Research Article

Enabling Fast ASIP Design Space Exploration: An FPGA-Based Runtime Reconfigurable Prototyper

Paolo Meloni,¹ Sebastiano Pomata,¹ Giuseppe Tuveri,¹ Simone Secchi,¹ Luigi Raffo,¹ and Menno Lindwer²

¹ *Department of Electrical and Electronic Engineering, University of Cagliari, 09123 Cagliari, Italy*

² *Silicon Hive B.V., High Tech Campus, 5656AE Eindhoven, The Netherlands*

Correspondence should be addressed to Paolo Meloni, paolo.meloni@diee.unica.it

Received 15 September 2011; Revised 22 December 2011; Accepted 22 December 2011

Academic Editor: Lech Jozwiak

Copyright © 2012 Paolo Meloni et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Application Specific Instruction-set Processors (ASIPs) expose to the designer a large number of degrees of freedom. Accurate and rapid simulation tools are needed to explore the design space. To this aim, FPGA-based emulators have recently been proposed as an alternative to pure software cycle-accurate simulator. However, the advantages of on-hardware emulation are reduced by the overhead of the RTL synthesis process that needs to be run for each configuration to be emulated. The work presented in this paper aims at mitigating this overhead, exploiting a form of software-driven platform runtime reconfiguration. We present a complete emulation toolchain that, given a set of candidate ASIP configurations, identifies and builds an overdimensioned architecture capable of being reconfigured via software at runtime, emulating all the design space points under evaluation. The approach has been validated against two different case studies, a filtering kernel and an M-JPEG encoding kernel. Moreover, the presented emulation toolchain couples FPGA emulation with activity-based physical modeling to extract area and power/energy consumption figures. We show how the adoption of the presented toolchain reduces significantly the design space exploration time, while introducing an overhead lower than 10% for the FPGA resources and lower than 0.5% in terms of operating frequency.

1. Introduction

A common feature of modern embedded systems is the need for highly optimized application-specific processing elements. Application-specific instruction-set processors (ASIPs) are often the only solution to the required functional and physical constraints able to provide, at the same time, high flexibility and programmability.

These processors are typically performance- and power-optimized for a specific application domain. The optimizations in terms of extension of the processor instruction-set often include vector processing and SIMD support, complex domain-specific arithmetic operations (e.g., MAC for digital signal processing). In terms of architecture organization, it is not infrequent to find register files with particular configurations (depth, data width, or number of ports), separate local memories for different kinds of application data, customized data channels that implement real-time data flow into and out of the processing units, or synchronization ports shared with other SoC blocks.

As a consequence of such extreme configuration possibilities, to efficiently explore the hardware-software customization of such systems appropriate emulation techniques are required to provide fast but accurate performance estimates. Along with the classical characterization of hardware modules and applications with classic functional metrics (i.e., execution time, cache performance, and resource congestion), there is increasing interest in obtaining early estimations of physical metrics, such as area occupation and power/energy consumption. For all these requirements, hardware-based emulation techniques have been proposed as an alternative, more scalable, solution to cycle-accurate software-based simulation approach.

FPGA devices, with their high flexibility, have shown to serve well for hardware-based emulation but, at the same time, the achievable advantages are mitigated by the overhead introduced by the physical synthesis/implementation flow. This overhead impacts on the emulation time and thus on the number of explorable design space points.

In this work, we address the FPGA-based on-hardware emulation for design space exploration of ASIP processors. We have developed a toolchain that uses software-driven runtime reconfiguration of the emulating platform to enable the evaluation of different architectural configurations after a single synthesis/implementation process, thus maximizing the speed-up of the overall evaluation process. The mechanism employed to reconfigure at runtime the emulating platform does not rely on standard partial reconfiguration capabilities that are offered by current FPGA devices and toolchains. Instead, we developed an algorithm to identify the logic to be placed on the FPGA and the hardware modules that support the actual logic reconfiguration.

The main contribution of this paper can be summarized as follows.

- (i) We conceived an algorithm to identify an overdimensioned platform that can be used to emulate different candidate architectures employing a software-based reconfiguration mechanism;
- (ii) We developed the hardware modules that allow the runtime reconfiguration mechanism to take place;
- (iii) We present the application of the algorithm and toolchain on an industrial ASIP design and programming flow [1];
- (iv) We discuss the advantages and evaluate the resource overhead of the proposed approach with two filtering kernels use cases.

The ultimate result of our work is a toolset capable of automatically building a reconfigurable prototyper that can serve as emulation platform to evaluate a set of candidate design points, selected by a design space exploration (DSE) algorithm. The definition of such algorithm is beyond the scope of this paper.

The remainder of the paper is organized as follows. First, an analysis of the state of the art in architectural emulation is presented in Section 2, while our approach to runtime reconfiguration of the emulation platform is introduced in Section 3. Section 4 defines the ASIP design space considered in this work. Then, in Sections 5 and 6, we explain the baseline of the presented approach and the hardware-software implementation for reconfiguration support. Section 7 presents some experiments and results validating the proposed approach, while Section 8 concludes the work and presents some possible future developments.

2. Related Work

Today, the vast majority of architectural simulation is performed, at maximum accuracy (i.e., cycle level) in software. Among the most famous solutions, many are still sequential, like SimpleScalar [2], Simics [3], Gem5 [4], or MPArm [5]. To cope with the increasing system complexity, parallel software simulators have entered the scene [6]. Some of them give up on pure cycle-level accuracy and look at complex techniques, like event-based simulation, statistical sampling, dynamic accuracy switching, and simulation state roll-back, to model complex, multicore architectures that run full

software stacks [7, 8]. Moreover, specific solutions have been developed for particular classes of processor architectures, and specifically optimized to enable rapid design space exploration of such architectures. For instance, in [9, 10], a software-based simulation and exploration framework targeting optimization of a parametric VLIW microprocessor is presented.

Despite these innovations in the field, there is general consensus on the fact that classic software approaches are not anymore able to provide cycle accuracy for complex multi-core hardware-software designs in reasonable simulation times. A promising alternative approach aims at preserving cycle accuracy, resorting to the extremely high speeds that can be achieved by running the target architecture on some kind of configurable hardware. FPGA devices naturally provide this configurable hardware substrate, potentially enabling hundreds of MHz of pure emulation speed already in the earliest stages of the design flow. In addition, their integration capacity, scalability, the relatively low cost, and the decreasing power consumption figures suggest FPGAs are going to be the reference platform for hardware-based emulation for the next future [11].

The most important contribution to the field of large-hardware FPGA platforms for simulation of complex systems is brought by the RAMP project [12]. Several research activities have been condensed within the scope of this large project, including the first FPGA prototype of a transactional memory chip multiprocessor [13], a thousand-core high-performance system running scientific benchmarks on top of a message-passing communication library [14] and a SPARC-based multithreaded prototype which implemented FPGA-accelerated emulation by introducing the separation between functional and timing simulation [15]. Within the RAMP project, moreover, runtime configuration has been investigated to the extent of some cache-related parameters.

