Agile SoC Development with Open ESP

Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman,
Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni*

{paolo,davide_giri,giuseppe,piccolboni,jzuck,cota,petracca,pilato,luca}@cs.columbia.edu

Department of Computer Science, Columbia University in the City of New York, New York, NY 10027

ABSTRACT

ESP is an open-source research platform for heterogeneous SoC design. The platform combines a modular tile-based architecture with a variety of application-oriented flows for the design and optimization of accelerators. The ESP architecture is highly scalable and strikes a balance between regularity and specialization. The companion methodology raises the level of abstraction to system-level design and enables an automated flow from software and hardware development to full-system prototyping on FPGA. For application developers, ESP offers domain-specific automated solutions to synthesize new accelerators for their software and to map complex workloads onto the SoC architecture. For hardware engineers, ESP offers automated solutions to integrate their accelerator designs into the complete SoC. Conceived as a heterogeneous integration platform and tested through years of teaching at Columbia University, ESP supports the open-source hardware community by providing a flexible platform for agile SoC development.

KEYWORDS

System-level design, SoC, accelerators, network-on-chip.

1 INTRODUCTION

Why ESP? ESP is an open-source research platform for heterogeneous system-on-chip (SoC) design and programming [18]. ESP is the result of nine years of research and teaching at Columbia University [11, 12]. Our research was and is motivated by the consideration that Information Technology has entered the age of heterogeneous computing. From embedded devices at the edge of the cloud to data center blades at the core of the cloud, specialized hardware accelerators are increasingly employed to achieve energy-efficient performance [9, 15, 31]. Across a variety of application domains, such as mobile electronics, automotive, natural-language processing, graph analytics and more, computing systems rely on highly heterogeneous SoC architectures. These architectures combine general-purpose processors with a variety of accelerators specialized for tasks like image processing, speech recognition, radio communication and graphics [20] as well as special-purpose processor cores with custom instruction sets, graphics processing units, and tensor manipulation units [32]. The shift of the silicon industry from homogeneous multicore processors to heterogeneous SoCs is particularly noticeable if one looks at the portion of chip area dedicated to accelerators in subsequent generations of state-of-the-art chips for smartphones [48], or at the amount of diverse processing elements in chips for autonomous driving [16].

ESP Vision. ESP is a platform, i.e., the combination of an architecture and a methodology [11]. The methodology embodies a set
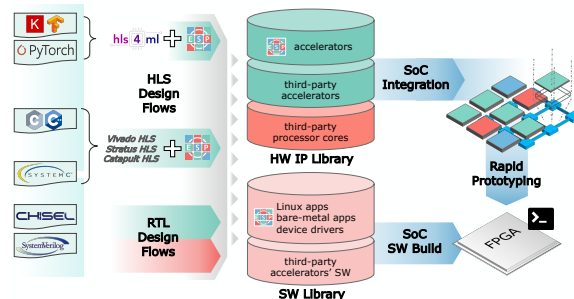

Figure 1: Agile SoC design and integration flows in ESP.

of agile SoC design and integration flows, as shown in Figure 1. The ESP vision is to allow application domain experts to design SoCs. Currently, ESP allows SoC architects to rapidly implement FPGA-based prototypes of complex SoCs. The ESP *scalable architecture* and its *flexible methodology* enable a seamless integration of third-party open-source hardware (OSH) components (e.g., the Ariane RISC-V core [1, 51] or the NVIDIA Deep-Learning Accelerator [3]). SoC architects can instantiate also accelerators that are developed with one of the many design flows and languages supported by ESP. The list, which continues to grow, currently includes: C/C++ with Xilinx Vivado HLS and Mentor Catapult HLS; SystemC with Cadence Stratus HLS; Keras TensorFlow, PyTorch and ONNX with hls4ml; and Chisel, SystemVerilog, and VHDL for register-transfer level (RTL) design. Hence, accelerator designers can choose the abstraction level and specification language that are most suitable for their coding skills and the target computation kernels. These design flows enable the creation of a rich library of components ready to be instanced into the ESP tile-based architecture with the help of the SoC integration flow.

Thanks to the automatic generation of device drivers from pre-designed templates, the ESP methodology simplifies the invocation of accelerators from user-level applications executing on top of Linux [25, 39]. Through the automatic generation of a network-on-chip (NoC) from a parameterized model, the ESP architecture can scale to accommodate many processors, tens of accelerators, and a distributed memory hierarchy [27]. A set of *platform services* provides pre-validated solutions to access or manage SoC resources, including accelerators configuration [41], memory management [39], and dynamic voltage frequency scaling (DVFS) [40], among others. ESP comes with a GUI that guides the designers through the interactive choice and placement of the tiles in the SoC and it has push-button capabilities for rapid prototyping of the SoC on FPGA.

Open-Source Hardware. OSH holds the promise of boosting hardware development and creating new opportunities for academia
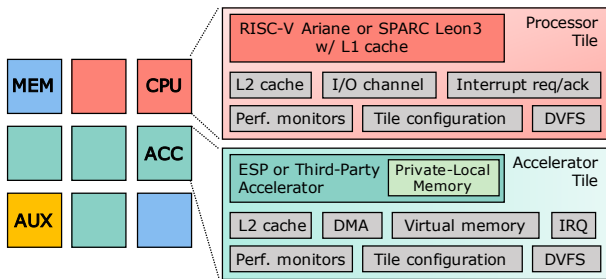

Figure 2: Example of a 3x3 instance of ESP with a high-level overview of the sockets for processors and accelerators.

and entrepreneurship [30]. In recent years, no other project has contributed to the growth of the OSH movement more than RISC-V [6]. To date, the majority of OSH efforts have focused on the development of processor cores that implement the RISC-V ISA and small-scale SoCs that connect these cores with tightly-coupled functional units and coprocessors, typically through bus-based interconnects. Meanwhile, there have been less efforts in developing solutions for large-scale SoCs that combine RISC-V cores with many loosely-coupled components, such as coarse-grained accelerators [19], interconnected with a NoC. With this gap in mind, we have made an open-source release of ESP to provide the OSH community with a platform for heterogeneous SoC design and prototyping [18].

