Journal of Signal Processing Systems (2021) 93:545–563
https://doi.org/10.1007/s11265-021-01640-8

# The TaPaSCo Open-Source Toolflow

## for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems

Carsten Heinz[1] · Jaco Hofmann[1] · Jens Korinth[1] · Lukas Sommer[1] · Lukas Weber[1] · Andreas Koch[1]

Received: 28 April 2020 / Revised: 4 November 2020 / Accepted: 12 January 2021 / Published online: 2 May 2021

**Abstract**
The integration of FPGA-based accelerators into a complete heterogeneous system is a challenging task faced by many researchers and engineers, especially now that FPGAs enjoy increasing popularity as implementation platforms for efficient, application-specific accelerators for domains such as signal processing, machine learning and intelligent storage. To lighten the burden of system integration from the developers of accelerators, the open-source *TaPaSCo* framework presented in this work provides an automated toolflow for the construction of heterogeneous many-core architectures from custom processing elements, and a simple, uniform programming interface to utilize spatially distributed, parallel computation on FPGAs. TaPaSCo aims to increase the *scalability* and *portability* of FPGA designs through automated *design space exploration*, greatly simplifying the scaling of hardware designs and facilitating iterative growth and portability across FPGA devices and families. This work describes TaPaSCo with its primary design abstractions and shows how TaPaSCo addresses portability and extensibility of FPGA hardware designs for systems-on-chip. A study of successful projects using TaPaSCo shows its versatility and can serve as inspiration and reference for future users, with more details on the usage of TaPaSCo presented in an in-depth case study and a short overview of the workflow.

**Keywords** FPGA · Reconfigurable computing · Design space exploration · System-on-Chip design · Design automation · High-level synthesis · Scalability · Portability · TaPaSCo · Heterogeneous computing · Parallel computing · RISC-V

## 1 Introduction

Compared to modern software development methods it has been and still is very hard to achieve **scalability** and **portability** for FPGA-based solutions.

While microprocessor instruction set architectures have become commodity and are nowadays mostly interchangeable due to high-level software programming abstractions and powerful compilers, FPGA development is still very close to the metal. An FPGA design based on one generation of FPGAs from one vendor is often difficult to port to the next generation, let alone to an FPGA of a different vendor. Even using a larger device in the same FPGA family often poses hardware architecture design challenges that require considerable reengineering to efficiently utilize the resources available on the larger chip. To make matters worse, there are significantly fewer engineers trained in FPGA technology than software engineers – and software engineers are already hard to come by for many companies. From the perspective of small or medium-sized companies, these factors make FPGA development slow, difficult, and often render it entirely infeasible.

The FPGA community has made considerable progress to remedy these problems over the past two decades: The standardization of the synthesizable subsets of the most common hardware design languages, VHDL and (System-)Verilog, provides at least a minimal foundation for portability between logic synthesis tools and later place-and-route steps of the FPGA vendors. To further raise the level of abstraction, there is a huge body of research concerned with *high level synthesis (HLS)*: HLS approaches aim to allow the use of higher level languages for hardware construction, both using new languages and novel concepts, for instance, BlueSpec [7] or Cλash [5], as well as traditional software programming languages, such as, C/C++ [10, 32, 73], Java [23, 31], and more recent parallel programming models, such as OpenCL [35] or

✉ Carsten Heinz
heinz@esa.tu-darmstadt.de

Extended author information available on the last page of the article.

OpenMP [62]. Unfortunately, engineers trained in Bluespec or Chisel are even harder to find than ”regular” HDL developers, and none of the HLS approaches have yet reached a level of stability, maturity and performance on par with compilers in the software development world.

This paper presents *TaPaSCo, the Task Parallel Systems Composer*, an open source toolchain addressing these challenges. TaPaSCo consists of a scriptable toolflow for the automated construction of heterogeneous, many-core System-on-Chip hardware architectures, and a set of APIs to facilitate task parallel computing on TaPaSCo FPGA accelerator designs. However, it shall be noted that its main contribution is not in either of these fields — instead, TaPaSCo aims to harness and amplify the power of existing tools and approaches by providing the missing *glue* between state of the art HLS tools and modern parallel computing paradigms and languages: It allows the designer of FPGA accelerators to raise their level of abstraction and disregard many specific features of the target FPGA by delegation of optimizing these choices to TaPaSCo's automated *design space exploration*. TaPaSCo simplifies re-targeting existing designs to new FPGAs and boards without requiring changes to the accelerators themselves. Furthermore, this allows to postpone the selection of a concrete target technology until much later in the design process. TaPaSCo's APIs complete the picture by providing the necessary foundations to implement higher-level runtimes (e.g., OpenCL, OpenMP) for platform-agnostic application software.

The rest of this paper is organized as follows: Section 2 contains a brief survey of related work, Section 3 introduces TaPaSCo and its primary design abstractions. Specifically, showing how TaPaSCo addresses portability, scalability and extensibility of FPGA hardware designs for systems-on-chip. The tool is actively used in a variety of projects, some of which are discussed in Section 4. The process of including RISC-V softcores, which are originally not designed to be used with the tool, as a PE in a TaPaSCo design, is presented in Section 5, including the actual commands for using the tool. The paper closes with an overview of the active development of the open-source tool TaPaSCo in Section 6 and concludes in Section 7.

## 2 Related Work

With the availability of VLSI FPGA designs in the 1990s came an interest in reconfigurable, general purpose computing on such architectures. Programmability was quickly identified to be one of the core bottlenecks for wide-spread adoption: FPGA programming required a highly specialized skill set at the intersection of electrical engineering and computer science, which was hard to find. The fact that the design methodologies involved in designing computing architectures to be executed on an FPGA were intended for traditional chip production (cf. [22]), with slow iteration cycles and long turn-arounds, did not help either. These problems started the research into *High-Level Synthesis (HLS)* tools to synthesize behaviorally equivalent hardware from high-level programming languages (such as C, C++, or domain-specific languages). Obviously, solving the data path synthesis problem is only half a solution: In order to provide convenient access to the generated accelerators, one needs to carefully integrate the accelerators into the overall system architecture, with high-speed access to memories, communication to the host CPU, and other basic infrastructure, which has a significant impact on the overall performance of the system.

### 2.1 Earlier Works

Many early works were therefore concerned with solving the problem of cache-coherent memory accesses [45, 71], and on-chip bus topologies to support reconfiguration of accelerators in predefined slots [41, 47]. Integration of custom accelerators into the host system as *hardware threads* was first proposed in [55], then extended upon in BORPH [61], ReconOS [47], FUSE [36] and LEAP [19]. Being the other half of the HLS solution to the problem sketched above, these works appeared in close relation to the growing maturity in academic and commercial HLS compilers such as Garp [9], Streams-C [20, 24], ROCCC [25], Comrade [21], Nymble [32], Bambu [56], LegUp [11] and Vivado HLS [73]. Nane et al. gave an excellent overview of the vast landscape of tools available by 2015 [50].

The work described in this paper builds upon this area of research and aims to further remove entry barriers to FPGA computing, and give researchers a fast-track to generate high performance accelerator architectures. It is, however, not focused on HLS itself; instead, it was initiated to address common problems occurring when employing these tools: When trying to assess the performance of HLS tools, one can either stop in *simulation* at the cycle count level (often using far-from-realistic assumptions about the behavior of memory in a real system), or perform the experiments on *real hardware*. The latter, however, requires one to implement the entire hardware and software stack required to run the experiments. Not only is this approach tedious and error-prone, but most importantly, the impact of the system design on overall performance and characteristics greatly reduces the comparability of different implementations.

This problem is precisely what motivated the work on ThreadPoolComposer [42], the authors' first effort in this area. TaPaSCo is based on ThreadPoolComposer, which is in turn is closely related to previous work on ReconOS [47], hthreads [54], and FUSE [36]. ThreadPoolComposer aimed to provide both programming and hardware abstractions to increase FPGA developer productivity. But unlike the other approaches, ThreadPoolComposer focused on typical high-performance computing systems using a mainstream, non-modified Linux kernel, and catering for commercial (OpenCL, OpenMP, X10 [12, 14]) and academic (FastFlow [2]) parallel programming frameworks.

## 2.2 Recent Works

In [28], ThreadPoolComposer was extended with automated design space exploration capabilities to increase scalability of the designs even further. TaPaSCo extends this significantly by providing a fully asynchronous job launch interface, support for a memory hierarchy of device-local and *Processing Element (PE)*-local memories, a unified kernel module interface, and offering support for a wide range of FPGA families from small embedded to high performance segments with PCIe Gen3/4-based data transfers (currently supported boards: Digilent ZedBoard, Digilent PyNQ, AVNET Ultra96, Xilinx ZC706, Xilinx ZCU102 UltraScale+ MPSoC, Xilinx VC709, Xilinx VCU118, NetFPGA SUME, Xilinx Alveo U250/U280). The addition of PE-local memories draws from scratchpad approaches (cf. [1, 19, 76]) and was essential for later work, e.g., an evaluation and survey of FPGA-compatible RISC-V cores in [27].