Other examples of hardware-based full-system emulators are [16–19], in which the FPGA-based layer is employed to accelerate the extraction of several metrics of the considered architecture, specified, and automatically instantiated in a modular fashion. These papers also quantify the speedup achievable through FPGA prototyping in three/four orders of magnitude in emulation speed, when compared to software-based simulators. Nevertheless, as mentioned, FPGA-based approaches are still affected by an overhead introduced by the synthesis and implementation flow [20]. With the standard flow, this amount of additional time has to be spent every time a hardware parameter is changed. Several approaches aim at the reduction of the number of necessary synthesis/implementations, by looking at FPGA reconfiguration and programmability capabilities.

In [21], the authors use the FPGA partial reconfiguration capabilities to build a framework for Network-on-Chip-based (NoC) emulation. Also in [22], relying on partial reconfigurability techniques, FPGAs are used to optimize register file size in a soft-core VLIW processor. Both of these solutions implement platform runtime reconfiguration employing specifically designed logic that is increasingly being included in the latest high-end FPGA devices. As opposed to these approaches, we devised a more generally applicable

mechanism, which is more oriented toward software-based reconfiguration and employs minimal hardware modifications to the logic under emulation.

The main use scenario of our toolchain employs it together with a DSE engine, that creates the candidate architectures according to the design requirements, asks for emulation of a set of those and iteratively proceeds in the exploration. Some examples that can benefit from using our approach during the evaluation phase are [23–25].

3. Approach Overview

As already introduced in Section 2, the objective of this work is to enable fast exploration of the processor configuration space, to identify the best customization for a given application. In order to do so, in the context of FPGA-based on-hardware emulation, we aim at minimizing the overhead introduced by the FPGA platform synthesis and implementation process, when different processor configurations have to be prototyped. In fact, the elementary approach to this problem would imply a different FPGA synthesis run for each candidate configuration, impacting significantly on the speed-up over pure software simulation. Instead of doing that, we investigate the possibility of identifying what we call a *worst case* processor configuration (WCC). The WCC is a processor configuration that is overprovided to include all the hardware resources necessary to emulate on FPGA every configuration included in the predefined set of candidates. The size and complexity of the WCC configuration will depend on the kind and number of architectural variations between the different configurations. We then synthesize and implement on the FPGA platform *only* the WCC processor configuration, thus limiting the mentioned time overhead to a single run. After its implementation on the FPGA, we map at runtime each specific configuration onto the implemented WCC, by activating/deactivating hardware subblocks when needed, through dedicated software-based configuration mechanisms. The specific hardware/software mechanisms for runtime reconfiguration that we implemented allow to select the functional blocks included in the WCC processor configuration and to adapt the connections between them. When emulating a configuration on top of a larger set of resources, the interconnection elements configuration is as relevant to the functional correctness as the block selection, since we do not want the WCC configuration to generate incorrect communication latencies due to delays that are not part of the currently emulated configuration.

The algorithm that we conceived to build the WCC configuration, together with the hardware modules that implement the mechanisms for its runtime configuration, were designed to preserve full cycle accuracy. In detail, the number of clock cycles that an arbitrary set of instructions will take to execute on the WCC configuration, when configured to emulate on the FPGA a candidate configuration, has to be exactly the same that it would take on the same candidate configuration, when it is synthesized and emulated alone on the FPGA. Similarly, all the functional metrics (congestion, latency, and CPI) will be exactly the same, when measured in terms of clock cycles. The only difference between the WCC

configuration and each single candidate configuration, when synthesized and placed on the FPGA, will be the number of utilized resources and the operating frequency of the FPGA emulator. Overall, this mechanism preserves the correct functional behavior of the ASIP processor and the binary compatibility of the WCC configuration to the executable code that would run on every candidate topology. The algorithm that we use to synthesize the WCC configuration is described in Section 6.1.

The design flow that implements the proposed prototyping technique refers, as baseline, to the industrial ASIP customization flow. This flow has been extended to provide the needed support for runtime configuration. On the hardware side, we added some further HDL generation capabilities, that have been integrated with the baseline flow and will be explained in Section 6.2. On the software side, we implemented the generation of software functions allowing a user to manage the reconfiguration at application level. The reference baseline flow is described in Section 5 and shown in Figure 2. The extensions, subject of this paper, are presented in Section 6 and in Figure 3.

4. Reference Architecture Template and DSE Strategy

This section will present the ASIP architecture template which we take as reference for our exploration purposes. We also define which variables identify the design space that we intend to explore. The considered processor template belongs to the class of VLIW ASIPs and is composed of instances of industrial IPs, based on a flexible *processor architecture template* (PAT). It employs an automatically retargeting compiler.

Figure 1 shows the main building blocks of the VLIW template. Every processor generated from this template consists of a composition of substructures called *processor slices*, which are complete vertical datapaths that propagate data through the processor template. The processor slices are composed of elementary functional elements called *template building blocks*, such as

- (i) register files: they hold intermediary data between processing operations and are configurable in terms of width; depth; number of read and write ports. Typically, ASIPs generated from this architecture template include many register files;
- (ii) issue slots: they are the basic units of execution within the processor. Every issue slot includes a set of function units (FUs), that implement the operations actually executable and compose the execution stage. From an operating viewpoint, every issue slot receives an operation code from the instruction decoding phase and, accordingly, accesses the register files and activates the proper function units;
- (iii) logical memories: these are containers for hardware implementing memory functionality;
- (iv) interconnect modules: configurable connections automatically instantiated and configured. They implement the required connectivity within the different

FIGURE 1: Reference VLIW ASIP template.

FIGURE 2: Baseline prototyping flow (evaluation time for N candidate architectures can be measured to be approximately N hours for a typical design case like those presented in Section 7).

processor building blocks. In detail, connections can appear on the forward propagation data path along the processor slice, but also on the backward path to implement the writeback into the different register files. The interconnect module from the register files to issue slots is called argument select network (ASN), while the interconnect from the issue slots to the register files is called result selection network (RSN).

In this work, we start from Silicon Hive’s Pearl processor and then enable a design space exploration that covers most of the degrees of freedom exposed by the PAT, but not all of them. We consider every candidate processor configuration as composed of one *control* slice plus some *processing* additional slices providing computational capabilities to the processor. The control processor slice is made of two issue slots, two general-purpose register files, one local memory, and one slave interface for external control. The two issue slots contain a minimal set of function units, which are mainly in charge of managing the program flow (handling the program counter and updating the status register) and the interaction with the program memory. This control logic includes a decoder that generates the opcodes to the function units from the VLIW instructions and a sequencer that handles instruction fetching.

