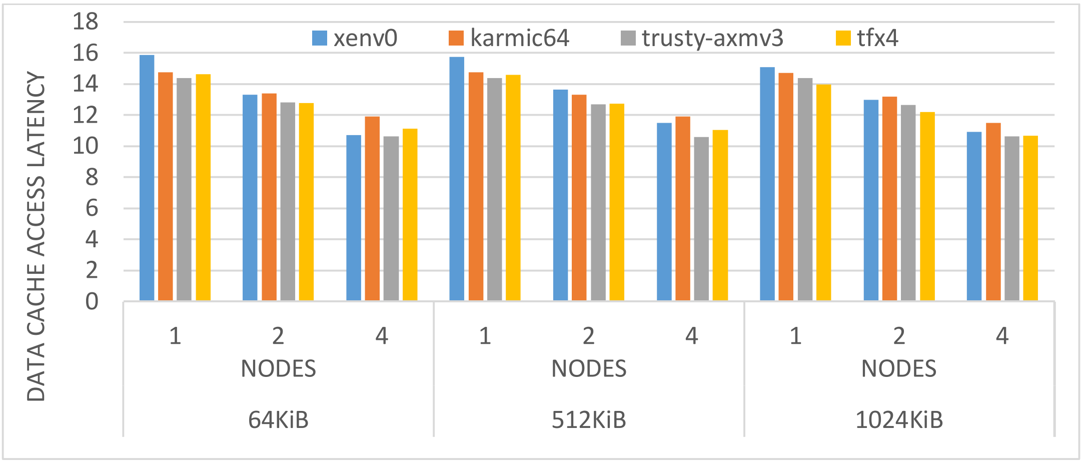


Figure 11 – Evaluation of the Data Access Latency in the COTSon framework when using the Matrix Multiplication benchmark by varying cache size, number of nodes of the distributed system and different Linux distribution ("xenv0"=Ubuntu 16.04, "karmic64"=Ubuntu9.10, "trusty-axmv3"=Ubuntu 14.04, "tfxv4"=Ubuntu 10.10). The DSE shows that the data-cache access latency is almost similar in each Linux distribution, but it is lowering when we increase the number of nodes. Thus, multiple nodes configuration can be more convenient in the DF-threads execution model.

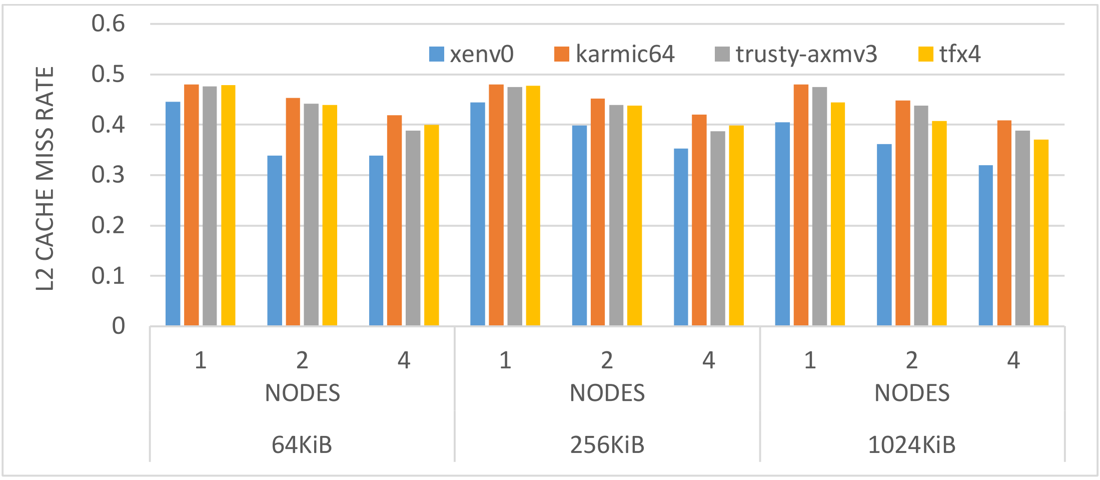


Figure 12 – Evaluation of the L2 Cache miss rate in the COTSon framework when using the Matrix Multiplication benchmark by varying cache sizes, number of nodes of the distributed system and different Linux distribution ("xenv0"=Ubuntu 16.04, "karmic64"=Ubuntu9.10, "trusty-axmv3"=Ubuntu 14.04, "tfxv4"=Ubuntu 10.10). The Linux distribution "xenv0" shows the lowest L2 miss rate compared to the other Linux distributions for all the presented configurations.