Both ThreadPoolComposer and TaPaSCo aim to exploit the optimization opportunities enabled by processing *complete* SoC designs for full reconfiguration, instead of using partial reconfiguration at runtime, an area in which ReconOS [47] excels: The ReCoBus-Builder generates a hot-pluggable bus architecture in which accelerators can be attached and detached at runtime, establish point-to-point connections with memories or other accelerators (similar to memory-mapped AMBA AXI protocols), and use local and shared scratchpad memories. A similar approach is taken by the ARTICo$^3$ framework [58], which focuses on compute intensive applications on distributed sensor networks and cyber-physical systems. Like ReconOS, it also supports partial reconfiguration at runtime to enable hot-swapping of accelerators, but aims at applications often performed by DSPs, which are typically absent in edge computing devices. The ARTICo$^3$ scheduling model is based on vectorization and instruction level parallelism: it allows to distribute a workload stream to a dynamic number of accelerators at runtime, utilizing both pipelining and parallel execution of the accelerators. For edge-computing devices memory is at a premium, which makes partial reconfiguration an excellent technique. Partial bitstreams are significantly smaller than full bitstreams, allowing the device to store more accelerator types and adapt dynamically to the runtime compute load. However, partial reconfiguration also has several important disadvantages. Since the partially reconfigurable resources are locked in place on the FPGA, it places severe constraints on the place-and-route stage, which results in sub-optimal area utilization and less re-use compared to complete-design workflows. For the same reason, it is often difficult to achieve high operating frequencies across the reconfiguration borders. Furthermore, it is challenging to accommodate truly heterogeneous accelerators which may have a large variation in accelerator sizes, since all accelerators have to fit in the same reconfigurable space/area/slot. In contrast, TaPaSCo aims at high-performance computing workloads, where storage space for the configurations is not limited in practice, but where optimization of the overall clock frequency is paramount, making partial reconfiguration a less attractive feature.

In [72], Wenzel et al. make a case for moving the tradeoff between synthesis time and runtime in the opposite direction: RapidSoC aims at speeding-up turnaround times for the synthesis of softcore-based SoCs, at the expense of reduced operating frequency. While not suitable for high-performance computing, this approach allows to employ custom-tailored computing solutions for a much wider scale of applications than would otherwise be economically feasible.

The complex tradeoffs involved in every FPGA design are examined further in [49] where the authors evaluate different time and space sharing strategies to deploy FPGAs as compute resources in cloud computing setups in the most efficient ways possible. In this scenario, the FPGA's utilization as a compute resource can be optimized by sharing the device among multiple distinct users at the same time. TaPaSCo also aims to accommodate truly heterogeneous workloads, but the underlying assumption is that they belong to a single application/user; while varying workloads in different phases of the application can still be covered by multiple TaPaSCo designs and full reconfiguration, it is assumed that the overall workload structure is known at synthesis time.

Redsharc [59] is an academic hardware/software system design framework with a similar approach as TaPaSCo; it shares concepts such as the grouping of heterogeneous PEs into clusters, and uniform, scriptable construction of cluster groups into architectures. However, the Redsharc source is not publicly available, is not portable and does not support current hardware. Furthermore, Redsharc is focused on hardware architectures processing regular data streams, whereas TaPaSCo explicitly supports more general hardware that also allows random-memory accesses.

TaPaSCo generates computing architectures for the task parallel model of computation by instantiating multiple, independent instances of accelerator cores. A similar approach is found in the ARACompiler [13], which only runs on the Xilinx ZC706 SoC board, but supports accelerator-side paging in interaction with a Linux OS running on the on-chip ARM CPU.

Whereas TaPaSCo supports both HLS and HDLs to define the accelerator cores, CMOST [78] generates a full system architecture from a C program by revisiting approaches found in HLS compilers [9, 11, 21]: It extends the loop unrolling and extraction techniques based on the polyhedral model to task based parallel computing, with similar split into a hardware-dependent and a hardware-independent part of the generated designs, support for SoCs and some PCIe-based systems (e.g., Xilinx VC707). Due to this approach, CMOST is limited to C programs and precludes many parallelization opportunities on the host side.

The topic of *Design Space Exploration (DSE)* is a cross-cutting concern in all FPGA research; the approach implemented in TaPaSCo is tuned toward this specific application, but precursors and similar ideas can be found throughout the literature. Recently, Lo and Chow explored the impact of high-level synthesis directives in general using a DSE approach [46], whereas Xu et al. implemented a parallel auto-tuning algorithm [75], similar to the batch approach in TaPaSCo. Sotiriou-Xanthopolous et al. presented a DSE approach for virtual platform simulators in [67], which allows to simulate different design variations from a single bitstream and estimate performance w.r.t. to energy efficiency, performance, and area consumption. The estimation method is similar to TaPaSCo's out-of-context synthesis, but the focus of the overall framework is on the acceleration of virtual platform simulation for ASIC designs.

An approach that is more similar to TaPaSCo than the previously described solutions is the relatively new ESP platform [48]. Like TaPaSCo it enables the composition of SoCs from different sources. For example, ESP allows the use of SystemVerilog, VHDL and Chisel for accelerator design using HDL. Apart from those, C/C++ and SystemC are supported for HLS-based accelerator design. Lastly, ESP also allows the automatic generation of accelerators from machine learning frameworks, such as Keras, Tensorflow or PyTorch. The models are transformed into accelerators using the hls4ml-framework [16]. All resulting accelerator designs are stored in an IP-library, which can then be used for rapidly prototyping SoCs. It is also possible to include processor cores within the overall SoC-design, which can then be programmed in software. Just like TaPaSCo, ESP provides a software stack that enables interfacing with the corresponding accelerators. Specific drivers for communicating with the different accelerators are automatically generated from provided templates. While portability is also a goal for ESP, it currently only supports the Xilinx VC707, VCU118/128, as well as the Virtex7/Virtex Ultrascale-based proFPGA boards. In comparison, TaPaSCo supports similar devices, but also includes entry-level boards such as the Digilent PyNQ/ZedBoard.

Another comparable work is the open-source project Connectal [40], which automatically generates hardware/software interfaces to FPGA-based accelerators. Using a BlueSpec Verilog (BSV)-based interface declaration, Connectal automatically creates a BSV-based wrapper for the accelerator, as well as a C++ wrapper that allows communicating with the accelerator from user-space. Connectal also works platform-independently and currently supports a number of Xilinx Zynq/Virtex7-based boards (Digilent ZedBoard/Zybo, Xilinx ZC702/706, VC707/709 and NetFPGA SUME).

With regards to the automatic generation of SoC-based multi-core RISC-V systems (cf. Section 5), Chipyard [3] offers functionality similar to TaPaSCo. It allows the integration of several RISC-V processor cores within a single SoC. In addition to the supported RISC-V cores (BOOM, Ariane [77] and Rocket [4]), accelerators can also be included in the designs. Most notably, the Nvidia Deep Learning Accelerator (NVDLA) is also supported. Chipyard leverages the Chisel3 HDL [6] for composition, simulation, and integration on a target FPGA. With this approach, SoC simulation can also be performed using FireSim [39], which enables FPGA-accelerated simulation of the resulting design using cloud resources provided by Amazon AWS F1 instances. With regard to the system composition, a similar approach is used by the HERO framework for research in RISC-V-based manycore SoCs [44].

## 2.3 Commercial Tools

High-level synthesis (HLS) tools are a major enabler for making FPGA-based reconfigurable computing accessible to developers with a software background. A first major push in commercial HLS tools for FPGAs occurred in 2010 with Xilinx's FPGA-vendor supported tool *Vivado HLS*. This tool has formed a key pillar of our design flow since the early days of ThreadPoolComposer, with TaPaSCo further automating some steps of the HLS process for the convenience of the user.

Building on Vivado HLS, Xilinx introduced FPGA vendor-supported *SoC* design tools in the forms of SDSoC and SDAccel, in 2015 and 2016, respectively, which bear some similarity to TaPaSCo. Both also allowed to compose SoC designs from individual kernels, but each was specialized for different domains and FPGA platforms. In contrast to TaPaSCo, no single tool could be used to target both embedded platforms (e.g., Zynq-7000, Zynq MPSoC) and data-center FPGAs (e.g., Virtex 7, UltraScale+).

SDSoC and SDAccel were later unified by Xilinx, with the addition of other components such as the open-source Xilinx Runtime (XRT), in the *Vitis* tool [74], which was announced in 2019. Vitis adopted a similar design methodology to the one used by TaPaSCo: Both tools support the kernel implementation in either a hardware description language, such as Verilog or VHDL, or in C/C++-based HLS, and both approaches allow for the combination of kernels developed with either one of those approaches. With the `v++` link command, Vitis allows to combine multiple instances of different kernels into a single design, and automates the connection with the hardware platform. This is similar to TaPaSCo's `compose` command (see Section 5 for more details). While TaPaSCo focuses on and optimizes for an architecture where kernels may access the FPGA memory randomly, and where all data is transferred via memory, Vitis also supports streaming kernels.