2 THE ESP ARCHITECTURE

The ESP architecture is structured as a heterogeneous tile grid. For a given application domain, the architect decides the structure of the SoC by determining the number and mix of tiles. For example, Figure 2 shows a 9-tile SoC organized in a 3×3 matrix. There are four types of tiles: processor tile, accelerator tile, memory tile for the communication with main memory, and auxiliary tile for peripherals, like UART and Ethernet, or system utilities, like the interrupt controller and the timer. To support a high degree of scalability, the ESP tiles are connected by a *multiplane NoC* [49].

The content of each tile is encapsulated into a *modular socket* (aka *shell*), which interfaces the tile to the NoC and implements the platform services. The socket-based approach, which decouples the design of a tile from the design of the rest of the system, is one of the key elements of the agile ESP SoC design flow. It highly simplifies the design effort of each tile by taking care of all the system integration aspects, and it facilitates the reuse of intellectual property (IP) blocks. For instance, the ESP accelerator socket implements services for DMA, cache coherence, performance monitors, and distributed interrupt requests.

At design time, it is possible to choose the set of services to instantiate in each tile. At runtime, the services can be enabled and many of them offer reconfigurability options, e.g., dynamic reconfiguration of the cache-coherence model [28].

The ESP architecture implements a distributed system that is inherently scalable, modular and heterogeneous, where processors and accelerators are given the same importance in the SoC. Differently from other OSH platforms, ESP proposes a system-centric view, as opposed to a processor-centric view.

2.1 System Interconnect

Processing elements act as transaction masters that access peripherals and slave devices distributed across remote tiles. All remote communication is supported by a NoC, which is a transparent communication layer. Processors and accelerators, in fact, operate as if all remote components were connected to their local bus controller in the ESP sockets. The sockets include standard bus ports, bridges, interface adapters and proxy components that provide a complete decoupling from the network interface. Figure 3 shows in detail the modular architecture of the ESP interconnect for the case of a six-plane NoC. Every platform service is implemented by a pair of proxy components. One proxy translates requests from bus masters, such as processors and accelerators, into transactions for one of the NoC planes. The other proxy forwards requests from the NoC planes to the target slave device, such as last-level cache (LLC) partitions, or Ethernet. For each proxy, there is a corresponding buffer queue, located between the tile port of the NoC routers and the proxy itself. In Figure 3, the color of a queue depends on the assigned NoC plane. The number and direction of the arrows connected to the queues indicate whether packets can flow from the NoC to the tile, from the tile to the NoC, or in both directions. The arrows connect the queues to the proxies. These are labeled with the name of the services they implement and the number of the NoC plane used for receiving and sending packets, respectively.

The current implementation of the ESP NoC is a packet-switched 2D-mesh topology with look-ahead dimensional routing. Every hop takes a single clock cycle because arbitration and next-route computation are performed concurrently. Multiple physical planes allow protocol-deadlock prevention and provide sufficient bandwidth for the various message types. For example, since a distributed directory-based protocol for cache coherence requires three separate channels, planes 1, 2 and 3 in Figure 3 are assigned to request, forward, and response messages, respectively. Concurrent DMA transactions, issued by multiple accelerators and handled by various remote memory tiles, require separate request and response planes. Instead of reusing the cache-coherence planes, the addition of two new planes (4 and 6 in Figure 3) increases the overall NoC bandwidth. Finally, one last plane is reserved for short messages, including interrupt, I/O configuration, monitoring and debug.

Currently, customizing the NoC topology is not automated in the ESP SoC integration flow. System architects, however, may explore different topologies by modifying the router instances and updating the logic to generate the header flit for the NoC packets [50].

2.2 Processor Tile

Each processor tile contains a processor core that is chosen at design time among those available: the current choice is between the RISC-V 64-bit Ariane core from ETH Zurich [1, 51] and the SPARC 32-bit LEON3 core from Cobham Gaisler [17]. Both cores are capable of running Linux and they come with their private L1 caches. The processor integration into the distributed ESP system is transparent: no ESP-specific software patches are needed to boot Linux. Each processor communicates on a local bus and is agnostic of the rest of the system. The memory interface of the LEON3 core requires a 32-bit AHB bus, whereas Ariane comes with a 64-bit AXI interface. In addition to proxies and bus adapters, the processor socket provides a unified private L2 cache of configurable size, which implements a
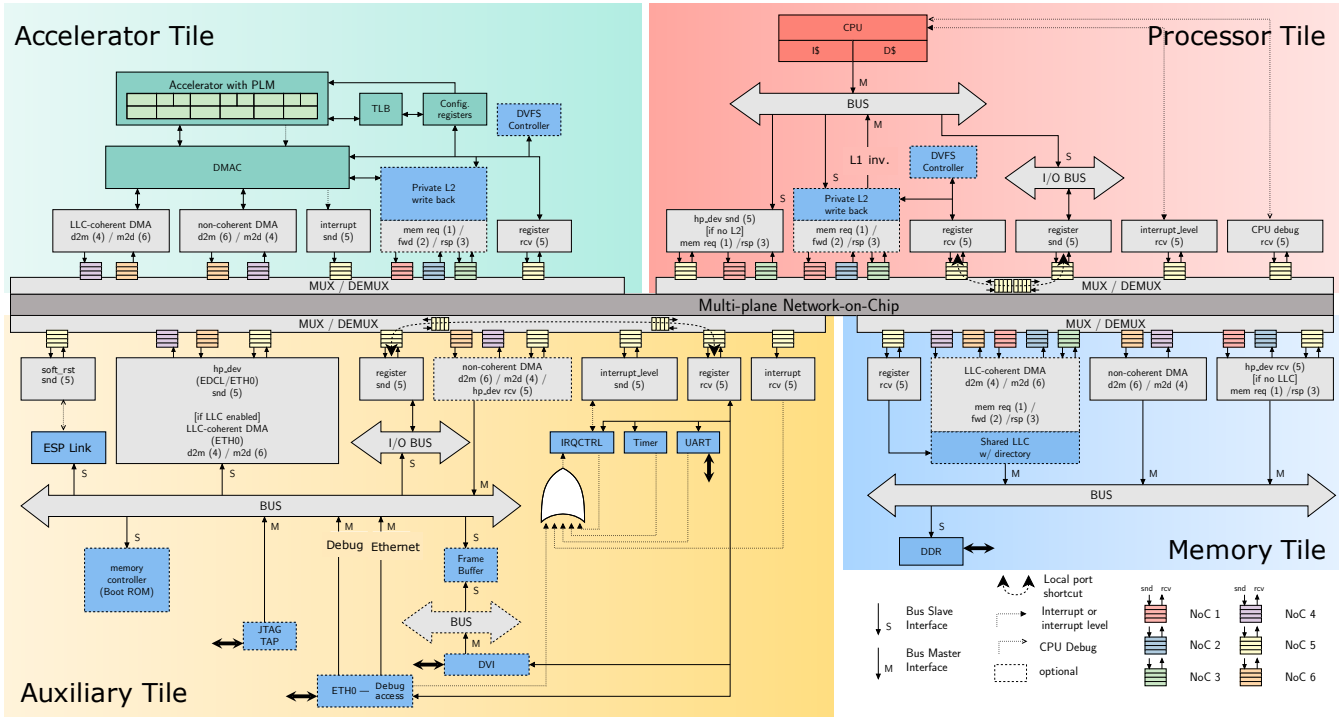
Figure 3: Detailed architecture of the NoC interface for four main ESP tiles.