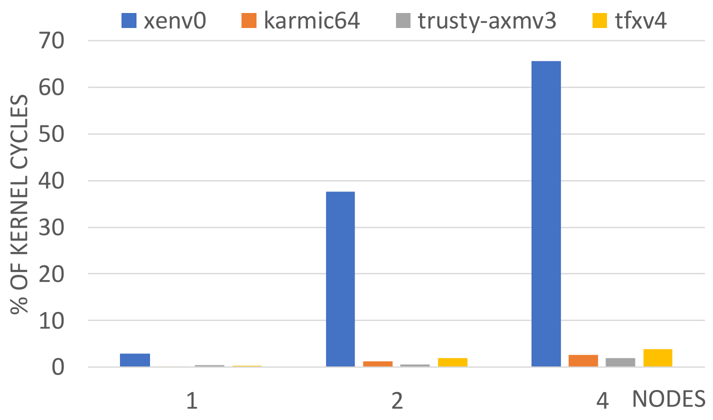


Figure 13 – Percentage of kernel cycles (over total number of cycles) in the COTSon framework when using the Matrix Multiplication benchmark with 512 as matrix size and 32KiB as cache size. We varied the number of nodes of the distributed system and the Linux distribution ("xenv0"=Ubuntu 16.04, "karmic64"=Ubuntu9.10, "trusty-axmv3"=Ubuntu 14.04, "tfxv4"=Ubuntu 10.10). This DSE test permitted to detect a much larger kernel activity of the "xenv0" distribution compared to the other three Linux distributions both in a single-node or multiple-node configurations.

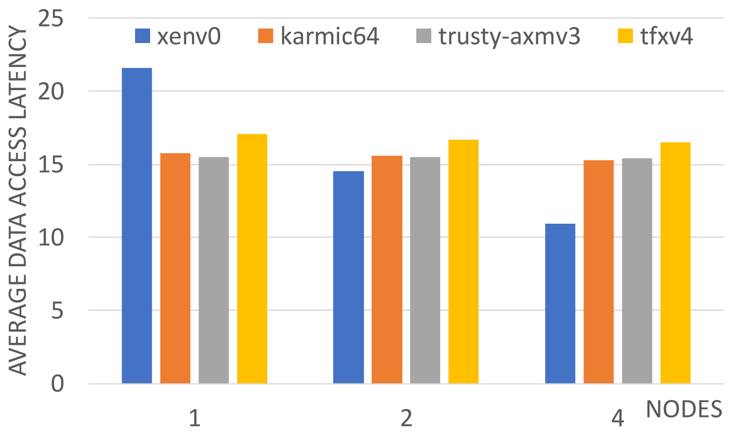


Figure 14 – Average data latency in the COTSon framework when using the Matrix Multiplication benchmark with 512 as matrix size and 32k as cache size. We varied the number of nodes of the distributed system and the Linux distribution ("xenv0"=Ubuntu 16.04, "karmic64"=Ubuntu9.10, "trusty-axmv3"=Ubuntu 14.04, "tfxv4"=Ubuntu 10.10). The data access latency of "xenv0"is improved when we have more nodes. This improvement has less impact on total cycles (Figure 15) than the impact of kernel activity (Figure 13).