For both Vitis and TaPaSCo, the host software is usually developed in C/C++, using the OpenCL or XRT API for Vitis, or the TaPaSCo API for TaPaSCo.

With Vitis, commercial tool development has caught up with the approach followed by ThreadPoolComposer/TaPaSCo since 2015. But even so, TaPaSCo still maintains a lead over Vitis with its automated design-space exploration capabilities, and the flexibility provided by TaPaSCo's powerful plug-in extension mechanism.

With the plug-in and feature mechanism implemented in the platform scripts, the hardware platform (sometimes also called *shell* or *base design*), into which kernels are automatically integrated, can easily be customized by enabling or disabling features or plugins. This, for example, allows to investigate the impact of a cache system on the current design by simply enabling the cache feature.

With its open-source nature, the TaPaSCo hardware platforms can easily be customized by extending the existing platform scripts, e.g., to add networking interfaces, as was done in [30], black-box extension of existing cores with caches or queues, using infrastructure cores to change the interconnect (e.g., switching between buses and networks-on-chip) between processing elements, and the use of interface adapters to communicate with third-party IPs. TaPaSCo's customizability is a key capability to enable high performance for a diverse range of computing approaches.

For more complex use-cases, deciding on a good SoC architecture while considering multi-dimensional trade-offs, e.g., between FPGA area, clock frequency, throughput etc. is a difficult problem. For example, in many cases it will not be obvious how many instances of a kernel should be placed on a specific FPGA platform, or what the optimal operating frequency of a design is for achieving maximal throughput. TaPaSCo's automatic design space exploration allows to automatically investigate multiple SoC *architectures* at different trade-off design points (see Section 3.4 for more details). The design-space exploration capabilities in Vitis are currently limited to examining different implementation strategies, e.g., for mapping, placement, and routing, for a given SoC architecture.

Intel, the second major FPGA vendor, on the other hand provides the Open Programmable Acceleration Engine (OPAE) [34]. However, OPAE is solely a software runtime to facilitate interfacing with accelerators. The actual FPGA bitstreams still need to be generated by traditional development tools from Intel, where the SoC composition is not automated by OPAE. Intel's Quartus tool provides some design-space-exploration capabilities, however only in a single domain and for a single kernel.

## 3 TaPaSCo

TaPaSCo consists of two main parts: An automated toolflow to generate System-on-Chip (SoC) designs based on custom processing elements (PEs, e.g., described in Verilog/VHDL, Bluespec, Chisel, or generated using HLS), and a general application programming interface (API) and accompanying libraries to facilitate platform-agnostic software development. In the following, Section 3.1 will focus on the former, Section 3.2 on the latter. Section 3.3, Section 3.4 and Section 3.5 describe how TaPaSCo addresses the central issues of portability, scalability and extensibility for future proofing FPGA designs.

### 3.1 Hardware Design Abstractions

TaPaSCo hardware designs as shown in Fig. 1 consist of a configurable number of *processing elements (PEs)*; PEs of the same *kind* are grouped into PE *clusters*, which are in turn grouped into the *Architecture* of the design. Finally, the *Platform* instantiates board- or FPGA-specific resources to implement data and control accesses, as well as signaling, thus leading to the complete system shown in Fig. 2.

TaPaSCo hardware designs are based on three fundamental abstractions (ordered by scope): **A1:** T-model of processing elements, **A2:** Architecture, and **A3:** Platform.

Each of the abstractions is implemented as a set of *scripts* in TaPaSCo: **A1** consists of scripts to configure the interface generation of supported HLS compilers. **A2** is implemented in a modular Tcl script to perform the wiring of *PEs* into *clusters*, and *clusters* into an *Architecture*, using suitable bus topologies. The fundamental idea is to keep the *Architecture* independent of the target FPGA, making it

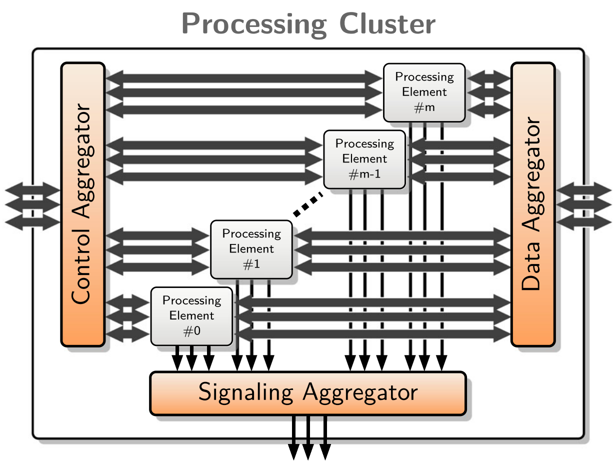


**Figure 1** Processing elements of same kind are grouped into *clusters* using channel aggregators for the three basic channels, e.g., AXI4 Interconnects and interrupt controllers.

reusable across targets. **A3** finally connects the *Architecture* to the surrounding hardware components of the target FPGA board. The scripts currently utilize the Vivado Tcl APIs to automate the wiring of high pin-count interfaces (e.g., AXI4).

The *T-model*, named for its T-shape shown in Fig. 3, defines the interface requirements for a TaPaSCo PE module and abstracts from implementation details. Each PE in the T-model requires three basic channels: 1. a control channel to communicate with the host, 2. a signaling channel to indicate completion, and 3. a data channel to access data.

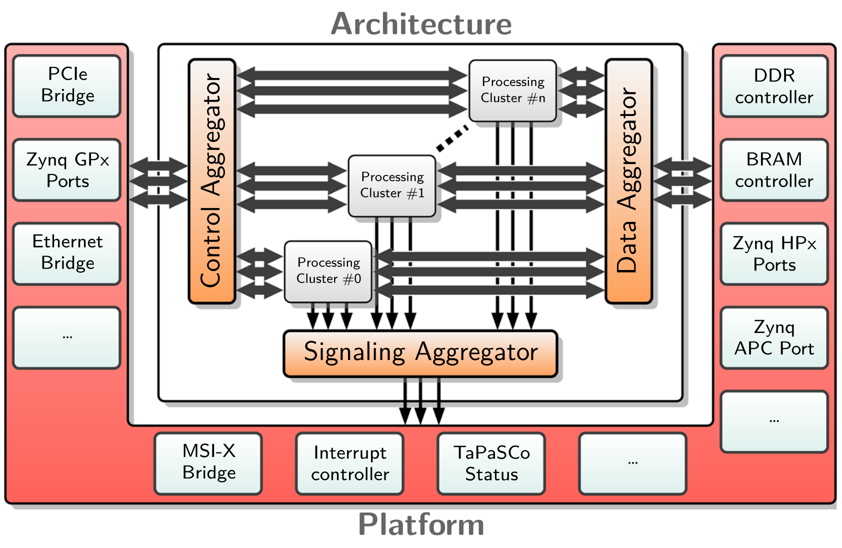
The exact nature of the channels (e.g., AXI4, Avalon, Wishbone, NoC, ...) is determined by **A2**, the *Architecture*: TaPaSCo supports heterogeneous PE architectures, i.e., groups of different PE kinds scaling linearly. To achieve this, PEs are grouped into PE *clusters*, each *cluster* containing all PEs of a kind and abstracting away the concrete number of individual PEs. The T-shape is repeated in a fractal-like fashion here: Each *cluster* itself is T-shaped, and can be wired like the PEs themselves (see Fig. 1).

On the outermost platform-independent level, this process is repeated across *clusters*. The collective term used to describe the automated wiring of all three levels is *Architecture*, i.e., the organization and wiring of PEs into a heterogeneous pool as shown in Fig. 2. In TaPaSCo, *Architectures* are designed to be platform-agnostic: Whatever protocol or technique is used to actually perform the wiring, this part of the design should remain portable.

TaPaSCo currently uses an *Architecture* based on AMBA AXI4: All control interfaces are AXI4Lite slaves, all memory interfaces are AXI4 masters, signaling is done via a single wire interrupt line. AXI4 Interconnects are used for both slave and master interfaces at the *cluster* and *Architecture* levels to wire the connections. Note that TaPaSCo's blackbox approach regarding the PE internals is suitable to support many different compute architectures: The AXI-based architecture has been used both in the random-access architecture discussed in Section 5 for near-data processing, as well as for a complex high-performance Stereovision accelerator based on a systolic array [28]. It would also be entirely possible to use TaPaSCo to connect only a single PE (containing a full architecture inside), see Section 4 for an example. TaPaSCo does not impose a model on the PE, it only focuses on providing easy spatial replication and convenient access from software.

The outermost abstraction is called the *Platform*: All parts of the hardware design which are specific to the

**Figure 2** Using the T-model to hierarchically repeat the automated wiring process to connect the individual clusters into a heterogeneous pool architecture. The availability and presence of components in the *Platform* layer depends on the concrete target FPGA and board; the *Architecture* is independent of their presence or availability, making the essential design portable.