directory-based MESI cache-coherence protocol. Processor requests directed to memory-mapped I/O registers are forwarded by the socket to the IO/IRQ NoC plane through an APB adapter. The only processor-specific component of the socket is an interrupt-level proxy, which implements the custom communication protocol between the processor and the interrupt controller and system timer in the auxiliary tile.

2.3 Memory Tile

Each memory tile contains a channel to external DRAM. The number of memory tiles can be configured at design time. Typically, it varies from one to four depending on the size of the SoC. All necessary hardware logic to support the partitioning of the addressable memory space is automatically generated and the partitioning is completely transparent to software. Each memory tile also contains a configurably-sized partition of the LLC with the corresponding directory. The LLC in ESP implements an extended MESI protocol, in combination with the private L2 cache in the processor tiles, that supports Linux with symmetric multiprocessing, as well as runtime reconfigurable coherence for accelerators [28].

2.4 Accelerator Tile

This tile contains the specialized hardware of a loosely-coupled accelerator [19]. This type of accelerator executes a coarse-grained task independently from the processors while exchanging large datasets with the memory hierarchy. To be integrated in the ESP tile, illustrated on the top-left portion of Figure 3, accelerators should comply to a simple interface that includes load/store ports for latency-insensitive channels [10, 13], signals to configure and

start the accelerator, and an acc_done signal to notify the accelerator completion and generate an interrupt for the processors. ESP accelerators that are newly designed with one of the supported design flows automatically comply with this interface. For existing accelerators, ESP offers a third-party integration flow [24]. In this case, the accelerator tile has only a subset of the proxy components because the configuration registers, DMA for memory access, and TLB for virtual memory [39] are replaced by standard bus adapters.

The set of platform services provided by the socket relieves the designer from the burden of "reinventing the wheel" with respect to implementing accelerator configuration through memory-mapped registers, address translation, and coherence protocols. Furthermore, the socket enables point-to-point communication (P2P) among accelerator tiles so that they can exchange data directly instead of using necessarily shared-memory communication.

Third-party accelerators can use the services to issue interrupt requests, receive configuration parameters and initiate DMA transactions. They are responsible, however, for managing shared resources, such as reserved memory regions, and for implementing their own custom hardware-software synchronization protocol.

The run-time reconfigurable coherence protocol service is particularly relevant for accelerators. In fact, there is no static coherence protocol that can necessarily serve well all invocations of a set of heterogeneous accelerators in a given SoC [26]. With the *non-coherent DMA* model, an accelerator bypasses the cache hierarchy to exchange data directly with main memory. With the *fully-coherent* model, the accelerator communicates with an optional private cache placed in the accelerator socket. The ESP cache hierarchy augments a directory-based MESI protocol with support for two models where

accelerators send requests directly to the LLC, without owning a private cache: the *LLC-coherent DMA* and the *coherent DMA* models. The latter keeps the accelerator requests coherent with respect to all private caches in the system, whereas the former does not. While *fully-coherent* and *coherent DMA* are fully handled in hardware by the ESP cache hierarchy, *non-coherent DMA* and *LLC-coherent DMA* demand that software acquires appropriate locks and flushes private caches before invoking accelerators. These synchronization mechanisms are implemented by the ESP device drivers, which are generated automatically when selecting any of the supported HLS flows discussed in Section 4.

2.5 Auxiliary Tile

The auxiliary tile hosts all shared peripherals in the system except from memory: the Ethernet NIC, UART, a digital video interface, a debug link to control ESP prototypes on FPGA and a monitor module that collects various performance counters and periodically forwards them through the Ethernet interface.

As shown in Figure 3, the socket of the auxiliary tile is the most complex because most platform services must be available to serve the devices hosted by this tile. The interrupt-level proxy, for instance, manages the communication between the processors and the interrupt controller. Ethernet, which requires coherent DMA to operate as a slave peripheral, enables users to remotely log into an ESP instance via SSH. The frame-buffer memory, dedicated to the video output, is connected to one proxy for memory-mapped I/O and one for non-coherent DMA transactions. These enable both processor cores and accelerators to write directly into the video frame buffer. The Ethernet debug interface [23], instead, uses the memory-mapped I/O and register access services to allow ESP users to monitor and debug the system through the *ESP Link* application. Symmetrically, UART, timer, interrupt controller and the bootrom are controlled by any master in the system through the counterpart proxies for memory-mapped I/O and register access. Hence, the auxiliary tile includes both pairs of proxies. These enable an additional communication path, labeled as *local port shortcut* in Figure 3, which connects the masters in the auxiliary tile (i.e. the Ethernet debug link) with slaves that do not share the same local bus. A similar shortcut in the processor tile allows a processor to flush its own private L2 cache and manage its local DVFS controller.