Moreover, we limit the degrees of freedom, limiting the processing slices to have only one issue slot and one register

FIGURE 3: Prototyping flow extended with runtime reconfiguration capabilities (evaluation time for N candidate architectures is approximately 1 hour and $N \times 20$ seconds).

file. This is to say that, in order to have more than one issue slot inside a processor configuration, more than one processing slice is required. Finally, FUs inside every issue slot are taken from a pool of predefined FUs. Although the industrial methodology supports full extensibility of the instruction set through the definition of custom instructions, for the scope of this work, this possibility is not taken into account.

Having defined these limitations on the range of possible template configurations, the design point including only the control slice is the simplest stand-alone processor configuration that can be evaluated. Other design points are processor configurations that instantiate an arbitrary number of additional processing slices and feature different parameterizations of the building blocks included in them. As a result of what was introduced so far, the design space under consideration is thus determined by the following degrees of freedom.

- (i) $N_{IS}(c)$ is the number of issue slots inside the generic configuration c ;
- (ii) $FU_set(x, c)$ is the set of function units in the generic issue slot x , for the configuration c ;
- (iii) $RF_size(x, c)$ is the size (depth) of the register file associated with issue slot x , in configuration c ;
- (iv) $n_mem(c)$ is the number of local memories in configuration c .

Although the way of specifying the parallel datapaths to be instantiated in the processors is limited, the configurability of the template keeps being very wide. With further engineering effort, that is beyond the scope of our research activity, the techniques and the tools that we present can be extended to overcome the mentioned limitations, keeping the correctness of the theoretical approach.

5. The Reference Design Flow

Figure 2 plots the baseline flow for configuration of the ASIP template described in Section 4. The reference toolchain is Silicon Hive's HiveLogic toolchain, composed by a core generator, a system generator and the HiveCC compiler. The figure shows the simplest mechanism to perform exploration of a given design space employing the baseline ASIP configuration flow. Every configuration to be evaluated during the DSE process is described using a proprietary description format. The description customizes the composition of the ASIP architecture under prototyping, in terms of number and kind of blocks, and their connectivity. In the baseline flow, every configuration description is passed to an RTL generator, that analyzes it and provides as output the VHDL hardware description of the whole architecture. This HDL code is then used as input for the FPGA implementation phase, that can be performed with commercial tools.

On the software side, in order to perform the evaluation of the architecture with the desired application, the target source code is compiled by means of an automatically retargetable compiler. The compiler is able to optimize the instruction flow with respect to the instruction set and to the architecture of the processor under prototyping. In order to do so, it extracts information on the underlying ASIP architecture from the same configuration specification provided in input by the DSE engine. The compiler then re-targets itself according to the considered ASIP configuration specification. After compilation, the program can be executed on the ASIP actually implemented on FPGA.

The hardware structures, within the scope of this work, have been instrumented with dedicated activity probes and counters, capable of collecting performance (in terms of numbers of cycles) and the activity figures (i.e., number of accesses) of the blocks in the considered ASIP. In addition to estimation of functional metrics, these activity traces are used to estimate physical metrics, such as dynamic power consumption. This estimation is performed through a layer of analytical modeling that will be described in Section 5.1.

The left side of Figure 2 highlights the time necessary for traversing the entire baseline flow, for each processor configuration. On a workstation equipped with a Core2 Q6850 processor running at 2 GHz, 8 GB of DDR3 RAM memory and running Ubuntu 10 Linux OS with the Xilinx ISE FPGA synthesis/implementation toolchain for a Xilinx Virtex5 LX330 device, we experienced roughly 20 seconds for the platform VHDL generation and code compilation. The largest part of the time was consumed by the FPGA synthesis/implementation toolchain which took about a hour to be completed. This time has been measured with the FPGA device operating far from its resource capacity limit. Therefore, the basic scenario for design space exploration, employing an entire run for each of the N candidate configurations, would require approximately N hours to complete.

5.1. Area and Power/Energy Models. After the execution, the collected emulation data is translated to estimated physical metrics by means of dedicated area; frequency and energy models and fed back to the DSE engine. The translation is performed by means of a set of analytical expressions that allow the evaluation of the energy, area, and critical path contributions of the functional blocks inside the library. Such analytical expressions depend both on their static parameterization and switching activity. The power modeling layer is able to separately account for leakage and dynamic power consumption. Such models refer to a target technology library.

The modeling phase has to acquire information on the architecture configuration, therefore it takes as input a description of the current configuration with a dedicated file generated during the RTL generation phase. Examples of the relevant architectural information are the number, kind, and depth of the different issue slots, number of ports, width and depth of the register files, size of the memories, and so on. The on-hardware emulation then provides the activity traces necessary to perform the analytic estimation of interest.

TABLE 1: Area models dependency recap. Subscripts for operations separate operation count for the single issue slot (IS) from the overall processor count (P).

	Area
FU	$\propto \text{FU_Port_Size}$
IS mux logic	$\propto \#operations_{IS}$
Register File	$\propto \text{RF depth, RF width}$
Decoder	$\propto \#operations_P$
Result select network	$\propto \#operations_P(\#RF \text{ ports} + \#IS \text{ output_ports})$
Sequencer	constant
FIFOs	$\propto \text{Mem_size}$
Program memory	$\propto \text{Mem_size}$
Data memory	$\propto \text{Mem_size}$

TABLE 2: Power models dependency recap. OPC stands for operation per cycle. Program memory is assumed to have 100% access rate.

	Leakage power	Dynamic power
FU	$\propto \text{FU_Port_Size}$	$\propto \text{FU_Port_Size, Access_Rate}$
IS mux logic	$\propto \text{Area}$	$\propto \text{Access_Rate}$
Register file	$\propto \text{Area}$	$\propto \text{RF width, Access_Rate, \#ports}$
Decoder	$\propto \text{Area}$	$\propto \text{OPC}$
Result select network	$\propto \text{Area}$	$\propto \text{Access_Rate, OPC}$
Sequencer	constant	constant
FIFOs	$\propto \text{Mem_size}$	$\propto \text{Mem_size, Access_Rate}$
Program memory	$\propto \text{Mem_size}$	$\propto \text{Mem_size}$
Data memory	$\propto \text{Mem_size}$	$\propto \text{Mem_size, Access_Rate}$

For space reasons, we do not list here all the model formulae for estimation of area occupation, leakage, and dynamic power consumption of every functional block. However, Tables 1 and 2 provide information on how the area and power models account for the different processor blocks. The entries in the tables report the dependency between the related block and the architecture parameters. These parameters are then used in conjunction with technology-dependent normalized values (e.g., per-bit area occupation and per-bit leakage power numbers) to obtain the actual metric estimation of interest. For instance, the second line of Table 1 reports on the area contribution related to multiplexing logic around the FUs inside an issue slot. Such area occupation is proportional to the number of operations actually available inside the issue slot. Similarly, the third line of Table 2 indicates that the leakage power consumed by the same logic depends linearly on the occupied area, while the dynamic power consumption is proportional to the logic access rate. This access rate factor is extracted, within the design flow, by the activity traces obtained through FPGA emulation.