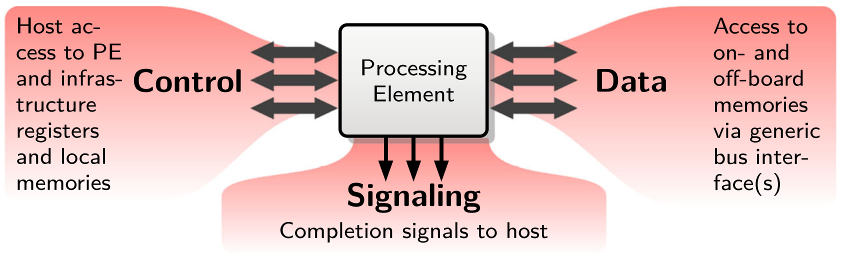

Host access to PE and infrastructure registers and local memories
Control
Processing Element
Data
Access to on- and off-board memories via generic bus interface(s)
Signaling
Completion signals to host

**Figure 3** Basic *T-shape* of processing elements: Each processing element has a *control channel*, a *data channel* and a *signaling channel*, all of which can be implemented by arbitrary means, e.g., AXI4, or Avalon.

target board (e.g., the FPGA, pin constraints, peripherals, memory) are generated by the *Platform* abstraction. Minimally, a *Platform* must connect the control interfaces to the host, provide some memory shared between PEs and the host, and an interface toward the host for PE signals. Furthermore, all peripherals and other infrastructure are instantiated here (e.g., memory controllers, interrupt controllers). *Platform* scripts can be seen as *smart base designs*: They instantiate target-specific infrastructure, but retain a significant amount of configurability without requiring manual intervention by the user.

Originally, both ThreadPoolComposer and TaPaSCo used a fixed address map scheme to facilitate communication between host and PEs. Now TaPaSCo solves this more elegantly by storing the on-chip address map in a custom hardware module generated *on the fly* during composition. This address map is then queried at runtime by the software layers. This approach yields great flexibility: E.g., every kind of PE may have a different number and/or differently sized control interfaces. It is also possible to integrate custom, user-defined infrastructure modules and use the TaPaSCo software layers to communicate with them: The TaPaSCo scripts for *Architecture* and *Platform* are skeletons providing numerous injection points for extensions, where *plug-ins* can be inserted to modify the design in flight.

The goal of TaPaSCo is to provide a framework that allows for fast SoC-generation for a wide range of FPGA platforms. By offering flexibility with regard to the size of the control interface (e.g., the number of writable registers), TaPaSCo allows users to easily use the tool-flow without the need for additional wrapping logic around the PEs. In the case of the configuration interfaces, this generalization does not have a measurable performance impact, as TaPaSCo will only write the registers actually requested by the user for each PE.

*Example 1* If a PE does not have a TaPaSCo-compatible register interface (see [57] for a more detailed description of the register conventions used by TaPaSCo), a plug-in can automatically instantiate a suitable wrapper to enable TaPaSCo to continue with the automated wiring.

A different example can be found in the `zedboard` *Platform*: The Digilent ZedBoard [15] has an on-board OLED display that can be used to show the number of completion signals at each slot; this is achieved by a plug-in that instantiates the corresponding display controller and wires it to the design. Such modifications are common, especially when exploring different variations of a design, e.g., using different DMA engines. To simplify the use of such plug-ins, TaPaSCo provides support for so called *features*: Features can be defined using a simple, but consistent key-value syntax and can be queried by plug-ins during composition. This allows the user to easily pass configuration values, and enable or disable specific plug-ins.

The hardware design abstractions described in this section enable TaPaSCo to provide a fully automated toolflow for composing hardware designs, which is one of the most important features of TaPaSCo. The TaPaSCo hardware design flow is depicted in Fig. 4 and shows how a complete design can be assembled from the individual PEs. For HLS-based PEs, TaPaSCo will first execute the vendor high-level synthesis (e.g., Vivado HLS), using the C/C++ sources provided by the user. For HDL-based PEs or softcores, this step is not required, instead they are expected to be provided in the IP-XACT format produced by many hardware design tools (e.g., Vivado).

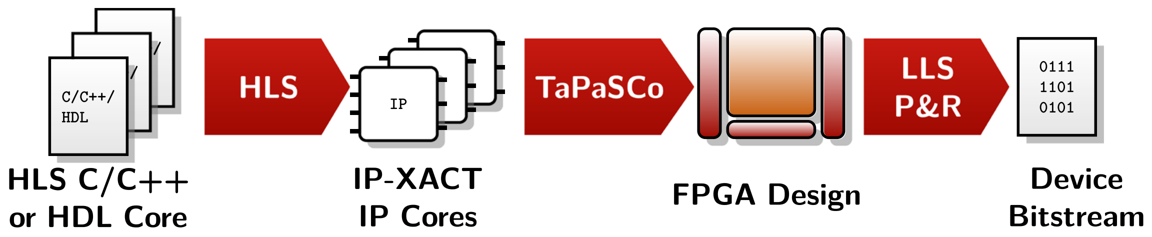


**Figure 4** TaPaSCo hardware design flow, producing a complete FPGA bitstream from the individual user-provided PE definitions.

After the user has specified the desired composition of the hardware design, i.e., the architecture, the kind of PEs, number of PEs per kind, operating frequency and (optionally) target platform, TaPaSCo will create a complete FPGA SoC-design, using the hardware abstractions and mechanisms described in this section. Eventually, TaPaSCo will invoke the vendor's synthesis tool (e.g., Vivado) to perform synthesis, placement and routing for the target FPGA, yielding the final bitstream that can be used to program the FPGA. The entire design process can often be initiated with a single command, as we will demonstrate in the example in Section 5.3.

## 3.2 Software Design Abstractions

Key to providing a productive environment for FPGA developers is to eliminate as many manual tasks as possible that are not directly related to the problem at hand. Specifically, this includes handling low-level communications with the hardware. Using an automated process, as described in Section 3.1, to construct hardware designs has the benefit of yielding very regular designs, which can consistently be used in software without requiring repetitive manual protocol implementations.

The core abstraction for the *application programming interface (API)* of TaPaSCo is the *task-parallel model*: Every computation is broken into *tasks*, which can execute in parallel. Each work item of a task is split into a number of individual *jobs*, each of which can be computed independently. This model is widely used in heterogeneous computing, because it accommodates different computing architectures by abstracting computation from concrete algorithm: The user submits jobs to the abstract machine, which are then processed by any of its available PEs, regardless of their internals. Even the original interface defined by TaPaSCo's predecessor ThreadPoolComposer was already sufficiently portable to also support execution on *digital signal processors* (DSPs), without having to change the host code (cf. [57]).

In TaPaSCo's software framework, a task corresponds to a *cluster*, and a job corresponds to one execution of a single PE. At this granularity, a domain expert can develop the core application by defining tasks and splitting work items into jobs; this is the top-most, user-facing API that TaPaSCo defines (for a concrete usage example see Section 5).

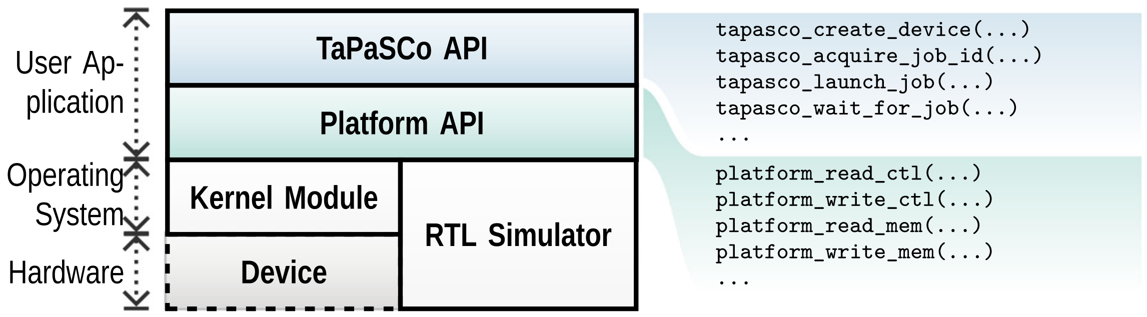
To implement this rather abstract API, TaPaSCo internally mirrors the abstractions of the hardware design (see Fig. 5): The *TaPaSCo library* is concerned with the *Architecture*. It manages PEs and the address map, performs the communication required to transfer data and arguments, launch a job, and wait for the result(s).

In order to implement the interactions in a platform-agnostic manner, the TaPaSCo library is implemented on top of the *platform library*, which encodes primitive operations, such as accessing a PE's registers, or allocate/free and read/write device-accessible memory. This allows any *Architecture* to be used on any *Platform* with the same user application code.

The platform library operations themselves are realized using an operating system layer implemented in the *TaPaSCo loadable kernel module (TLKM)*: Without going into unnecessary details, TaPaSCo uses a fixed set of `ioctl` commands, which need to be implemented at most once for each *Platform* (often code can even be shared among families of devices). The commands are sufficiently generic to accommodate a wide variety of transport mechanisms, from shared memory (e.g., Zynq, MPSoC) to PCIe Gen3 (e.g., VC709). Please see the documentation at [68] for more details on the internal APIs.