3 THE ESP SOFTWARE STACK

The ESP accelerator's Application Programming Interface (API) library simplifies the invocation of accelerators from a user application, by exposing only three functions to the programmer [25]. Underneath, the API invokes the accelerators with the automatically generated Linux device drivers. The API is lightweight and can be targeted from existing applications or by a compiler. For a given application, the software execution of a computationally intensive kernel can be replaced with hardware accelerators by means of a single function call (esp_run()). Figure 4 shows the case of an application with four computation kernels, two executed in software and two implemented with an accelerator. The configuration argument passed to esp_run() is a simple data structure that specifies which accelerator(s) to invoke, how to configure them, and their point-to-point dependencies, if any. By using the esp_alloc() and esp_free() functions for memory allocation, data can be truly
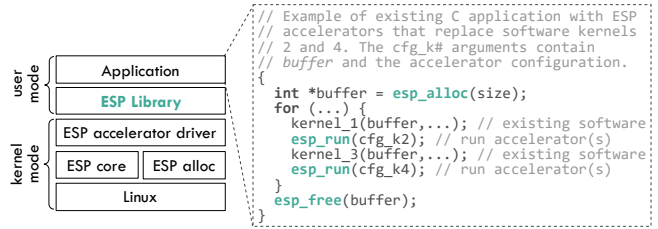

Figure 4: ESP accelerator API for seamless shared memory.

shared between accelerators and processors, i.e., no data copies are necessary. Data are allocated in an efficient way to improve the accelerator's access to memory without compromising the software's performance [39]. The ESP software stack, combined with the generation of device drivers for new custom accelerators, makes the accelerator invocation as transparent as possible for the application programmers.

4 THE ESP METHODOLOGY

The ESP design methodology is flexible because it embodies different design flows, for the most part automated and supported by commercial CAD tools. In particular, recalling Figure 1, the *accelerator design flow* (on the left in the figure) aids the creation of an IP library, whereas the *SoC flow* (on the right) automates the integration of heterogeneous components into a complete SoC.

4.1 Accelerator Flow

The end goal of this flow is to add new elements to the library of accelerators that can be automatically instantiated with the SoC flow. Designers can work at different abstraction levels with various specification languages:

- Cycle-accurate RTL descriptions with languages like VHDL, Verilog, SystemVerilog, or Chisel.
- Loosely-timed or un-timed behavioral descriptions with SystemC or C/C++ that get synthesized into RTL with high-level synthesis (HLS) tools. ESP currently supports the three main commercial HLS tools: Cadence Stratus HLS, Mentor Catapult, and Xilinx Vivado HLS.
- Domain-specific libraries for deep learning like Keras TensorFlow, PyTorch, and ONNX, for which ESP offers a flow combining HLS tools with hls4ml, an OSH project [2, 22].

HLS-Based Accelerator Design. For the HLS-based flows, ESP facilitates the job of the accelerator designer by providing ESP-compatible accelerator templates, HLS-ready skeleton specifications, multiple examples, and step-by-step tutorials for each flow.

The push in the adoption of HLS from C/C++ specifications has many reasons: (1) a large codebase of algorithms written in these languages, (2) a simplified hardware/software co-design (since most embedded software is in C), and (3) a thousand-fold faster functional execution of the specification than the counterpart RTL simulation. On the other hand, HLS from C/C++ has also shown some limitations because of the lack of ability to specify or accurately infer concurrency, timing, and communication properties of the hardware systems. HLS flows based on the IEEE-standard
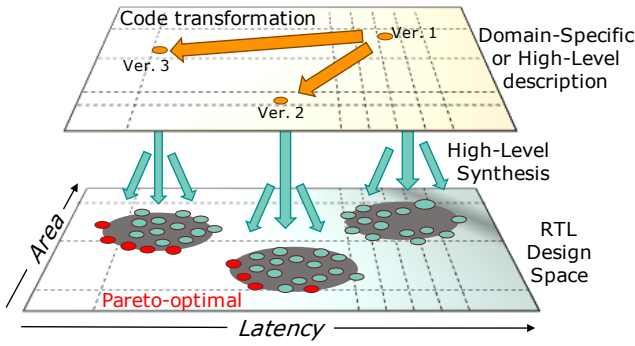
Figure 5: HLS-based accelerator design in ESP.

Figure 6: Structure of ESP accelerators.

language SystemC overcome these limitations, thus making SystemC the de-facto standard to model protocols and control-oriented applications at a level higher than RTL.

In ESP, we support and encourage the use of both C/C++ and SystemC flows for HLS and we have defined a set of guidelines to support the designers in porting an application to an HLS-ready format. The ideal starting point is a self-contained description of the computation kernel, written in a subset of the C/C++ language [42]: a limited use of pointers and the absence of dynamic memory allocation and recursion are important; also, aside from common mathematical functions, no external library functions should be used. This initial software transformation is oftentimes the most important step to obtain good quality source code for HLS [41].

The designer of an ESP accelerator should aim at a well-structured description that partitions the specification into concurrent functional blocks. The goal is to obtain any synthesizable specification that enables the exploration of a vast design space, by evaluating many micro-architectural and optimization choices. Figure 5 shows the relationship between the C/C++/SystemC design space and the RTL design space. The HLS tools provide a rich set of configuration knobs to obtain a variety of RTL implementations, each corresponding to a different cost-performance tradeoff point [36, 37]. The knobs are push-button directives of the HLS tool represented by the green arrows. Designers may also perform manual transformations of the specification (orange arrows) to explore the design space while preserving the functional behavior. For example, they can expose parallelism by removing false dependencies or they can reduce resource utilization by encapsulating sections of code with similar behavior into callable functions [41].

Accelerator Structure. The ESP accelerators are based on the loosely-coupled model [19]. They are programmed like devices by applications that invoke device drivers with standard system calls, such as `open` and `ioctl`. They perform coarse-grained computations while exchanging large data sets with the memory hierarchy. Figure 6 shows the structure and interface common to all ESP accelerators. The interface channels allow the accelerator to (1) communicate with the CPU via memory-mapped registers (`conf_info`), (2) program the DMA controller or interact with other accelerators (`load_ctrl` and `store_ctrl`), (3) exchange data with the memory hierarchy or other accelerators, (`load_chnl` and `store_chnl`), and (4) notify its completion back to the software application (`acc_done`).
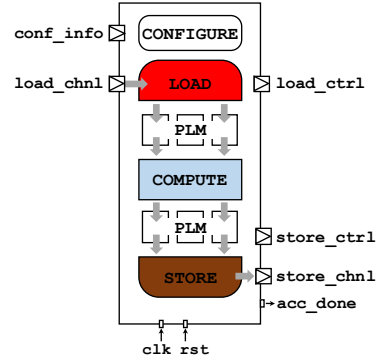
These channels are implemented with latency-insensitive communication primitives, which HLS tools commonly provide as libraries (e.g. Mentor MatchLib Connections [33], Cadence Flex Channels [38], Xilinx `ap_fifo`). These primitives preserve functional correctness in the presence of latency variation both in the computation within the accelerator and in the communication across the NoC [10]. This is obtained by adding *valid* and *ready* signals to the channels. The valid signal indicates that the value of the data bundle is valid in the current clock cycle, while the ready signal is de-asserted to apply backpressure. The latency-insensitive nature of ESP accelerators allows designers to fully exploit the ability of HLS to produce many alternative RTL implementations, which are not strictly equivalent from an RTL viewpoint (i.e., they do not produce the same timed sequence of outputs for any valid input sequence), but they are members of a latency-equivalent design class [14]. Each member of this class can be seamlessly replaced with another one, depending on performance and cost targets [44].