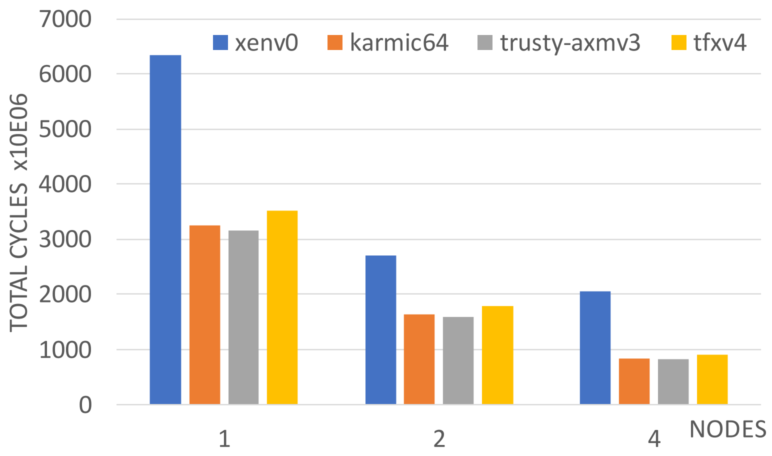


Figure 15 – Total number of cycles in the COTSon framework when using the Matrix Multiplication benchmark with 512 as matrix size and 32k as cache size. We varied the number of nodes of the distributed system and the Linux distribution ("xenv0"=Ubuntu 16.04, "karmic64"=Ubuntu9.10, "trusty-axmv3"=Ubuntu 14.04, "tfxv4"=Ubuntu 10.10). The DSE allow us to detect that the four Linux distributions permit to obtain a good scalability when we increase the number of nodes. However, the "xenv0" confirms the worst performance in terms of executed cycles, due to the huge number of kernel cycles shown in the Figure 13.

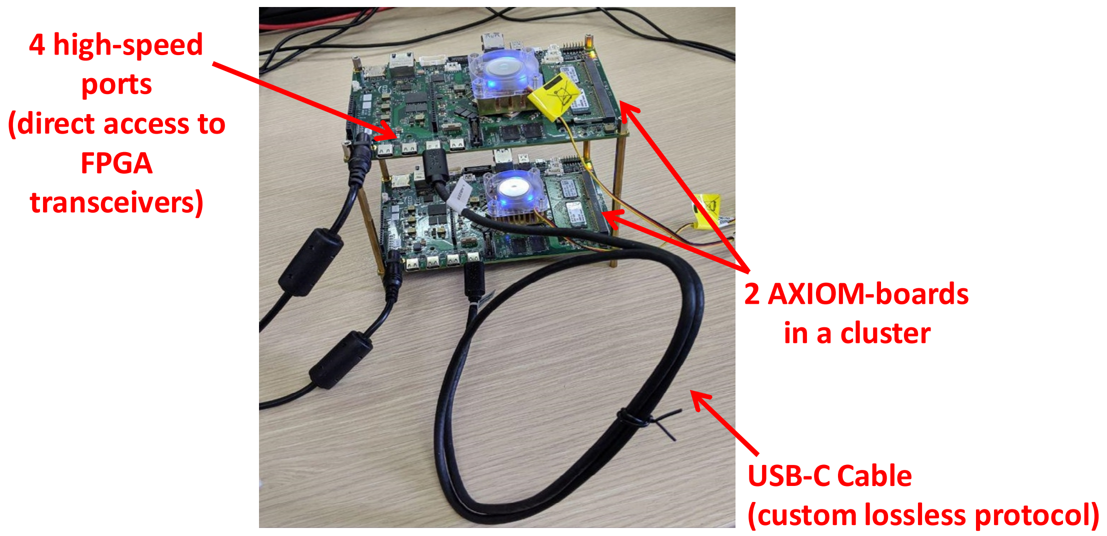

4 high-speed ports (direct access to FPGA transceivers)
2 AXIOM-boards in a cluster
USB-C Cable (custom lossless protocol)

Figure 16 – Two AXIOM boards interconnected up to 18-Gbps via inexpensive USB-C cables. The AXIOM board is based on a Xilinx Zynq Ultrascale + ZU9EG platform, four high-speed ports (up to 18-Gbps), an Arduino socket, and DDR4 extensible up to 32 GiB. As can be seen from the picture we do not need any external switch but just two simple USB-C cables to connect the two systems.