## 3.3 Portability

The overall approach outlined in Sections 3.1 and 3.2 has proven to be very useful to isolate the domain expert (i.e., the application developer) from the details of the chosen target platform: A TaPaSCo application's code does not need to be changed when executing on a different TaPaSCo platform.



```
tapasco_create_device(...)
tapasco_acquire_job_id(...)
tapasco_launch_job(...)
tapasco_wait_for_job(...)
...
platform_read_ctl(...)
platform_write_ctl(...)
platform_read_mem(...)
platform_write_mem(...)
...
```

**Figure 5** Software Layer Hierarchy in TaPaSCo: The top-level API provides task-parallel abstraction, the *Platform API* realizes a thin user-space layer above either **a** the operating system primitives implemented in the *TaPaSCo Loadable Kernel Module (TLKM)*, which in turn interacts directly with the device(s), or, alternatively, **b** interfaces with a RTL simulator of the hardware design.

This also applies to the hardware level: If a hardware module conforms to the TaPaSCo interface requirements, it can be used on any supported platform. Furthermore, TaPaSCo was designed to be easily extensible to new platforms: Within one year of TaPaSCo's initial presentation in [43], the number of supported platforms has doubled from seven to fourteen. The list of of supported platforms ranges from small embedded boards using Zynq devices, up to high-performance PCIe-based expansion cards with large UltraScale+ FPGA devices.

## 3.4 Scalability

Scaling a TaPaSCo design, e.g., from using five PEs of a certain kind to 30 PEs, requires only automated rebuilding of the hardware design via TaPaSCo. Everything else, including the application code, does not need to be changed and will adapt automatically to the new design.

Furthermore, additional support for *design space exploration* in TaPaSCo simplifies a crucial task in optimization: When designing an SoC with a large number of PEs, there will always be a trade-off between the number of PEs in the design and its operating frequency. More PEs means more potential for spatial parallelism and better area utilization; however, with increasing area, path lengths in the design also increase, making timing closure increasingly more difficult to achieve. Finding a good trade-off for any given application can be a very tedious and slow trial-and-error process.

TaPaSCo supports the user by providing an *automated design space exploration (DSE)* along the three axes of operating frequency, area utilization, and use of design variants. Each axis can be separately activated or deactivated in a DSE run, e.g., to determine only the highest operating frequency for a fixed number of PEs, or to find the maximal number of PEs that will fit on a given device at a fixed operating frequency.

The algorithm first computes upper and lower bounds for each activated axis. For the operating frequency, TaPaSCo uses an *out-of-context synthesis run* (abbreviated as OOC here) to perform a full place-and-route on an otherwise empty target FPGA. Since this design is almost unconstrained, this yields an overly optimistic approximation of the achievable operating frequency. The lower bound is usually determined by the target FPGA; by default, TaPaSCo cuts off at 50 MHz, discarding compositions with a lower operating frequency.[1] The remaining interval is then divided evenly in 5 MHz steps by default, each step being the frequency component of a coordinate in the discrete design space.

Bounds for area utilization are also based on out-of-context synthesis: OOC yields an estimate of the area used by each kind of PE. The area utilization for the entire design is then estimated using a stoichiometric linear extrapolation based on the number of PEs of each kind and an estimation for the architectural overhead. By default, TaPaSCo assumes zero overhead, making a very optimistic approximation. This is justified, as modern place-and-route tools perform very extensive optimizations and can compact similar circuits very aggressively, sometimes yielding lower values for area utilization than the linear extrapolation would suggest. Since these optimization efforts are very hard to estimate a-priori for any given design, TaPaSCo compensates by using an optimistic approximation of the design overhead instead, to avoid discarding viable designs prematurely. To increase or decrease the area utilization, the initial composition is scaled linearly in the number of PEs. This yields the area component of the design space coordinates.

Design *variants* specified by the user represent different implementations of the same PE kind, e.g., using more Block RAM, or more pipeline stages, or different sizes of FIFOs. For each *cluster*, a single variant is chosen; the design variants are then generated combinatorially by combining with every variant of every other PE kind in the composition. This yields the choice of a design variant as the third coordinate component within the design space.

Due to combinatorial explosion, the size of the design space quickly exceeds the limits for brute force exploration. To counter this, TaPaSCo supports different heuristic functions to evaluate each element in the design space, and then explores batches of elements ordered by their score. At each step, the design space is pruned, e.g., discarding the elements which have a lesser score than the best element found so far.

Overall, the runtime of the automatic design-space exploration is still highly dependent on the user design, target platform and number of dimensions to explore, and can vary between a few hours and several days. To accelerate this process, TaPaSCo supports *parallel* investigation of multiple design-points (the number can be specified by the user) in a high-performance computing cluster-setup, employing the Slurm Workload Manager [60]. This feature was used for the extensive design-space exploration in [28]. The design points found by the TaPaSCo DSE that yielded the highest framerates for that application were non-obvious solutions that would have been difficult to choose manually even for experienced hardware designers.

*Example 2* Assume the user specifies an initial composition consisting of three different PE kinds, called `A`, `B` and `C`,

[1]The 50 MHz cut-off point has been chosen based on the application scenarios investigated so far. However, thanks to TaPaSCo's extensible design, this could easily be changed in the implementation and could be customized on a per-platform basis.

with two PEs in the `A` *cluster*, four PEs in the `B` *cluster*, and six PEs in the `C` *cluster*. In TaPaSCo syntax this would be expressed as `[A x 2, B x 4, C x 6]`; in the following, such a configuration is called a *composition*. When scaling linearly, the smallest composition with the same ratios containing all PEs is thus `[A x 1, B x 2, C x 3]`. Also assume that TaPaSCo has determined via OOC that the largest composition fitting on the target FPGA is `[A x 3, B x 6, C x 9]`. This would yield three viable compositions in the design space. Furthermore assume that the OOC for `A` has given us an $f_{max}$ of 100 MHz, 75 MHz for `B` and 150 MHz for `C`. Since all PEs are clocked at the same frequency, `B` provides the upper bound on frequency at 75 MHz. Leaving the lower cut-off at the 50 MHz default yields six frequency coordinates: 50 MHz, 55 MHz, 60 MHz, 65 MHz, 70 MHz and 75 MHz. Finally, assume that only `A` has variants, called `A0` and `A1`. Thus, the design space TaPaSCo will explore will contain a total of 36 elements (listed in Table 1).

Details of the actual DSE algorithm, including the heuristics used for pruning the search space, have been presented in [29].

### 3.5 Extensibility

Given the vast variety of scenarios in which FPGAs are often used, it is impossible for a generic toolchain such as TaPaSCo to anticipate and support every use-case out of the box. Instead of a one-size-fits-all approach, TaPaSCo aims for a high degree of modularity and extensibility in all parts and aspects. Using plug-ins and features to customize the hardware design generated by TaPaSCo has already been discussed in Section 3.1. Adding new *Platform*s or *Architecture*s is very easy as well. But one of the core goals of TaPaSCo is to provide a re-usable foundation for further work and to eliminate some of the tedious work for every prototyping engineer or researcher. Therefore, most parts of TaPaSCo are designed in a modular fashion and allow for their standalone usage.

*Example 3* Some people may not be interested in using the task-parallel abstractions provided by the TaPaSCo API, but would still like to use the rest of the toolchain to iterate their designs more quickly; in this case, the *Platform* API can be used on its own to directly interact with the hardware. For others, the TaPaSCo API may not be sufficiently abstract yet; in this case, TaPaSCo can be used as a *foundation* for implementing more complex environments and frameworks, such as OpenMP (cf. [62]), or OpenCL.

## 4 TaPaSCo Success Stories

TaPaSCo and its direct predecessor *ThreadPoolComposer* (TPC) [42] have successfully been used for a number of applications. As TaPaSCo is available on a broad range of different platforms, from small, power-efficient boards to large, datacenter-scale devices, the application scenarios of TaPaSCo are likewise very diverse, ranging from embedded computer vision to high-performance computing (HPC), distributed data analytics, and machine learning.