The execution flow of an ESP accelerator consists of four phases, *configure*, *load*, *compute*, and *store*, as shown in Figure 6. A software application configures, checks, and starts the accelerator via memory-mapped registers. During the load and store phases, the accelerator interacts with the DMA controller, interleaving data exchanges between the system and the accelerator's private local memory (PLM) with computation. When the accelerator completes its task, an interrupt resumes the software for further processing.

For better performance, the accelerator can have one or more parallel computation components that interact with the PLM. The organization of the PLM itself is typically customized for the given accelerator, with multiple banks and ports. For example, the designer can organize it as a circular or ping-pong buffers to support the pipelining of computation and transfers with the external memory or other accelerators. Designers can leverage PLM generators [45] to implement many different memory subsystems, each optimized for a specific combination of HLS knobs settings.

Accelerator Behavior. The charts of Figure 7 show the behavior of two concurrent ESP accelerators (ACC0 and ACC1) in three possible scenarios. The two accelerators work in a producer-consumer fashion: ACC0 generates data that ACC1 takes as inputs. The accelerators are executed two times and concurrently; the consumer starts as soon as the data is ready; finally, both the accelerators perform
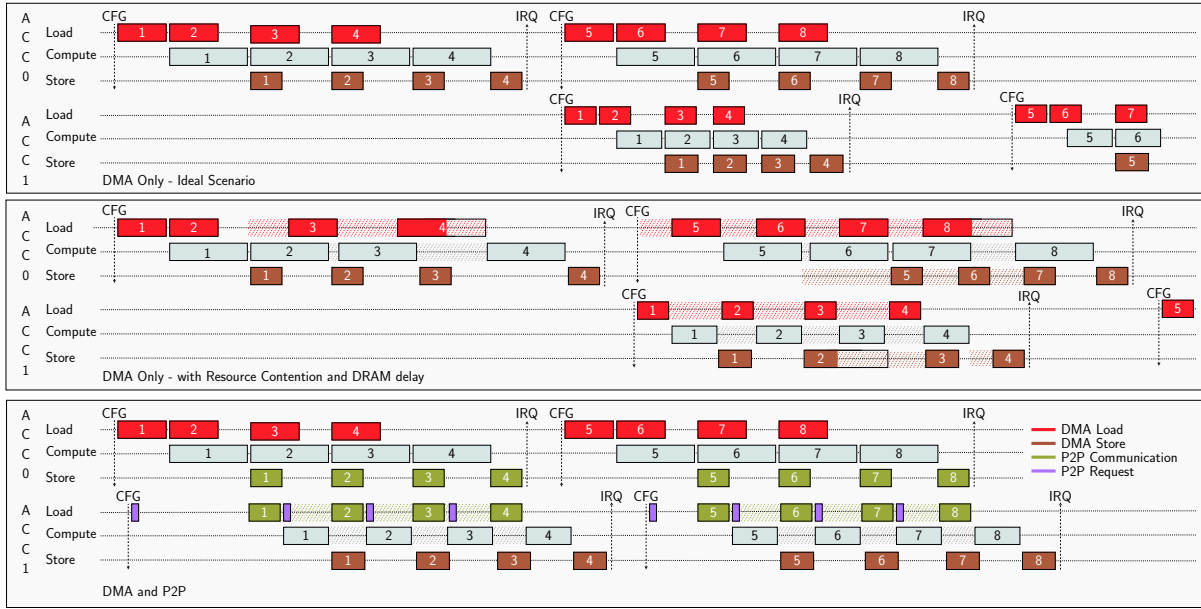
Figure 7: Overlapping of computation and communication of ESP accelerators.

burst of load and store DMA transactions, in red and brown respectively. The completion of the configuration phase and interrupt request (acc_done) are marked with CFG and IRQ, respectively.

In the top scenario, the two accelerators communicate via external memory. First, the producer ACC0 runs and stores the resulting data in main memory. Upon completion of the producer, the consumer ACC1 starts and accesses the data in main memory; concurrently, the producer ACC0 can run a second time. The data exchange happens through memory at the granularity of the whole accelerator data set. This scenario is a *virtual pipeline of ESP accelerators through memory*. Ping-pong buffering on the PLM for both load and store phases allows the overlap of computation and communication. In addition, load and store phases are allowed to overlap. This is only possible by assuming to have dedicated memory channels for each accelerator (e.g. two ESP memory tiles). As long as the NoC and memory bandwidth are not saturated, the performance overhead is limited only to the driver run time and the interrupt handling procedures. We consider this scenario ideal.

In complex SoCs, it is reasonable to expect resource contention and delays with the main memory. This can potentially limit the latency and throughput of the accelerators, as shown in the middle scenario of Figure 7, where some of the DMA transactions get delayed for both the producer and consumer accelerators. The ESP library and API allows designers to replace the described software pipeline, with an actual pipeline of accelerators, based on *point-to-point communication (P2P) over the NoC*. The communication method does not need to be chosen at design time; instead, special configuration registers are used to overwrite the default DMA behavior. Beside relieving memory contention, P2P communication can actually improve latency and throughput of communicating accelerators, as shown in the bottom scenario of Figure 7. Here, each output transaction of the producer ACC0 is matched to an input

transaction of the consumer ACC1 (in green). Differently from the previous scenarios, the data exchange via P2P happens at a smaller granularity: a single store transaction of the producer ACC0 is a valid input for the consumer ACC1. A designer should take into account this assumption when designing accelerators for a specific task.

Accelerator Templates and Code Generator. ESP provides the designers with a set of accelerator templates for each of the HLS-based design flows. These templates leverage concepts of object-oriented programming and class inheritance to simplify the design of the accelerators in C/C++ or SystemC and enforce the interface and structure previously described. They also implicitly address the differences existing among the various HLS tools and input specification languages. For example, the latency-insensitive primitives, which come with the different vendors, may have slightly different APIs, e.g. Put()/Get() vs. Read()/Write(), or timing behavior. With some HLS tools, the designer has to specify some extra wait() statements in SystemC to generate the correct RTL code. In the case of C/C++ designs a combination of HLS directives and coding style must be followed to ensure that extra memories are not inadvertently inferred and the phases are correctly synchronized.