On reading the tables, we point out that the $\#operations_{IS}$ parameter is different from the $\#operations_P$ in that the former refers to the operations executable by a single issue slot, while the latter refers to all the operations simultaneously executable by the entire processor, that is, by all the issue slots. We also underline the difference between the operation count parameters $\#operations_{IS}$, $\#operations_P$, and the current clock tick operation count parameter OPC. The first two parameters are static quantities, depending only on the architecture configuration, and impact on area occupation and static power consumption, while the latter is a dynamic quantity that accounts for the number of currently executing operations and obviously impacts on dynamic power consumption.

The models, that are the formulae used to calculate power and area and whose main dependencies are reported in Tables 1 and 2, were obtained through limited experimentation. As part of the experiments, different processors were synthesized and analyzed in detail, using Synopsys front-end and back-end tools, including lay-out and wireload models. The results of these analyses were used to calculate the detailed normalized per-bit power and area numbers. However, the results are not yet suitable for very wide ranges of parameters. For example, the linear dependency on port width for function unit metrics is not always applicable. To date, the experiments leading to the normalized per-bit power and area numbers involved 16 and 32 bit datapaths. When scaling these datapaths between these numbers and comparing additional detailed synthesis results with results from the models, we found that the total overall accuracy of the formulae remains within 10%.

6. The Proposed Design Flow

To allow fast prototyping of multiple candidate interconnect configurations inside the system, the baseline flow has been extended with a utility that analyzes the whole set of configurations under prototyping, synthesizes the WCC, and creates the configurable hardware and the software functions needed to map each candidate configuration on top of the overdimensioned hardware.

Figure 3 shows the extended flow. By comparison with Figure 2, it can be noticed how many ASIP configuration descriptions can be passed as input to the flow. Based on such input sets, the extended flow identifies the WCC and creates its configuration description, in compliance with the same description format of the reference flow. As a consequence of this modification of the flow, Figure 3 shows how the time necessary to perform the evaluation of N different candidate ASIP configurations is now reduced to roughly $N \times 20 \text{ sec} + 1 \text{ hour}$. Reported times are meaningful examples of the duration of every evaluation step, as can be measured in real design cases. The precise numbers are obviously dependent on the application, on the hardware architectures, and on the system used for the implementation flow.

The synthesis algorithm is described in Section 6.1. The hardware and software support implementation details are respectively provided in Sections 6.2 and 6.3.

6.1. The WCC Synthesis Algorithm. Algorithm 1 is the algorithm used to identify the worst case configuration, for the considered input set of candidate ASIP configurations. In the extended flow, all the design points under test must be provided to the flow at the beginning of the iterative process.

The WCC is defined iteratively while analyzing all the candidate configurations. At every iteration, it is updated according to the design point currently under analysis. At iteration N (i.e., parsing the N th candidate configuration under test c).

- (i) The number of issue slots inside c is identified and compared with previous iterations. A maximum search is performed, then, if needed, the WCC is modified to instantiate $N_{IS}(WCC)$ issue slots. For every issue slot of every candidate configuration c , there must be one and only one corresponding issue slot in the WCC;
- (ii) For every issue slot x inside c , the size of the associated register file is identified and compared with previous iterations. A maximum search is performed, then, if needed, the register file related to the issue slot x inside the WCC is resized to have $RF_size(x, WCC)$ locations. Since there is one and only one issue slot in the WCC that corresponds to the issue slot x of c , the related register file in WCC can be identified without any ambiguity;
- (iii) For every issue slot x inside c , the set of FUs is identified and compared with previous iterations. The issue slot x inside the WCC is modified, if needed, to instantiate a set of FUs being the minimum superset of FUs used in previous configurations.

To clarify the WCC construction algorithm, let us consider a possible design space with the following three different candidate configurations:

- (i) a first candidate configuration with 3 issue slots, respectively instantiating the functional units $\{FU_A, FU_B\}$, $\{FU_A, FU_C, FU_D\}$, and $\{FU_B, FU_C\}$; the register files of sizes 12, 8, and 16 (always expressed in terms of 32 bit registers);
- (ii) a second candidate configuration with 2 issue slots, respectively instantiating the functional units $\{FU_A, FU_B, FU_D\}$, and $\{FU_A, FU_B, FU_C\}$; the register files of sizes 24 and 16;
- (iii) a third candidate configuration with 4 issue slots, respectively instantiating the functional units $\{FU_C\}$, $\{FU_A, FU_B, FU_C, FU_E\}$, $\{FU_C, FU_D, FU_E\}$, and $\{FU_A, FU_B, FU_C\}$; the register files of sizes 8, 32, 24, and 16.

Decomposing the outer loop of the algorithm, the WCC is iteratively constructed as follows.

- (1) The WCC instantiates 3 issue slots, with functional units $\{FU_A, FU_B\}$, $\{FU_A, FU_C, FU_D\}$, and $\{FU_B, FU_C\}$; register file of sizes 12, 8, and 16, respectively;

```

Input: A set of  $K$  candidate ASIP configurations
Output: Worst Case Configuration (WCC)
 $N_{IS}(WCC) = 1;$ 
 $RF\_size(1, WCC) = 0;$ 
 $FU\_set(1, WCC) = \{\};$ 
for each candidate configuration  $c$  with  $c = 1, \dots, K$  do
     $N_{IS}(WCC) = \max\{N_{IS}(i)\};$  with  $i = 1, \dots, c$ 
    for each issue slot  $x$  of the configuration  $c$  do
         $RF\_size(x, WCC) = \max\{RF\_size(x, i)\};$  with  $i = 1, \dots, c$ 
         $FU\_set(x, WCC) = FU\_set(x, c) \cup FU\_set(x, i);$  with  $i = 1, \dots, c$ 
    end
end

```

ALGORITHM 1: Worst case configuration identification.

- (2) The WCC is then modified to instantiate 3 issue slots, with functional units $\{FU_A, FU_B, FU_D\}$, $\{FU_A, FU_B, FU_C, FU_D\}$, and $\{FU_B, FU_C\}$; register file of sizes 24, 16, and 16, respectively. It can be noticed how the functional unit sets are being populated according to the union of the candidate configurations sets;
- (3) Finally, the WCC is modified to instantiate 4 issue slots, with functional units $\{FU_A, FU_B, FU_C, FU_D\}$, $\{FU_A, FU_B, FU_C, FU_D, FU_E\}$, $\{FU_B, FU_C, FU_D, FU_E\}$, and $\{FU_A, FU_B, FU_C\}$; register file of sizes 24, 32, 24, and 16, respectively.

6.2. Hardware Support for Runtime Reconfiguration. The software runtime reconfiguration capability is supported by two hardware modules, automatically generated and instantiated in the overdimensioned WCC architecture basing on the set of different input configurations that are passed to the exploration engine.