This section presents some of the concrete applications using TaPaSCo and describes how these applications exploit the *scalability* and *portability* provided by TaPaSCo, as well as other, more specific features. The description in this section is intended as a reference and also inspiration

**Table 1** Initial design space for TaPaSCo DSE run.

| F \ R | 1 | 2 | 3 |
|---|---|---|---|
| 50 | $[A0 \times 1, B \times 2, C \times 3]$ | $[A0 \times 2, B \times 4, C \times 6]$ | $[A0 \times 3, B \times 6, C \times 9]$ |
| | $[A1 \times 1, B \times 2, C \times 3]$ | $[A1 \times 2, B \times 4, C \times 6]$ | $[A1 \times 3, B \times 6, C \times 9]$ |
| 55 | $[A0 \times 1, B \times 2, C \times 3]$ | $[A0 \times 2, B \times 4, C \times 6]$ | $[A0 \times 3, B \times 6, C \times 9]$ |
| | $[A1 \times 1, B \times 2, C \times 3]$ | $[A1 \times 2, B \times 4, C \times 6]$ | $[A1 \times 3, B \times 6, C \times 9]$ |
| 60 | $[A0 \times 1, B \times 2, C \times 3]$ | $[A0 \times 2, B \times 4, C \times 6]$ | $[A0 \times 3, B \times 6, C \times 9]$ |
| | $[A1 \times 1, B \times 2, C \times 3]$ | $[A1 \times 2, B \times 4, C \times 6]$ | $[A1 \times 3, B \times 6, C \times 9]$ |
| 65 | $[A0 \times 1, B \times 2, C \times 3]$ | $[A0 \times 2, B \times 4, C \times 6]$ | $[A0 \times 3, B \times 6, C \times 9]$ |
| | $[A1 \times 1, B \times 2, C \times 3]$ | $[A1 \times 2, B \times 4, C \times 6]$ | $[A1 \times 3, B \times 6, C \times 9]$ |
| 70 | $[A0 \times 1, B \times 2, C \times 3]$ | $[A0 \times 2, B \times 4, C \times 6]$ | $[A0 \times 3, B \times 6, C \times 9]$ |
| | $[A1 \times 1, B \times 2, C \times 3]$ | $[A1 \times 2, B \times 4, C \times 6]$ | $[A1 \times 3, B \times 6, C \times 9]$ |
| 75 | $[A0 \times 1, B \times 2, C \times 3]$ | $[A0 \times 2, B \times 4, C \times 6]$ | $[A0 \times 3, B \times 6, C \times 9]$ |
| | $[A1 \times 1, B \times 2, C \times 3]$ | $[A1 \times 2, B \times 4, C \times 6]$ | $[A1 \times 3, B \times 6, C \times 9]$ |

F = Target Design Frequency, R = Replication Factor.

for future users of TaPaSCo, detailing how to use some of TaPaSCo's unique features.

### 4.1 Integration of High-Level Synthesis

One possible entry point to TaPaSCo's automated toolflow is the use of high-level synthesis, compiling a high-level, behavioral description of an algorithm into an actual FPGA-based accelerator.

In [8], Brugnoni et al. discuss some of the challenges posed to developers with a strong software background that try to use C/C++-based high-level synthesis to create a heterogeneous system with accelerators. Next to many pitfalls related to the performance of the generated accelerator, the integration of the accelerators into a single, heterogeneous system-on-chip is identified as one of the biggest challenges for users of high-level synthesis who are not also experienced FPGA designers. According to their study and evaluation, TaPaSCo is very well suited to overcome this problem and to make FPGA-based accelerators more accessible for software developers. By providing a fully automated flow from a behavioral algorithmic description to a fully integrated accelerator, TaPaSCo enables users with little or no knowledge about hardware development and embedded system design to leverage the power and flexibility of FPGAs.

For performance optimization of their example algorithm, Brugnoni et al. used an iterative development process. During such a process, it is important to be able to quickly assess the accelerator, not only its performance, but also with regard to hardware resource consumption and operating frequency. In their research on high-level synthesis scheduling algorithms [52, 53], Oppermann et al. used TaPaSCo for exactly this purpose: Harnessing the easy-to-use out-of-context synthesis feature that is part of TaPaSCo's high-level synthesis and import flow (cf. Section 3.4), it was possible easily to evaluate a large number of examples and scheduler configurations with regard to effects of the scheduling algorithm on resource requirements and the critical path delay in hardware.

Another example for the integration of HLS-generated accelerators into heterogeneous systems with TaPaSCo can be found in [63]. The HLS compilation flow described there generates multi-core, multi-threaded hardware accelerators from OpenMP parallel loops. In that work, TaPaSCo was not only used to integrate and interface with the generated accelerators, but also *extended* to better suit the needs of the application: In order to reduce the pressure of multi-threaded accesses on the memory subsystem and reduce memory access latency, the TaPaSCo base architecture was extended to include small, thread-private caches inserted between accelerator and the FPGA's external DDR memory.

In a truely heterogeneous setup that combines HLS-generated accelerators with other accelerators such as GPUs, the integration of all these different accelerators into a single application can be challenging. Ideally, all accelerators should expose a single, unified interface for offloading computations from the application onto these accelerators. One candidate for such a unified interface are the device offloading constructs first introduced in version 4.0 of the OpenMP standard. The OpenMP device directives and clauses clearly define a device offloading execution model and provide means to specify which and how to transfer data between host application and accelerator.

Based on TaPaSCo, the work in [62] developed one of the first prototypes for OpenMP device offloading to FPGA-based accelerators: First, the `clang` compiler was extended with an FPGA-specific OpenMP toolchain that leverages TaPaSCo's automated HLS toolflow to run Xilinx Vivado HLS for OpenMP target regions in the input code, eventually yielding a complete bitstream without manual intervention by the user.

For the interfacing with the accelerator at runtime, LLVM's OpenMP runtime was extended with a plug-in that, based on TaPaSCo's portable software API, controls data allocation and -transfers as well as execution of the accelerator. In summary, the `clang` integration and runtime plug-in enable users to create and interface with HLS-generated FPGA accelerators solely using the directives defined in the OpenMP standard. This is also good example of how TaPaSCo can be integrated into a larger toolflow and runtime setup.

### 4.2 Custom HDL-based Accelerators

However, the challenge of integrating an accelerator into a heterogeneous system is not only faced by HLS users, but also by developers of custom accelerators written in hardware description languages such as Verilog or Bluespec. With its second entry-point based on packaged IP cores, TaPaSCo can also be used for the integration of such custom accelerators.

An example of this usage scenario in the context of accelerators for machine learning workloads can be found in [64, 65], where a custom accelerator for the inference in *Sum-Product Networks* (SPN), a class of probabilistic models, was presented. This work is also an excellent example for the high efficiency and low overhead of the TaPaSCo framework: Through full pipelining of the accelerator, including the interface to external DDR memory generated by TaPaSCo and the use of AXI4 burst transfers, the approach was able to achieve very high throughput for the processing of batches of input queries. Despite the much lower operating frequency, the accelerator was able to significantly outperform modern CPUs and GPUs, proving that TaPaSCo is very well suited for such high-performance usage scenarios. In later extensions of this work in [66, 70], the evaluation showed that the overhead for data-transfers between host CPU and FPGA memory had been improved in more recent versions of TaPaSCo, leading to even higher speedups.

The same accelerator was also used to demonstrate TaPaSCo's *portability*: As domain experts from the machine learning community usually do not run and maintain FPGA boards in on-premise setups, the accelerator architecture was ported [51] to the reconfigurable cloud, namely the F1 instances found in Amazon's AWS EC2 cloud. As the TaPaSCo architecture is completely platform-independent and TaPaSCo provides suitable platform integration for the F1 instances, the accelerator could be ported without any changes to the core itself. TaPaSCo's automated design-space exploration was then used to determine the best composition (number of cores and frequencies) in the cloud setup, eventually yielding throughput figures outperforming 12-core Xeon CPUs and Nvidia's V100 GPU.

The automated design-space exploration was also heavily used for the computer vision accelerator in [28, 29]. The accelerator implements the *Semi-Global Matching* (SGM) algorithm used to determine the distance of objects through comparison of images from two different cameras. In contrast to prior work, the developed accelerator did not only allow to process multiple rows of the input images in parallel, but also introduced a more fine-grained level of parallelism by evaluating multiple disparity values, i.e., candidates for the distance of the same pixel in the two images, in parallel. With both, the number of rows and disparity values to process in parallel being configurable, TaPaSCo's automated DSE identified non-obvious tradeoff points yielding frame rates superior to CPU- and GPU-based (up to 35x higher frame-rate) approaches at much lower energy consumption. The approach also demonstrated the portability of TaPaSCo by reusing the same accelerator architecture across a number of very different platforms, ranging from embedded SoCs (ZedBoard) to PCIe-based FPGA extension cards (VC709).

Applications have also always been a major driver to include new functionality into TaPaSCo, such as the network interface support developed for the in-network co-processor for analytical SQL workloads presented in [30]. Modern FPGAs often provide high-speed network interfaces and are very well suited for the implementation of accelerators for in-network and network-attached processing. An example of such a platform is the NetFPGA SUME board supported by TaPaSCo, which comes with four 10 Gbit/s SFP+ network connections. Using this board as a reconfigurable switch, it was possible to move computations from workers in a distributed data-processing system into to the network. Based on the extended version of TaPaSCo, three-stage query processing pipelines that used network- and memory-infrastructure of the FPGA board were developed. By moving hash-table computation and probing from the workers to the reconfigurable switch and thereby reducing the number of expensive shuffle operations, a speedup of up to 7x for left-deep SQL join plans could be achieved.

### 4.3 Softcores

A special case of custom IP cores are *softcores*, i.e., CPU cores provided as HDL for the inclusion in a larger system. Especially with the rising popularity of the open RISC-V ISA, the integration of such softcores has become an important task for TaPaSCo. TaPaSCo provides a number of features that facilitate this task, e.g., the support for local memories directly attached to the processing elements that can be used for instruction and low-latency data memory for RISC-V softcores. In [27], a catalog of open-source RISC-V cores was integrated with TaPaSCo and evaluated with regard to suitability for FPGAs and benchmark performance. More details on how these softcores were integrated with TaPaSCo can be found in the next section.