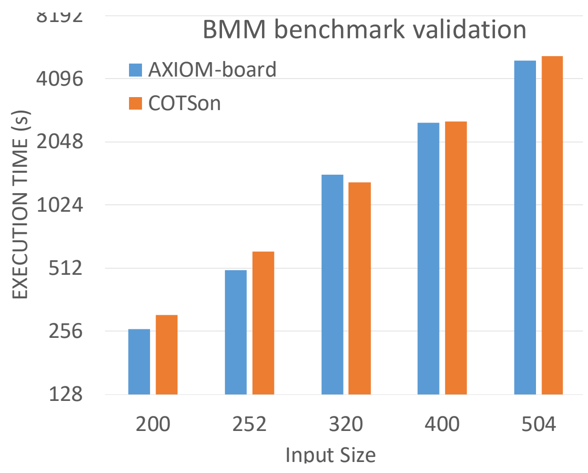
Figure 16 shows our evaluation setup of two AXIOM boards interconnected via USB-C cables, without the need of an external switch. By using synergistically our framework and Vivado tool-chain, we synthesized DF-Thread execution model on Programmable Logic (PL). Table 5 reports resource utilization of the key components of the implemented design on PL in order to perform BMM benchmark across two AXIOM boards.

Table 5 – Resource utilization of the key components of the implemented programmable logic on ZU9EG FPGA (AXIOM board)

Component	LUT	LUTRAM	FF	BRAM	GT
DF-Thread	10.03	1.8	6.01	5.43	-
NIC [4]	41.36	8.96	19.29	15.19	50

5.2. Validating the AXIOM board against the COTSon simulator

An important step in the design is to make sure that the design in the physical board is matching the system that was modeled in the COTSon simulator. As an example, we show in Figure 17 the execution time in the case of the BMM and RADIX-SORT benchmarks, when running on the simulator and on the AXIOM board, while we vary the input data size. The timing are matching closely, thus confirming the validity of our approach. We scaled the inputs in such a way that the number of operations doubles from left to right (input size). On the left (Figure 17), we have the BMM benchmark, where the input size represents the size of the square matrices, which are used in the multiplication. On the right (Figure 17), we have the Radix-Sort benchmark, where the input size represents the size of the list to be sorted.



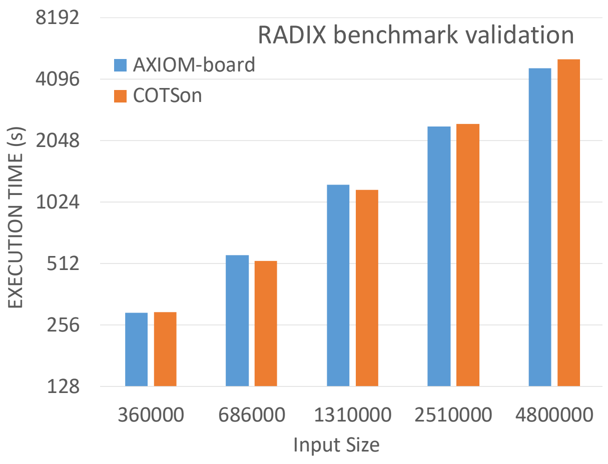


Figure 17 - Validation of the execution time of the simulator against the AXIOM-Board. We used the Blocked Matrix Multiplication (BMM) and Radix-Sort benchmarks with different sizes (weak scaling). The results on the actual board match closely the simulations.

Conclusions

In this paper, we presented our workflow in developing an architecture that could be controlled by the designer in order to match the desired key performance metrics. We found that it is very convenient to use synergistically the Xilinx HLS tools and the COTSon+MYDSE framework in order to select a candidate architecture instead of developing everything just with the HLS tools.