Next to templates, ESP provides a further aid for the accelerator design: an interactive environment that generates a fully-working and HLS-ready accelerator *skeleton* from a set of parameters passed by the designer. The skeleton comes with a unit testbench, synthesis and simulation scripts, a bare-metal driver, a Linux driver, and a sample test application. This is the first step of the accelerator design flow, as shown on the top-right of Figure 8. The skeleton is a basic specification that uses the templates and contains placeholder for manual accelerator-specific customizations. The parameters passed by the designers include: unique name and ID, desired HLS tool flow, a list of application-specific configuration registers, bit-width of the data tokens, size of the data set and number of batches of data sets
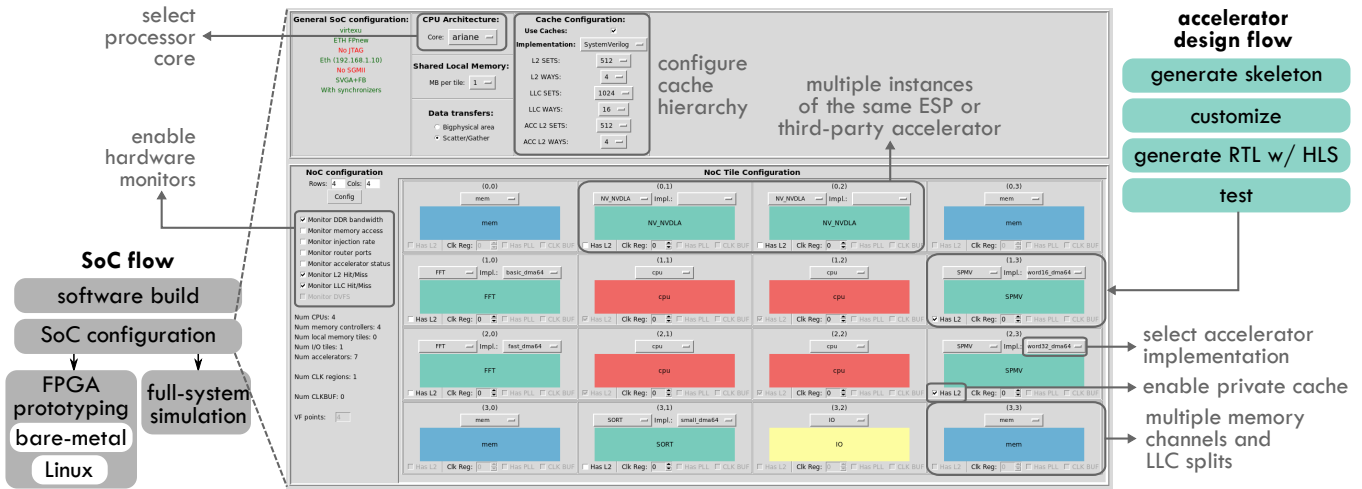
Figure 8: Overview of the accelerator and SoC design flows with an example of SoC design configuration on the ESP GUI.

to be executed without interrupting the CPU. Next to application-specific information, designers can choose architectural parameters that set the minimum required size of the PLM and the maximum memory footprint of the application that invokes the accelerator. These parameters have effect on the generated accelerator skeleton, device-driver, test application, and on the configuration parameters for the ESP socket that will host the accelerator.

Starting from the automatically generated skeleton, designers must customize the accelerator computation phase, leveraging the software implementation of the target computation kernel as a reference. In addition, they are responsible for customizing the input generation and output validation functions in the unit testbench and in the bare-metal and Linux test applications. Finally, in case of complex data access patterns, they may also need to extend the communication part of the accelerator and define a more complex structure for the PLM. The ESP release offers a set of online tutorials that describe these steps in details with simple examples, which demonstrate how the first version of a new accelerator can be designed, integrated and tested on FPGA in a few hours [18].

The domain specific flow for embedded machine learning is fully automated [25]. The accelerator and the related software drivers and application are generated in their entirety from the neural-network model. ESP automatically generates also the accelerator tile socket and a wrapper for the accelerator logic.

4.2 Third-Party Accelerator Integration

For existing accelerators, ESP provides a third-party accelerator integration flow (TPF). The TPF skips all the steps necessary to design a new accelerator and goes directly to SoC integration. The designer must provide some information about the existing IP block and a simple wrapper to connect the wires of the accelerator's interface to the ESP socket. Specifically, the designer must fill in a short XML file with a unique accelerator name and ID, the list and polarity of the reset signals, the list of clock signals, an optional prefix for the AXI master interface in the wrapper, the user-defined width of AXI optional control signals and the type of interrupt request (i.e., level or edge sensitive). In addition, the TPF requires

the list of RTL source files, including Verilog, SystemVerilog, VHDL and VHDL packages, a custom Makefile to compile the third-party software and device drivers, and the list of executable files, libraries and other binary objects needed to control the accelerator.

Currently, ESP provides adapters for AXI master (32 and 64 bits), AHB master (32 bits) and AXI-Lite or APB slave (32 bits). As long as the target accelerator is compliant with these standard bus protocols, the Verilog top-level wrapper consists of a simple wire assignment to expose bus ports to the ESP socket and connect any non-standard input port of the third-party accelerator (e.g. disable test mode), if present. After these simple manual steps, ESP takes care of the whole integration automatically. We used the TPF to integrate the NVDLA [3]. An online tutorial in the ESP release demonstrates the design of a complete SoC with multiple NVDLA tiles, multiple memory tiles and the Ariane RISC-V processor. This system can run up to four concurrent machine-learning tasks using the original NVDLA software stack[1] [24].

4.3 SoC Flow

The center and the left portion of Figure 8 illustrate the agile SoC development enabled by ESP. Both the ESP and third-party accelerator flows contribute to the pool of IP components that can be selected to build an SoC instance. The ESP GUI guides the designers through an interactive SoC design flow that allows them to: choose the number, types and positions of tiles, select the desired Pareto-optimal design point from the HLS flows for each accelerator, select the desired processor core among those available, determine the cache hierarchy configuration, select the clock domains for each tile, and enable the desired system monitors. The GUI writes a configuration file that the ESP build flow can include to generate RTL sockets, the system memory mapping, NoC routing tables, the device tree for the target processor architecture, software header files, and configuration parameters for the proxy components.