The first module is the *instruction adapter*, a programmable decoder that interprets and delivers every single chunk of the VLIW instruction to the relevant hardware element. For each candidate architecture in input, knowing the complete set of architecture parameters, the instruction bits can be split in subranges that identify specific control directives to the datapath. Examples of such bit ranges are operation codes (that activate specific function units and specific operations inside the issue slots), index values (used to address the locations to be accessed in the register files), and configuration patterns (used to control the connectivity matrices that regulate the propagation of the computing data through the datapath). The width and the position of the boundaries between the bit ranges are not fixed but instead depend on the architectural configuration that must execute the instruction.

The configurable instruction adapter is in charge of translating the instructions produced by the compiler, which retargets itself for each candidate ASIP configuration, into an instruction executable on the WCC. All the sequences in the instruction related to a given slice of the configuration under evaluation are adapted in size and dispatched to the corresponding slice of the WCC. The value of each control

directive is modified to ensure the instruction will provide the correct functionality on the overdimensioned prototype, despite the presence of additional hardware. All slices that do not exist in the configuration under test are disabled using dedicated opcodes.

Figure 4 shows an example of how the instruction adapter works. In the example, an instruction produced by the compiler for a configuration under test c requires the activation of the FU in charge of performing shift operations (*shu*) in IS1. Inside the candidate configuration c alone, the instruction decoder would statically split the VLIW instructions as it is stored in the program memory into different opcodes and pass each of them to the proper issue slot. Inside the issue slot IS1, only the *shu* function unit would then be activated, basing on the opcode value.

In the extended flow, where the number of issue slots is potentially different (more issue slots are usually instantiated in the WCC) from the one of each candidate configuration, the instruction adaptation is necessary to execute the same instruction binary on the WCC. The adapter is adequately programmed via software through a memory-mapped register write, in order to obtain information on the configuration identifier. According to this value, it then decodes the different instruction fields, generates a new (longer) instruction word, and dispatches the new opcodes according to the mapping strategy. In the example of Figure 4, IS1 is mapped onto IS m in the WCC. The opcode originally targeted for IS1 is thus dispatched to IS m . Its value is translated to activate the *shu*, taking into account the architectural composition of IS m in terms of its worst case function units. Since the opcode values may differ from each candidate configuration to the WCC, the opcode width is adapted to the WCC architecture. Similar dispatching/translation is applied by the adapter to the other instruction fields.

The second hardware module, the *memory router*, is introduced in order to support different connections between the pool of data memories instantiated in the WCC and the issue slots. The baseline flow supports, directly inside the application code, the explicit positioning of variable and data structures in each memory inside the architecture. To keep this capability in the extended flow and to allow at the same time the possibility of arbitrarily dispatching

FIGURE 4: Example of instruction adapting.

the operations to the issue slots inside the WCC, the memory router provides connectivity between memories and issue slots to be programmable according to the configuration under prototyping. The programmability mechanism follows the same logic of the opcode dispatching process implemented inside the instruction adapter that we previously mentioned. An identifier is stored, for each candidate configuration, inside a memory-mapped register. Its value drives the connections to the different memory modules.

6.3. Software Support for Runtime Reconfiguration. Software support for reconfiguration is realized simply writing a memory-mapped register, which stores a unique configuration identifier and acts as an architecture selector, directly accessible by a function call at C application level. The automatic flow provides, in the form of a simple API, the function that accesses this register. The value stored in the register, as already described in Section 6.2 controls the instruction adapter and the memory router, to select one among the candidate configurations under emulation. The generated routines are suitable to be compiled and linked by the application executable file running on an host processor controlling the ASIP.

In case the user does not modify the application code to instrument it with the function calls that select the architecture configuration under test, the extended framework provides an alternative way, which employs Xilinx System Generator (SysGen) [26]. The utility was used to enable direct access from a host workstation to the configuration selector and to allow easy access to the emulation results. SysGen consists of a set of Matlab/Simulink blocks and routines that enable, among other features, selecting a hardware system, implemented on an FPGA device, as if it was a Matlab/Simulink block, and allow the cooperation of software-based simulation and on-hardware prototyping.

SysGen provides the capability of automatically creating the hardware and software support for data exchange between the FPGA board and a host processor. It also enables stimuli generated by a software-based simulation environment (such as a Matlab function itself or an HDL simulated stimuli generator like ModelSim) to be fed as input to the hardware. In this way, after every execution, the user can choose the next configuration to evaluate directly from the Simulink graphic interface and automatically restart the application.

6.4. Hardware Implementation Degradation and Overhead Reduction Techniques. As we specified in Section 3 and in the definition of the instruction adapter and memory router in Section 6.2, the modules were designed to not introduce any spurious latency cycle in the WCC with respect to the single configurations. For this reason, cycle accuracy is always guaranteed. However, as a consequence of the provision of runtime reconfiguration capabilities, a degradation of the quality of results may be expected with respect to the hardware implementation of a single sample configuration on the FPGA device. In particular, two aspects related to the implementation quality can be affected and potentially preclude the usability of the approach. Firstly, the FPGA area occupation of the WCC determines whether the prototyping platform fits on one given target programmable device or not. Secondly, in case the additional hardware (instruction adapter and memory router) affects the critical path of the candidate system, the WCC working frequency might have an impact on the emulation time.

These possible drawbacks have been evaluated in detail while implementing the flow extensions. In order to minimize the area occupation of the WCC netlist, the minimum needed subset of hardware resources is identified by the WCC synthesis algorithm. The mapping of the instruction fields inside the instruction adapter is optimized to reduce as much

as possible the number of different adaptations that must be allowed by the circuitry, to minimize its area and, being the adapter a combinational module, its propagation delay to prevent impacting the overall working frequency of the prototype. The resulting area/frequency overhead is quantified in Section 7.3. We report there the experimental results and discuss how the mentioned overhead can be effectively controlled and how the proposed approach is applicable to systems characterized by considerable complexity.

7. Use Cases

In this section, we present two use case scenarios for the previously described runtime reconfiguration techniques. The first use case involves exploration of a possible configuration space for the architecture of a single ASIP, running an image filtering kernel application. The second use case involves exploration of a multi-ASIP system, composed of three processors, a packet-based on-chip interconnection switch, two different shared modules (a UART controller and a hardware Test&Set semaphore bank for lock-based synchronization) and a MicroBlaze platform control processor. The running application is an MJPEG codec, partitioned and mapped on the three ASIP processing elements. For both sets of experiments, the adopted hardware FPGA-based platform features a Xilinx Virtex5 XC5VLX330 device, counting over 2M equivalent gates.

7.1. Single-ASIP Exploration. We will present the results obtained while performing the single-ASIP architecture selection process over a set of 30 different ASIP configurations. The explored design points were identified considering different permutations of the following processor architectural parameter values:

- (i) $N_{IS}(c)$: 2 or 3 and 4 or 5;
- (ii) $FU_set(x, c)$: from 3 to 10 FUs per issue slot;
- (iii) $RF_size(x, c)$: 8 or 16 or 32 entries, each 32 bits wide;
- (iv) $n_mem(c)$: 2 or 3 or 4 or 5.