## 5 Case Study: RISC-V-based Many-Core Architecture

While the previous section gave an overview of some of the different use-cases of TaPaSCo and its versatility, this section describes the typical design process of a FPGA-based SoC using TaPaSCo in more detail. Even more detailed usage instructions for TaPaSCo can be found in the documentation located in the Github repository [68]. To illustrate the user-friendly nature of TaPaSCo's workflow, we use a case study scenario, in this case the inclusion of RISC-V softcores into the system.

TaPaSCo automates and speeds up the development process of FPGA designs, but still requires full hardware synthesis to build a bitstream. As a result, development cycles are longer than in "traditional" software engineering. To combine both worlds and enable fast prototyping with shorter iterations, RISC-V softcore processors are introduced to TaPaSCo [27]. With this setup, the synthesis run is required only once. In subsequent iterations, only the firmware for the softcore has to be compiled again. In later performance optimization steps, the functionally verified firmware can incrementally be converted into dedicated accelerators through HLS or by creating a custom HDL core. The following text discusses the design of a RISC-V processing element, and illustrates its integration into TaPaSCo to accelerate the entire design and implementation process, showing the actual commands required to assemble the SoC. The use-case also employs some of the more advanced memory capabilities of TaPaSCo to describe and manage more complex memory systems (e.g., distinguishing between PE-global and PE-local memories).

### 5.1 RISC-V Processing Elements

To allow TaPaSCo to automatically construct the SoC design, the first step is to design a TaPaSCo-compatible PE. One way to do this is shown in [27], which describes the process to wrap seven different open-source RISC-V cores for compatibility with TaPaSCo. To this end, the design is wrapped into the T-shape, previously discussed in Section 3.1, as follows:

For the *data channel*, existing AXI4 memory interfaces of the RISC-V core can be used for memory accesses. However, the *signaling* and *control* interfaces require additional modules. The control interface is implemented as an AXI4Lite register file module written in Bluespec, called `RVController`. This module uses the direct wire interface of the RISC-V processor to hold the processor in reset until the start register is written. The processor will then start to execute its program, can read execution parameters from the `RVController`, and, at the end, the processor signals the completion writing into a register. The `RVController` then puts the processor back into reset. Finally, it raises the interrupt on the external line to signal completion to the host and allows it collect the results.

Attached to the RISC-V processor is a local BRAM, which is used to store the firmware. Through an AXI4 controller, the host is connected to the second port of the local BRAM. This is used to load a binary firmware image into the BRAM. Accordingly, the host can directly transfer the RISC-V firmware using the standard mechanisms of TaPaSCo (see Section 5.2). The diagram in Fig. 6 shows the final PE design for the prototype.

To provide a convenient way of generating the IP-XACT [33] packaged T-shape that TaPaSCo requires, the wrapper and connection logic generation is scripted. It is available as open-source at [69].

For instance, these scripts can generate a TaPaSCo-ready Piccolo RISC-V processor PE which can be directly imported with the command:

```
tapasco import piccolo32_pe.zip as 1337
```

The `import` command performs two actions: (1) It makes the PE contained in `piccolo32_pe.zip` available to TaPaSCo using the PE *type ID* 1337. This PE type ID will later be used in the runtime to identify the kind of PE a job requires. The ID identifies the functionality of a PE, e.g., the abstract algorithm. Different implementations or algorithms performing the *same* computation will usually share the

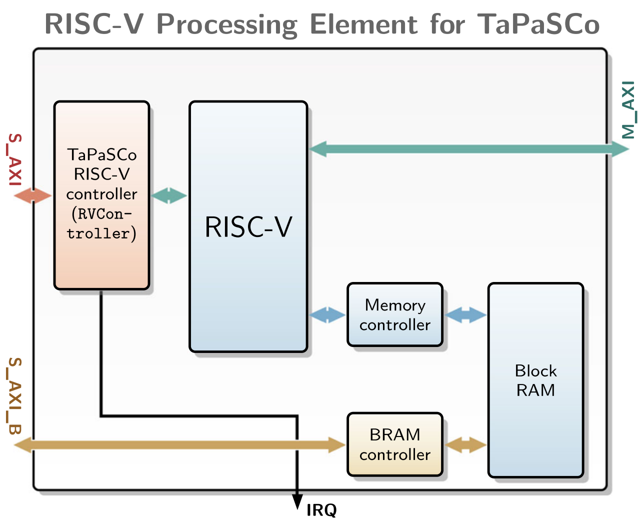

RISC-V Processing Element for TaPaSCo
S_AXI
S_AXI_B
M_AXI
TaPaSCo RISC-V controller (RVController)
RISC-V
Memory controller
Block RAM
BRAM controller
IRQ

**Figure 6** RISC-V processing element: `RVController` provides an AXI4Lite slave interface for TaPaSCo; BRAM is accessible via an AXI4 controller from the RISC-V, as well as from the host (adapted from [27]).

same type ID. (2) The `import` command performs OOC synthesis and place-and-route for all targeted platforms, unless called with the `--skipEvaluation` command argument. The OOC results can be used by later DSE tasks and give estimates for the area utilization $A$ and the maximal operating frequency $F_{max}$ of a single instance of the imported PE. A restriction to a single platform, e.g., the ZedBoard, can be achieved by adding the argument `-p zedboard`.

### 5.2 PE Local Memory

Running a program on the softcore requires a binary firmware image and a location to store it. To offer quickly accessible storage close to a given PE, TaPaSCo uses the concept of *PE-local memory*. The local memory is accessible by the softcore and also by the host. The firmware image itself is simply another parameter to the TaPaSCo job and is automatically copied to the local memory before execution. The hardware realization uses dual-port BRAMs, separated into instruction and data memory if necessary, where one port of each BRAM is connected to the softcore, while the second port is accessible by the host.

All memory transfers to local memory are integrated into the TaPaSCo software API. As part of a job execution, TaPaSCo automatically handles the addressing and all data transfers to the local memory.

In addition to the local memory, a PE has access to *global memory*. Depending on the platform, in most cases this is the larger on-board DDR memory attached to the FPGA, but could also be different memory technologies such as High

Bandwidth Memory (HBM), which is supported for some UltraScale+ FPGAs.

### 5.3 TaPaSCo FPGA Composition

The core goal of TaPaSCo is to free the engineer from having to focus on anything not directly related to the acceleration problem at hand. In this example this means that TaPaSCo constructs the entire on-chip architecture, leaving the developer free to concentrate on the RISC-V PEs and the application code, instead of dealing with e.g., memory interfaces or address maps. A fully working bitstream with two RISC-V processors is generated with a single command:

```
tapasco -v compose [piccolo32_pe x 2] @
50MHz -p zedboard
```

The `compose` command can be used to construct a specific composition, without using any design space exploration; in this case, the composition will include two instances of our RISC-V PE running at 50 MHz and a bitstream will be generated for the ZedBoard. The low operating frequency was picked on purpose here. Together with the small number of PEs it ensures that the synthesis time will be reasonably short, which allows for frequent iterations working in tandem with the software engineers on the application side. For the final evaluation of the prototype, TaPaSCo offers the design space exploration feature to find a good trade-off between number of instances (in this case the number of RISC-V softcores) and the clock frequency. By default, TaPaSCo optimizes for job throughput, which means, the number of computation jobs per second. However, to estimate job throughput, a good approximation of the average computation time required for each job is required. Luckily, this is very simple: Once the RISC-V program is assembled, the number of clock cycles for any given input can be determined by offline simulation. To include all memory effects, the delay introduced by external memories has to be modeled for a more accurate simulation. A number of ways can be used to provide this data to TaPaSCo's DSE, the simplest being re-importing the PE giving the total estimated job execution time:

```
tapasco import piccolo32_pe.zip as 1337
--averageClockCycles 1250000
```

Running TaPaSCo's automated design space exploration is now as simple as:

```
tapasco explore [piccolo32_pe x 2] in
area, freq -p zedboard
```

The `explore` command takes an *initial composition* and a list of design space dimensions; the initial composition determines the ratio of different PE kinds to each other. For example, considering an initial composition `[A x 1, B x 2]`, the TaPaSCo DSE only uses compositions where there are twice as many instances of $B$ as of $A$ when exploring the area dimension. The design space dimensions `area` and `freq` activate the exploration along the area utilization and operating frequency axes, respectively. Finally, the `-p zedboard` platform filter is set this time to restrict the exploration to a single platform.

By default, `explore` will spawn one thread for each active CPU core on the executing machine performing a single composition run in parallel, taking the top elements of the ordered design space (in this case ordered by their estimated job throughput). The DSE will repeat this process until it finds a design that achieves timing closure automatically (see [28] for a more thorough discussion of the DSE algorithm itself). Depending on the design and the available resources, this process can take a varying amount of time, but does not require any user interaction. After a few hours, or days, depending on the complexity of the design, TaPaSCo generates a working bitstream with close-to-ideal operating frequency and number of PEs. However, note that the result might not be optimal due to discretization errors and sub optimal place-and-route quality which can lead to local extrema that trap the optimization strategy.