We illustrated the main features of the COTSon simulator and the "MYDSE" tool-set, and we motivated their purpose in our simulation methodology. Thanks to the "functional-directed" approach of the COTSon simulator, we can define the architecture of any architectural components (i.e., a cache) for an early DSE and migrate to HLS only the selected architecture. Our DSE tool-set facilitates the modelling of architectural components in the earlier stages of the design.

We have modified the classical HLS tool-flow, by inserting a modelling phase with an appropriate simulation framework, which can facilitate the architecture definition and reduce significantly the developing time.

We described the simple example of defining a two-way set associative cache through the timing model of COTSon. After, we illustrated the code migration from COTSon to Xilinx HLS tool, showing that the timing description made in the COTSon simulator is conveniently close to the final HLS description of our architecture. However, synthesizing of the HLS description of the cache design in Vivado HLS takes about four hours on a powerful workstation, while we were able to simulate it in COTSon in a few seconds.

By using the workflow presented in this paper, we were able to successfully prototype a preliminary design of our Dataflow programming model (called the DF-Threads) for a reconfigurable hardware platform leading to the AXIOM software/hardware platform, a real system that includes the AXIOM board and a full software stack of more than one million lines of code made available as open-source (https://git.axiom-project.eu/).

References

[1] S. Mittal and J. S. Vetter, “A survey of CPU-GPU heterogeneous computing techniques,” *ACM Comput. Surv.*, vol. 47, no. 4, p. 69, 2015.

[2] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini, “An integrated open framework for heterogeneous MPSoC design space exploration,” in *Proceedings of the conference on design, automation and test in Europe: proceedings*, 2006, pp. 1145–1150.

[3] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, “Heterogeneous chip multiprocessors,” *Computer (Long. Beach. Calif).*, vol. 38, no. 11, pp. 32–38, 2005.

[4] D. Theodoropoulos *et al.*, “The AXIOM platform for next-generation cyber physical systems,” *Microprocess. Microsyst.*, vol. 52, pp. 540–555, 2017.

[5] R. Dimond, S. Racaniere, and O. Pell, “Accelerating large-scale HPC Applications using FPGAs,” in *2011 IEEE 20th Symposium on Computer Arithmetic*, 2011, pp. 191–192.

[6] A. Portero, Z. Yu, and R. Giorgi, “TERAFLUX: Exploiting tera-device computing challenges,” *Procedia Comput. Sci.*, vol. 7, pp. 146–147, 2011.

[7] R. Giorgi *et al.*, “TERAFLUX: Harnessing dataflow in next generation teradevices,” *Microprocess. Microsyst.*, vol. 38, no. 8, pp. 976–990, 2014.

[8] S. Wong *et al.*, “Early Results from ERA—Embedded Reconfigurable Architectures,” in *2011 9th IEEE International Conference on Industrial Informatics*, 2011, pp. 816–822.

[9] S. Wong *et al.*, “ERA--Embedded Reconfigurable Architectures,” in *Reconfigurable Computing*, Springer, 2011, pp. 239–259.

[10] R. Giorgi, “AXIOM: A 64-bit reconfigurable hardware/software platform for scalable embedded computing,” in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, 2017, pp. 1–4.

[11] R. Giorgi, M. Procaccini, and F. Khalili, “AXIOM: A scalable, efficient and reconfigurable embedded platform,” *Des. Autom. Test Eur. Eur. event Electron. Syst. Des. test*, 2019.

[12] D. Theodoropoulos *et al.*, “The AXIOM project (agile, extensible, fast i/o module),” in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015, pp. 262–269.

[13] R. Giorgi, F. Khalili, and M. Procaccini, “Energy efficiency exploration on the zynq ultrascale+,” in *The 30th International Conference on Microelectronics (ICM)*, 2018.

[14] SARC, “http://www.sarc-ip.org.” .

[15] R. Giorgi, Z. Popovic, and N. Puzovic, “Implementing fine/medium grained tlp support in a many-core architecture,” in *International Workshop on Embedded Computer Systems*, 2009, pp. 78–87.