A single make target is sufficient to generate the bitstream for one of the supported Xilinx evaluation boards (VCU128, VCU118 and

---

[1]A minor patch was required to run multiple NVDLAs in a Linux environment.

VC707) and proFPGA prototyping FPGA modules (XCVU440 and XC7V2000T). Another single `make` target compiles Linux and creates a default root file system that includes accelerators' drivers and test applications, together with all necessary initialization scripts to load the ESP library and memory allocator. If properly listed during the TPF, the software stack for the third-party accelerators is loaded into the Linux image as well. When the FPGA implementation is ready, users can load the boot loader onto the ESP boot memory and the Linux image onto the external DRAM with the *ESP Link* application and the companion module on the auxiliary tile. Next *ESP Link* sends a soft reset to the processor cores, thus starting the execution from the boot loader. Users can monitor the boot process via UART, or log in with SSH after Linux boot completes. The online tutorials explain how to properly wire the FPGA boards to a simple home router to ensure connectivity.

In addition to FPGA prototyping, designers can run full-system RTL simulations of a bare-metal program. If monitoring the FPGA with the UART serial interface, they can run bare-metal applications on FPGA as well. The development of bare-metal and Linux applications for an ESP SoC is facilitated by the ESP software stack described in Section 3. The ESP release offers several examples.

The agile ESP flow allowed us to rapidly prototype many complex SoCs on FPGA, including:

- An SoC with 12 computer vision accelerators, with as many dynamic frequency scaling (DFS) domains [40].
- A multi-core SoC booting Linux SMP with tens of accelerators, multiple DRAM controllers, and dynamically reconfigurable cache coherence models [28].
- A RISC-V based SoCs where deep learning applications running on top of Linux invoke loosely-coupled accelerators designed with multiple ESP accelerator design flows [25].
- A RISC-V based SoCs with multiple instances of the NVDLA controlled by the RISC-V Ariane processor [24].

5 RELATED WORK

The OSH movement is supported by multiple SoC design platforms, many based on the RISC-V open-standard ISA [6, 29]. The *Rocket Chip Generator* is an OSH project that leverages the Chisel RTL language to construct SoCs with multiple RISC-V cores connected through a coherent TileLink bus [35]. The *Chipyard* framework inherits Rocket Chip's Chisel-based parameterized hardware generator methodology and also allows the integration of IP blocks written in other RTL languages, via a Chisel wrapper, as well as domain-specific accelerators [5]. *Celerity* used the custom co-processor interface *RoCC* of the Rocket chip to integrate five Rocket cores with an array of 496 simpler RISC-V cores and a binarized neural network (BNN) accelerator, which was designed with HLS, into a 385-million transistor SoC [21]. *HERO* is an FPGA-based research platform that allows the integration of a standard host multicore processor with programmable manycore accelerators composed of clusters of RISC-V cores based on the PULP platform [4, 34, 47]. *OpenPiton* was the first open-source SMP Linux-booting RISC-V multicore processor [8]. It supports the research of heterogeneous ISAs and provides a coherence protocol that extends across multiple chips [7, 46]. *Blackparrot* is a multicore RISC-V architecture that

offers some support for the integration of loosely-coupled accelerators [43]; currently, it provides two of the four cache-coherence options supported by ESP: fully-coherent and non-coherent.

While most of these platforms are built with a processor-centric perspective, ESP promotes a system-centric perspective with a scalable NoC-based architecture and a strong focus on the integration of heterogeneous components, including particularly loosely-coupled accelerators. Another feature distinguishing ESP from the other open-source SoC platforms is the flexible system-level design methodology that embraces a variety of specification languages and synthesis flows, while promoting the use of HLS to facilitate the design and integration of accelerators.

6 CONCLUSIONS

In summary, with ESP we aim at contributing to the open-source movement by supporting the realization of more scalable architectures for SoCs that integrate more heterogeneous components, thanks to a more flexible design methodology that accommodates different specification languages and design flows. Conceived as a heterogeneous integration platform and tested through years of teaching at Columbia University, ESP is naturally suited to foster collaborative engineering of SoCs across the OSH community.

REFERENCES

[1] Ariane. https://github.com/pulp-platform/ariane.
[2] HLS4ML. https://fastmachinelearning.org/hls4ml/.
[3] NVIDIA Deep Learning Accelerator (NVDLA). www.nvdla.org.
[4] PULP. https://pulp-platform.org/.
[5] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanovic, and B. Nikolic. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21.
[6] K. Asanovic and D. Patterson. 2014. The Case for Open Instruction Sets. *Microprocessor Report* (Aug. 2014).
[7] J. Balkind, T. Chang, P. J. Jackson, G. Tziantzioulis, A. Li, F. Gao, A. Lavrov, G. Chirkov, J. Tu, M. Shahrad, and D. Wentzlaff. 2020. OpenPiton at 5: A Nexus for Open and Agile Hardware Design. *IEEE Micro* 40, 4 (2020), 22–31.
[8] J. Balkind, K. Lim, F. Gao, J. Tu, D. Wentzlaff, M. Schaffner, F. Zaruba, and L. Benini. 2019. OpenPiton+ Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*. 1–6.
[9] S. Borkar and A. Chen. 2011. The Future of Microprocessors. *Communication of the ACM* 54 (May 2011), 67–77. Issue 5.
[10] L. P. Carloni. 2015. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proceedings of the IEEE* 103, 11 (Nov. 2015), 2133–2151.
[11] L. P. Carloni. 2016. The Case for Embedded Scalable Platforms. In *Proc. of the Design Automation Conference (DAC)*. 17:1–17:6.
[12] L. P. Carloni, E. G. Cota, G. Di Guglielmo, D. Giri, J. Kwon, P. Mantovani, L. Piccolboni, and M. Petracca. 2019. Teaching Heterogeneous Computing with System-Level Design Methods. In *Workshop on Computer Architecture Education (WCAE)*. 1–8.

[13] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. 1999. A Methodology for "Correct-by-Construction" Latency Insensitive Design. In *Proc. of the International Conference on Computer-Aided Design*. 309–315.

[14] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. 2001. Theory of Latency-Insensitive Design. *IEEE Transactions on CAD of Integrated Circuits and Systems* 20, 9 (Sept. 2001), 1059–1076.