A filtering kernel was compiled for every candidate configuration, and the resulting binaries were executed on the WCC prototype, adequately reconfigured. Although the above-mentioned design space could seem small, this design case is very realistic. In fact several DSE engines, like [25], when selecting the best system-level configuration over millions of design points, often starts by exploring the system-level composition (number and kind of cores in the system) or the interconnection topology and application task mapping. This first space design space pruning is usually performed using a tuned high-level software simulator, which is very fast but not capable of detecting the precise functional and physical behavior of the microarchitectural modules. To be effective, the simulator is usually fed with latency and power numbers related to the execution of an application on a given processor instance. Such numbers are typically obtained from a single-processor detailed prototyping. This use case does not only allow to assess the feasibility of our

approach, but also presents an example of such kind of analysis.

In Figure 5, we show the results of the evaluation obtained with respect to total execution time, total latency, total energy, and power dissipation. The energy results have been calculated assuming an identical clock frequency for all the considered ASIP configurations. In theory, energy could be modeled only after an operating frequency estimation step is performed. The reason behind this is obviously the different critical paths that might appear in different configurations. However, since in this use case our logic synthesis results identified the critical path in the same piece of logic for all the ASIP configurations, the operating frequency can be assumed to be the same without introducing any error.

In this paper, to avoid the disclosure of sensitive industrial information, we do not report absolute power, frequency, and area numbers. We backannotate the emulation results referring to “comparative” numbers for energy and area contributions for the functional blocks in the ASIPs. The area, power, and energy figures are thus reported respectively in $R\mu m^2$ (relative square microns), $R\mu W$ (relative microwatts), and $R\mu J$ (relative microjoules), while, for the execution time, we report the number of cycles. The use of such relative units does not hide the usefulness of the performed analysis for a prospective designer approaching comparative architecture selection. Multiconstraint optimization can be effectively performed. For example, imposing a constraint on maximum execution time (e.g., 200 K cycles), the user could identify a subset of candidates satisfying the constraint (configurations $\#\{0, 5, 8, 9, 12, 16, 19, 24\}$). Then, among these, one could choose the best configuration with respect to power or area (#24).

From the performed analysis a designer could estimate *topology*₂₉ to be the joint optimal configuration, for the considered target application, from the points of view of energy consumption, execution time, and area occupation. To identify possible computation bottlenecks and power hot-spots inside the architecture, performance, and power profiling at the functional unit level can also be obtained, referring to each single functional unit included in the configurations under test. As an example, we show in Figure 6 a plot reporting power consumption of each function unit in a particular configuration, during the execution of the already mentioned filtering kernel binaries.

The cycle-accurate correctness of the emulation of a candidate configuration with the WCC one is guaranteed by construction of the WCC architecture. In fact, every instruction that traverses the ASIP datapath, both in the candidate configuration and in the WCC, undergoes the same exact logic path. The WCC architecture does not insert any new pipeline stage in the instruction path with respect to the ASIP. What can change is only the operating frequency of the WCC and thus the resulting execution real time, due to the more complicated combinatorial logic (e.g., instruction adapter), but the emulated CPI will be exactly the same (as a count of clock cycles). To confirm this behavior, we compared the experiment results obtained by prototyping all the candidate configurations on the WCC architecture with the results of their stand-alone evaluation. As expected, we

FIGURE 5: Use case results. Every configuration is labeled with a 4-tuple, whose elements represent the total number of issue slots, the register file capacity (in 32 bit words), the number of fully featured issue slots, the number of data memories. Execution cycles are reported for the different configurations under emulation. Moreover, the modeled power consumption (expressed in $R\mu W$), area occupation (expressed in $R\mu m^2$), and total energy consumption (expressed in $R\mu J$) figures are reported.

found that exactly the same “functional-related” (execution time, latency, and switching activity) performances are estimated. Cycle/signal level accuracy can thus be assessed for the presented approach.

In order to evaluate the speed-up that was allowed by the proposed approach, we should firstly consider the time needed for the same design exploration performed using the classic approach. This requires the designer to go through the implementation flow for all the candidate design points. The related time depends on the complexity of the considered design point. For the set of candidates in this example, implementation time ranged from roughly 20 minutes to 45 minutes, for a total of 15 hours. When using our approach, on the other hand, only one synthesis is required. Such synthesis of the WCC required roughly 45 minutes, allowing

for a 20x speed-up. In this analysis we are not accounting for the time needed for the execution of the kernels on the prototypes, since it is negligible (few seconds) with respect to the implementation time.

7.2. Multi-ASIP Configuration. We present now a second use case, which validates the proposed emulation approach inside the exploration of a multi-ASIP system. Several are the reasons for which such kind of use case is important. Firstly, since cycle-accurate simulation becomes slower as the size of the simulated system increases, obtaining execution traces of multicore systems by means of software-based methods quickly becomes unpractical. As opposed to software-based simulators, the emulation speed achievable with FPGA devices is not affected, in general, by the system size. The only

FIGURE 6: Power consumption for each FU in a particular configuration, composed of four issue slots, with register files of 16 entries, reported in $R\mu W$.

requirement is that the system under prototyping fits in the target configurable device. Therefore, a multi-ASIP system exploration should, in terms of overall speed-up, favor FPGA-based emulation approaches as opposed to pure software simulation.

Moreover, the use case shows that it is possible to cross-optimize the microarchitectures of the ASIPs, exploiting results obtained from a complete system prototyping. In fact, we show how the designer is able to observe the mutual influence among the processors and between the processors and the surrounding environment (interconnect infrastructure, peripherals, and shared memories) without relying on further software-based simulation steps.

We present the prototyping results obtained by the execution of an M-JPEG encoder on a parallel MPSoC composed of a host processor and three ASIPs, interconnected by means of a Network-on-Chip subsystem. The system is represented in Figure 7.

The application is partitioned into four parallel computing tasks, communicating through FIFO channels, according to a programming model based on Kahn Process Networks ([27]). In detail:

- (i) the host processor is in charge of initializing the program and data memories inside the ASIPs and executing a parallel task (named *Video_in*). The *Video_in* task dispatches the input stream pixels, the header and footer information to the other tasks;
- (ii) the second task then involves the DCT-encoding calculations and is mapped on the first ASIP (ASIP1);
- (iii) the third task takes care of the quantization process (Q in short) and is mapped on the second ASIP (ASIP2);
- (iv) the fourth and last task performs the variable length encoding part (VLE) and is executed by the third ASIP (ASIP3).

In order to explore the microarchitectures of the single ASIPs, ASIP1 and ASIP2 were enriched with the support for

fast prototyping. ASIP3 on the contrary has been implemented as a single static configuration. By doing so, we are able to investigate on the impact that the customization of ASIP1 and ASIP2 has on the metrics related with the execution of the VLE task, which runs on ASIP3.

In detail, in the presented results the number of issue slots and the kind of included function units inside ASIP1 and ASIP2 were the variables that defined the exploration space. We let ASIP1 and ASIP2 processors issue slot counts assume all the possible combinations of values from 2 to 4.