[16] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, “COTSon: infrastructure for full system simulation,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, 2009.

[17] R. Giorgi, M. Procaccini, and F. Khalili, “Analyzing the Impact of Operating System Activity of Different Linux Distributions in a Distributed Environment,” in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 422–429.

[18] Xilinx, “https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.” .

[19] R. Giorgi, M. Procaccini, and F. Khalili, “A Design Space Exploration Tool Set for Future 1K-core High-Performance Computers,” 2019.

[20] R. Giorgi and P. Faraboschi, “An introduction to DF-Threads and their execution model,” in *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, 2014, pp. 60–65.

[21] R. Giorgi, “Exploring dataflow-based thread level parallelism in cyber-physical systems,” in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 295–300.

[22] R. Giorgi, “Scalable embedded computing through reconfigurable hardware: comparing df-threads, cilk, OpenMPI and jump,” *Microprocess. Microsyst.*, vol. 63, pp. 66–74, 2018.

[23] J. Chen, M. Annavaram, and M. Dubois, “SlackSim: A Platform for Parallel Simulations of

CMPs on CMPs," *SIGARCH Comput. Arch. News*, vol. 37, no. 2, pp. 20–29, Jul. 2009.
[24] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer (Long. Beach. Calif).*, vol. 35, no. 2, pp. 59–67, Feb. 2002.
[25] M. M. K. Martin *et al.*, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Comput. Arch. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.
[26] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev, "MPTLsim: A simulator for X86 multicore processors," in *2009 46th ACM/IEEE Design Automation Conference*, 2009, pp. 226–231.
[27] A. Canis *et al.*, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 33–36.
[28] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 270–277.
[29] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004, p. 75.
[30] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–4.
[31] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A high-level synthesis tool for DSP applications," in *High-Level Synthesis*, Springer, 2008, pp. 147–169.
[32] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: Delftworkbench Automated Reconfigurable VHDL Generator," in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 697–701.
[33] ACE CoSy, "http://www.ace.nl." .
[34] Cadence, "https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html." .
[35] Intel, "https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls/pdf." .
[36] Xilinx, " https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html." .
[37] Xilinx, "https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html." .
[38] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A DSL compiler for accelerating image processing pipelines on FPGAs," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, pp. 327–338.
[39] A. Lotfi and R. K. Gupta, "ReHLS: Resource-Aware Program Transformation Workflow for High-Level Synthesis," in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 533–536.
[40] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A Unified Backend for Targeting FPGAs from DSLs," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–8.
[41] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, 2011, pp. 39–48.
[42] A. Portero *et al.*, "Simulating the Future Kilo-x86-64 Core Processors and Their Infrastructure," in *Proceedings of the 45th Annual Simulation Symposium*, 2012, pp. 9:1--9:7.
[43] J. E. Miller *et al.*, "Graphite: A distributed parallel simulator for multicores," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.
[44] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005.
[45] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *ACM SIGARCH computer architecture news*, 2003, vol. 31, no. 2, pp. 84–97.
[46] R. Ierusalimschy, W. Celes, and L. H. de Figueiredo, "The evolution of Lua," 2005.
[47] R. Giorgi, "Exploring future many-core architectures: The TERAFLUX evaluation

framework,” in *Advances in Computers*, vol. 104, Elsevier, 2017, pp. 33–72.
[48] S. Windh *et al.*, “High-level language tools for reconfigurable computing,” *Proc. IEEE*, vol. 103, no. 3, pp. 390–408, 2015.
[49] D. D. Gajski, N. D. Dutt, A. C. H. Wu, and S. Y. L. Lin, *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.
[50] R. Giorgi, N. Bettin, P. Gai, X. Martorell, and A. Rizzo, “AXIOM: a flexible platform for the smart home,” in *Components and Services for IoT Platforms*, Springer, 2017, pp. 57–74.

The author(s) declare(s) that there is no conflict of interest regarding the publication of this paper.