[15] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2016. A Cloud-Scale Acceleration Architecture. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13.

[16] H. Chishiro, K. Suito, T. Ito, S. Maeda, T. Azumi, K. Funaoka, and S. Kato. 2019. Towards Heterogeneous Computing Platforms for Autonomous Driving. In *Proc. of the International Conference on Embedded Software and Systems (ICESS)*.

[17] Cobham Gaisler. LEON3. www.gaisler.com/index.php/products/processors/leon3.

[18] Columbia SLD Group. ESP Release. www.esp.cs.columbia.edu.

[19] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. 2015. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Proc. of the Design Automation Conference (DAC)*. 202:1–202:6.

[20] W. Dally, Y. Turakhia, and S. Han. 2020. Domain-Specific Hardware Accelerators. *Communication of the ACM* 63, 7 (June 2020), 48–57.

[21] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (Feb. 2018), 30–41.

[22] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (July 2018), P07027–P07027.

[23] J. Gaisler. 2004. An Open-Source VHDL IP Library with Plug & Play Configuration. *Building the Information Society* (2004).

[24] D. Giri, K.-L. Chiu, G. Eichler, P. Mantovani, N. Chandramoorth, and L. P. Carloni. 2020. Ariane + NVDLA: Seamless Third-Party IP Integration with ESP. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[25] D. Giri, K-L. Chiu, G. Di Guglielmo, P. Mantovani, and L.P. Carloni. 2020. ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning. In *Proc. of the Conference on Design, Automation, and Test in Europe (DATE)*. 1049–1054.

[26] D. Giri, P. Mantovani, and L. P. Carloni. 2018. Accelerators & Coherence: An SoC Perspective. *IEEE Micro* 38, 6 (Nov. 2018), 36–45.

[27] D. Giri, P. Mantovani, and L. P. Carloni. 2018. NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators. In *Proc. of the International Symposium on Networks-on-Chip (NOCS)*. 1:1–1:8.

[28] D. Giri, P. Mantovani, and L. P. Carloni. 2019. Runtime Reconfigurable Memory Hierarchy in Embedded Scalable Platforms. In *Proc. of the Asia and South Pacific Design Automation Conference (ASPDAC)*. 719–726.

[29] S. Greengard. 2020. Will RISC-V Revolutionize Computing? *Commun. ACM* 63, 5 (April 2020), 30–32.

[30] G. Gupta, T. Nowatzki, V. Gangadhar, and K. Sankaralingam. 2017. Kickstarting Semiconductor Innovation with Open Source Hardware. *IEEE Computer* 50, 6 (June 2017), 50–59.

[31] M. Horowitz. 2014. Computing's Energy Problem (and What We Can Do About It). In *International Solid-State Circuits Conference (ISSCC)*. 10–14.

[32] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. 2018. A Domain-Specific Architecture for Deep Neural Networks. *Commun. ACM* 61, 9 (Aug. 2018), 50–59.

[33] B. Khailany, E. Krimer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. L. Xi, Y. Zhang, and B. Zimmer. 2018. A Modular Digital VLSI Flow for High-Productivity SoC Design. In *Proc. of the Design Automation Conference (DAC)*. 1–6.

[34] A. Kurth, P. Vogel, A. Capotondi, A. Marongui, and L. Benini. 2017. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*. 1–7.

[35] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanovic. 2016. An Agile Approach to Building RISC-V Microprocessors. *IEEE Micro* 36, 2 (Mar.-Apr. 2016), 8–20.

[36] H-Y. Liu and L. P. Carloni. 2013. On Learning-based Methods for Design-Space Exploration with High-Level Synthesis. In *Proc. of the Design Automation Conference (DAC)*. 1–7.

[37] H-Y. Liu, M. Petracca, and L. P. Carloni. 2012. Compositional System-Level Design Exploration with Planning of High-Level Synthesis. In *Proc. of the Conference on Design, Automation, and Test in Europe (DATE)*. 641–646.

[38] M. Meredith. 2008. High-level SystemC Synthesis with Forte's Cynthesizer. In *High-Level Synthesis*. Springer, 75–97.

[39] P. Mantovani, E. G. Cota, C. Pilato, G. Di Guglielmo, and L. P. Carloni. 2016. Handling Large Data Sets for High-Performance Embedded Applications in Heterogeneous Systems-on-Chip. In *Proc. of the Intl. Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*. 1–10.

[40] P. Mantovani, E. G. Cota, K. Tien, C. Pilato, G. Di Guglielmo, K. Shepard, and L. P. Carloni. 2016. An FPGA-Based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems. In *Proc. of the Design Automation Conference (DAC)*. 157:1–157:6.

[41] P. Mantovani, G. Di Guglielmo, and L. P. Carloni. 2016. High-level Synthesis of Accelerators in Embedded Scalable Platforms. In *Proc. of the Asia and South Pacific Design Automation Conference (ASPDAC)*. 204–211.

[42] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on CAD of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604.

[43] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor. 2020. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102.

[44] L. Piccolboni, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. 2017. COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators. *ACM Transactions on Embedded Computing Systems* 16, 5s (Sept. 2017), 150:1–150:22.

[45] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. 2017. System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip. *IEEE Transactions on CAD of Integrated Circuits and Systems* 36, 3 (March 2017), 435–448.

[46] Princeton Parallel Group. OpenPiton. https://parallel.princeton.edu/openpiton/.

[47] D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu. 2014. Energy Efficient Parallel Computing on the PULP Platform with Support for OpenMP. In *Convention of Electrical Electronics Engineers in Israel (IEEEI)*.

[48] Y. S. Shao, B. Reagen, G. Wei, and D. Brooks. 2015. The Aladdin Approach to Accelerator Design and Modeling. *IEEE Micro* 35, 3 (May-Jun 2015), 58–70.

[49] Y.-J. Yoon, N. Concer, and L. P. Carloni. 2013. Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance. *IEEE Transactions on CAD of Integrated Circuits and Systems* 32, 12 (Dec. 2013), 1906–1919.

[50] Y.-J. Yoon, P. Mantovani, and L. P. Carloni. 2017. System-Level Design of Networks-on-Chip for Heterogeneous Systems-on-Chip. In *Proc. of the International Symposium on Networks-on-Chip (NOCS)*. 1–6.

[51] F. Zaruba and L. Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration Systems* 27, 11 (Nov. 2019), 2629–2640.