Figure 8 plots the results of the exploration in terms of execution times (measured in clock ticks) and IPC per ASIP. The execution times are probed on the ASIP3 processor. Considering the typical communication pattern of an M-JPEG encoder and considering the task mapping explained in Figure 7, it is clear how the ASIP3 will have to wait for the ASIP1 and ASIP2 to complete their assigned tasks (DCT and Q) to complete its own task. In fact, this is a classical example of dataflow communication pattern; therefore, the execution time of ASIP3 fully characterizes the overall application execution time.

The power results were instead acquired as a sum of the power dissipation occurring in the ASIP1 and ASIP2 processors, for their different configurations. ASIP3 was not considered in the power estimation process, since its architectural configuration has been kept constant.

By looking only at the execution time results, it seems that the architectural modifications performed inside the ASIP2 processor (which is assigned the quantization task) were unable to produce an impact on the ASIP3 execution time. This means that, as for what regards the execution time, the DCT task is the most hungry, and optimizing its execution is key to obtaining an overall performance improvement.

Interestingly, changing the number of issue slots from 2 to 3 does not affect the execution time. We could argue that this is related to how the VLIW compiler exploits the available issue slots to exploit the DCT task available parallelism.

We are also able to provide an estimation of the system power and energy consumption. Figure 9 shows the energy (measured in $R\mu J$) and power consumption (measured in $R\mu W$) figures. It is important to mention that, as already explained in Section 7.1, the energy estimation has been carried out without accounting for the real operating frequency of the processors. This is still a reasonable assumption, since the most relevant piece of logic is included in all the processor configurations and limits the critical path, regardless of the architectural modifications that we made. We are also assuming that the entire processing section of the system runs within a single-frequency domain.

By looking at the power numbers, on the other hand, we see that the architectural changes to both the ASIP1 and ASIP2 processor configurations have an impact on the overall results. The numbers suggest that the most convenient architectural configuration, in terms of energy, features 2 issue slots in the ASIP1 processor and 4 in the ASIP2 processor.

In the use case exploration, we kept the NoC switch configuration constant and changed the configurations of ASIP1 and ASIP2. This means that the candidate configurations

FIGURE 7: Multi-ASIP platform under exploration.

with the highest number of issue slots could have experienced more network congestion than the ones with less issue slots, simply because of the increased memory references issued per cycle. Since this difference exists among the various candidate configurations, the WCC architecture exactly reproduces it when configured to emulate the different candidate configurations. This is another consequence of the cycle accuracy that the WCC architecture has with respect to the direct emulation of each single candidate configuration.

Both the functional results and the cycle counts obtained with our FPGA approach and the baseline software simulation were completely equivalent. But, while cycle-accurate software simulation required few minutes (roughly five on average per configuration), onboard execution on the FPGA prototype required only few seconds (roughly two) to emulate each candidate architecture. A synthesis/implementation flow, performed on an Intel Quad-Core machine with commercial tools, required less than half an hour to be completed. Such time obviously depends on the size of the system, but can be estimated in the order of one hour for moderately complex systems. According to the mentioned numbers, the presented approach allows time saving that

increases with the number of candidate topologies under prototyping, outperforming soon (for approximately ten candidate design points involved in the design process) software-based simulation. Moreover, the software-based simulation is not always effective. For example, if design cases imply the evaluation of runtime middleware policies or network routing protocols, whose effectiveness must be measured on execution times much longer than a single-processing kernel, simulation times become unaffordable, making FPGA emulation the only available evaluation method and our approach fundamental for its application to DSE.

7.3. Hardware Overhead due to Runtime Configurability.

In this section, we present a quantitative analysis on the overhead introduced by the addition of reconfiguration support in terms of occupied hardware resources and critical path degradation. For this scope, referring to the previously presented use case, we compare the results obtained by a synthesis of the WCC and the most hardware hungry configuration under test (when implemented for a stand-alone evaluation, without support for runtime reconfiguration). Again, the WCC is generated to support 30 different

(a) Execution times

(b) IPC per ASIP

FIGURE 8: Second use case results. Every horizontal axis captures the number of issue slots inside processors ASIP1 and ASIP2. Execution cycles are reported for the different configurations under emulation. The IPC for every ASIP is also reported.

configurations. This degree of exploration is reasonably interesting for this kind of applications, though larger pools of configurations can always be used.

As can be noticed from Table 3, the introduced overhead in FPGA resource utilization is limited, also considering that WCC is built iteratively from a set of 30 different candidates. Considering that we are comparing the WCC to the most power hungry of the 30 input ASIP configurations, we would expect the comparison result to be more related to the overhead introduced by the logic necessary for the reconfiguration mechanism (i.e., instruction adapter and memory router modules) than to the overprovision of actual architectural functional blocks (i.e., issue slots and function units). Table 3 confirms that, in terms of FPGA resources, this overhead is limited to 10% of the largest candidate ASIP configuration. Moreover, since the memory router and instruction adapter are mostly combinational modules, the overhead is almost entirely consumed by the look-up table (LUT) FPGA slice logic.

Also, a comparison related to the logic synthesis maximum operating frequency is presented, being this a limiting

(a) Power consumption

(b) Energy consumption

FIGURE 9: Second use case results. Every horizontal axis captures the number of issue slots inside processors ASIP1 and ASIP2. Execution cycles are reported for the different configurations under emulation. The modeled power consumption (expressed in $R\mu W$) and energy consumption (expressed in $R\mu J$) figures are reported. Values are expressed as offset with respect to a zero-point (lowest value of power and energy consumption).

factor for the overall emulation speed. Results presented in Table 4 show that critical path is almost not impacted (less than 0.1%) by the presence of the hardware structures implementing reconfiguration support. This result is particularly relevant, since the instruction adapter and memory router are made out of almost completely combinational logic; therefore, their potential impact on the critical path was high. However, after the insertion of such modules, the critical path still resides inside more complex function units (e.g., multiply and accumulate) and is not affected by the inserted logic. The minimal increase is due to unpredictable behaviors of the synthesis algorithm.

TABLE 3: FPGA hardware overhead figures.

	Occupied slices	Slice registers	Slice LUTs
Largest configuration	19859	6923	16387
WCC	21278 (+7.1%)	6931 (+0.001%)	17951 (+9.5%)

TABLE 4: FPGA critical path overhead figures.

	Critical path
Largest configuration	9.809 ns
WCC	9.817 ns

The results show how the overhead reduction mechanisms explained in Section 6.4 effectively allow to perform prototyping of a reasonable number of different configurations without significant emulation time degradation.

8. Conclusions

In this work, an approach to ASIP configuration selection, based on FPGA-based emulation platforms, is presented and evaluated. The main point of strength of the proposed approach is that, by using the runtime software-based reconfiguration capabilities of the hardware platform, several emulation steps could be performed after a single-FPGA synthesis and implementation run. In such a way, we showed how different VLIW ASIP architectures could be emulated on hardware by *mapping* them via software on a larger *worst case* configuration.

In addition to functional correctness of the emulation approach, the experimental data proved that the overhead introduced by the overprovision of hardware resources to the worst case configuration that is actually implemented on hardware does not preclude the feasibility of the approach, neither in terms of FPGA slices (less than 10% overhead) nor in terms of critical path and operating frequency (negligible overhead). In addition to the classical functional metrics (e.g., execution time, access rate, IPC, and resource congestion), the presented framework has been proven to be able to produce physical metrics (e.g., area obstruction, leakage static power, dynamic power, and energy consumption) for a prospective implementation of the ASIP system. Moreover, the reconfiguration mechanism for configuration selection is compatible with a preexisting wider framework for FPGA emulation of ASIP based multicore architectures. The presented use case validates the usefulness of the framework as an effective support to quantitative design space exploration or simply as an environment for rapid prototyping of complex multicore platforms.

Future developments of the proposed software-based reconfiguration approach include an accurate comparison, in terms of reconfiguration speed and resource overhead, to proprietary (Xilinx) tools that exploit newest devices hardware partial FPGA reconfiguration capability. On the modeling side, we intend to tune the models with further experiments. Also, the models for the function units need to be

adapted depending on the type of the module. For example, it is likely quadratic models work better for units which involve multipliers. However, this may not be the case for adder units.

Acknowledgments

The research leading to these results has received funding from the European Community Seventh Framework Programme (FP7/2007–2013) under Grant agreement no. 248424, MADNESS Project, from ARTEMIS JU—ASAM Project, and by the Region of Sardinia, Young Researchers Grant, PO Sardegna FSE 2007–2013, L.R.7/2007 “Promotion of the scientific research and technological innovation in Sardinia.”

References

- [1] SiliconHive, *Hivelogic Configurable Parallel Processing Platform*, SiliconHive, 2010.
- [2] T. Austin, E. Larson, and D. Ernest, “SimpleScalar: an infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 12–67, 2002.
- [3] P. S. Magnusson, M. Christensson, J. Eskilson et al., “Simics: a full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 12–58, 2002.
- [4] N. Binkert, B. Beckmann, G. Black et al., “The gem5 simulator,” *Computer Architecture News—SIGARCH*, vol. 39, pp. 1–7, 2011.
- [5] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, “MPARM: exploring the multi-processor SoC design space with systemC,” *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 41, no. 2, pp. 169–182, 2005.
- [6] J. E. Miller, H. Kasture, G. Kurian et al., “Graphite: a distributed parallel simulator for multicores,” in *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA ’10)*, pp. 1–12, January 2010.
- [7] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, “COTSon: infrastructure for full system simulation,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61.
- [8] A. Falcon, P. Faraboschi, and D. Ortega, “Combining simulation and virtualization through dynamic sampling,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS ’07)*, pp. 72–83, April 2007.
- [9] G. Ascia, V. Catania, M. Palesi, and D. Patti, “A system-level framework for evaluating area/performance/power trade-offs of VLIW-based embedded systems,” in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC ’05)*, vol. 2, pp. 940–943, January 2005.
- [10] A. G. Di Nuovo, M. Palesi, D. Patti, G. Ascia, and V. Catania, “Fuzzy decision making in embedded system design,” in *Proceedings of the 4th International Conference on Hardware Software Codesign and System Synthesis (CODES+ISSS ’06)*, pp. 223–228, ACM, October 2006.
- [11] K. D. Underwood and K. S. Hemmert, “Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance,” in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM ’04)*, pp. 219–228, April 2004.

- [12] J. Wawrzyniek, D. Patterson, M. Oskin et al., "RAMP: research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.
- [13] S. Wee, J. Casper, N. Njoroge et al., "A practical FPGA-based framework for novel CMP research," in *Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '07)*, pp. 116–125, ACM, February 2007.
- [14] A. Krasnov, A. Schultz, J. Wawrzyniek, G. Gibeling, and P. Y. Droz, "RAMP Blue: a message-passing manycore system in FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 27–29, August 2007.
- [15] Z. Tan, A. Waterman, R. Avizienis et al., "RAMP Gold: an FPGA-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference (DAC '10)*, pp. 463–468, usa, June 2010.
- [16] P. Del Valle, D. Atienza, I. Magan et al., "Architectural exploration of MPSoC designs based on an FPGA emulation framework," in *Proceedings of the Design of Circuits and Integrated Systems Conference (DCIS '06)*, pp. 12–18, Barcelona, Spain, November 2006.
- [17] N. Genko, D. Atienza, G. De Micheli et al., "A novel approach for network on chip emulation," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '05)*, vol. 2, pp. 2365–2368, May 2005.
- [18] D. Chiou, D. Sunwoo, J. Kim et al., "FPGA-accelerated simulation technologies (FAST): fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO '07)*, pp. 249–261, December 2007.
- [19] S. Lukovic and L. Fiorin, "An automated design flow for NoC-based MPSoCs on FPGA," in *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping—Shortening the Path from Specification to Prototype (RSP '08)*, pp. 58–64, IEEE Computer Society, Washington, DC, USA, 2008.
- [20] P. Meloni, S. Secchi, and L. Raffo, "An FPGA-based framework for technology-aware prototyping of multicore embedded architectures," *IEEE Embedded Systems Letters*, vol. 2, no. 1, pp. 5–9, 2010.
- [21] Y. E. Krasteva, F. Criado, E. d. l. Torre, and T. Riesgo, "A fast emulation-based NoC prototyping framework," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (RECONFIG '08)*, pp. 211–216, IEEE Computer Society, Washington, DC, USA, 2008.
- [22] S. Wong, F. Anjam, and F. Nadeem, "Dynamically reconfigurable register file for a softcore vliw processor," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE '10)*, pp. 969–972, March 2010.
- [23] G. Palermo, C. Silvano, and V. Zaccaria, "Multi-objective design space exploration of embedded systems," *Journal of Embedded Computing*, vol. 1, pp. 305–316, 2005.
- [24] T. Givargis, F. Vahid, and J. Henkel, "System-level exploration for pareto optimal configurations in parameterized systems-on-a-chip," in *Proceedings of the IEEE/ACM international conference on Computer-aided design (ICCAD '01)*, pp. 25–30, IEEE Press, Piscataway, NJ, USA, 2001.
- [25] J. E. Coffland and A. D. Pimentel, "A software framework for efficient system-level performance evaluation of embedded systems," in *Proceedings of the ACM Symposium on Applied Computing (SAC '03)*, pp. 666–671, ACM, New York, NY, USA, March 2003.
- [26] Xilinx, "System Generator for DSP," http://www.xilinx.com/support/sw_manuals/sysgen_gs.pdf.
- [27] E. Cannella, O. Derin, and T. Stefanov, "Middleware approaches for adaptivity of kahn process networks on networks-on-chip," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP '11)*, pp. 1–8, Tampere, Finland, November 2011.