### 5.4 Application Development with TaPaSCo

The last missing piece for this prototype is the application software: To be precise, the two missing artifacts are the RISC-V programs to execute on the PEs, and a host program that offloads the computations to the FPGA. Discussing the former is out-of-scope for this paper, but the latter is examined briefly here to give an idea of software development with TaPaSCo. Listing 1 contains excerpts from the host program, focusing on the main offloading loop.

The code assumes that the executable binary code of the target RISC-V program has been inserted as the array `prog` into the source code, and the actual input data has already been split into a number of `JobData` segments suitable for parallel processing, stored as `data`. The former can be generated automatically by the provided scripts, which basically take the compiled code and convert it to a usable image format for transfer. The latter, depends on the concrete use-case, but in many simple scenarios, such as summing-up an array of numbers, an `array_view` data structure can be used on a raw data block to perform a useful split very easily and at practically no runtime cost. A TaPaSCo job for each data element is then launched in Line 17.

This line looks intentionally, but deceptively, simple. In fact, there is an enormous amount of work being performed under the hood, which can only be briefly described in this context: In the `launch` call, the PE type ID 1337

**Listing 1** Excerpt of the main loop of the host program (C++17).

```
#include <thread>
#include <vector>
#include <tapasco.hpp>
using namespace tapasco;
...
    /* Perform automatic initialization of first device: */
    Tapasco tapasco;
    auto prog { /* RISC-V binary program code */ };
    auto progW = makeWrappedPointer(prog.data(), prog.size());
    auto progWIn = makeInOnly(progW); /* Do not copy back after execution */
    std::vector<JobData> data {
      { /* data for job 1 */}, { /* data for job 2 */ }, ...
    };
    std::vector<std::thread> threads;
    /* Launch jobs asynchronously. */
    for(auto jd : data)
      threads.push_back(std::thread( // Launch a Job
        tapasco.launch(1337, makeLocal(progWIn), jd);
      ));
    /* Wait for all jobs to finish. */
    for(auto& t : threads)
      if(t.joinable()) t.join();
    /* do something with the result */
```

is used to identify the target PE kind. The program `prog` is wrapped in class constructors called `makeLocal` and `makeInOnly`, which serve as a type annotation for TaPaSCo to determine the requested memory location (here: local) and movement (here: only copied *to* the device). Seeing a `Local` argument, TaPaSCo allocates PE-local memory for the data block (as opposed to device-global memory) at the PE where the execution will take place. Furthermore, TaPaSCo copies the executable code `prog` to the PE memory and passes the handle returned by the allocation to the RISC-V controller. For `jd`, TaPaSCo performs almost the same procedure, only that memory is allocated on the *device-global* memory shared by all PEs. `launch` then proceeds to perform the setup for the launch, starts the PE, and returns a closure to the bottom half of the launch to be executed asynchronously.

Waiting for a job completion is achieved in the bottom half, which consists of 1. waiting for the corresponding completion signal 2. copying back data from the device-global memory for `jd` to the CPU's memory location for `jd`, 3. releasing of the device allocated memory for `prog` and `jd` 4. releasing the PE. This approach hides the fact that a PE for the operation type 1337 may not be available when `launch` is called. In this case, the job will be queued and executed as soon as a PE is available. Since `prog` is marked `InOnly`, it will only be copied *to* the device, but not *back* after execution. On the other hand, since `jd` is *not* marked `InOnly`, it will both be copied to the device prior to the execution, as well as back to main memory afterwards. There exists another type annotation called `OutOnly`, which allows to specify the third case of elements, which need to be allocated on the device, but not copied to the device *before* execution, only *from* the device afterwards. This can be used when data is generated on the device.

### 5.5 Scaling to Larger Devices with TaPaSCo

Assuming the initial prototype on the ZedBoard is satisfactory, it can now be scaled-up to boards with larger reconfigurable devices using TaPaSCo: Targeting the much larger ZCU102 board is simply a case of running DSE again with `-p zcu102`, which generates a new bitstream, likely with significantly more PEs than on the ZedBoard, and likely even running at a higher clock frequency. Note that the application code shown in Listing 1 does *not* need to be changed at all to make use of the new PEs, only a recompilation for the new host processor architecture is required.

## 6 Future Work

A project such as TaPaSCo offers a plethora of opportunities when it comes to further developments. The current status of the majority these developments can be tracked at [18].

Advances range from software engineering efforts, such as introducing a more user-friendly CLI parser, to support for new infrastructure capabilities. For instance, partial reconfiguration support, which allows changing selected PEs on a running FPGA, is currently in a testing state. Introducing new platforms is also a big topic, TaPaSCo tries to support as many platforms as possible with a single tooflow. Recent additions include Amazon AWS support and HBM-based FPGAs.

On the hardware side of TaPaSCo, there are many more open cases. Network-on-chip integration is a hot topic and is currently being evaluated based on HopLite [37, 38]. Depending on the platform, a NoC can increase throughput and improve routability of designs but may also be too costly for smaller devices. The goal here is to provide a seamless integration without relying on user interaction. The NoC integration is also a possible starting point for allowing direct communication between PEs without shared memory, such as direct inter-PE streams or synchronization.

Another development focus are improvements to PE scheduling. Currently, TaPaSCo relies on the host PC to distribute jobs across PEs. On-chip scheduling is being investigated to reduce latency, especially on weaker host machines or for network-attached FPGAs, which have increased round-trip times to the host [26].

Support for heterogeneous memory is also considered. Currently, platforms with additional memories rely on the TaPaSCo plug-in system for integration. Proper integration and a finer-grained selectability of connections to specialized memories for certain PEs could impre and simplify the use of TaPaSCo on these devices.

Work is also being done to redevelop the current runtime in Rust for better maintainability and extendability, while still providing the ease of use of TaPaSCo.

The open source nature of TaPaSCo allows for easy integration of custom requirements and issues posted on GitHub are frequently addressed by the core developer team. In addition, pull requests provide a convenient way to join the community development effort and are very welcome. Information about contributing to the project is available at [17].

## 7 Conclusion

This work illustrates how TaPaSCo reduces the development effort required to implement *scalable, portable FPGA-based computing architectures* by providing both hardware and software abstractions for embedding custom accelerators in FPGA designs. In addition, Section 3.4 argues that TaPaSCo's *design space exploration* facilities remove guesswork and manual design iterations, while improving upon the final result [29].

TaPaSCo is extensively used in a wide number of application domains. Classical areas, such as image processing and machine learning, are prime examples of TaPaSCo aiding designers by letting them focus on their designs instead of infrastructure. But Section 4 also shows that there are many applications that utilize TaPaSCo just as a baseline, expanding it with their own plugins, for instance, to support network processing.

Even complex IP blocks which were not originally intended to be used as processing elements, such as RISC-V softcores, can be integrated into TaPaSCo, thanks to its flexible flow and high flexibility.

TaPaSCo is freely available as open-source software. It provides a reproducible baseline and is easy to extend, simplifying benchmarking and performance evaluation for the academic FPGA community. TaPaSCo is licensed under the GNU LGPLv3 and available on GitHub [68].

**Acknowledgements** The authors would like to thank Xilinx, Inc. for supporting their work by hardware and software donations; as well as Bluespec, Inc. for making their design tools available for education and research. This work was partially funded by the EU in the FP7 research project REPARA (ICT-609666) and the German Federal Ministry for Education and Research (BMBF) with the funding ID 01 IS17091 B.

**Funding** Open Access funding enabled and organized by Projekt DEAL.

## References

1. Adler, M., Fleming, K. E., Parashar, A., Pellauer, M., & Emer, J. (2011). Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (pp. 25–28).
2. Aldinucci, M. (2017). FastFlow: high-level and efficient streaming on multi-core. Programming multi-core and many-core computing systems.
3. Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y. S., Asanović, K., & Nikolić, B. (2020). Chipyard: integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro*, *40*(4), 10–21. https://doi.org/10.1109/MM.2020.2996616.
4. Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., ... Waterman, A. (2016). The rocket chip generator. Tech. rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley.
5. Baaij, C. (2010). Clash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro conf. on digital system design: architectures, methods and tools (DSD)*.
6. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., & Asanovic, K. (2012). Chisel: Constructing hardware in a Scala embedded language. In *Proc. DAC 2012*.
7. BlueSpec Inc. (2003). BlueSpec SystemVerilog, http://bluespec.com/technology/ (2003). acc: 05/16/2018.
8. Brugnoni, S., Corbat, T., Sommerlad, P., Suter, T., Korinth, J., Chevallerie, D. d.e.l.a., & Koch, A (2016). Automated generation

of reconfigurable systems-on-chip by interactive code transformations for high-level synthesis. In *Third International Workshop on FPGAs Software Programmers (FSP)*.
9. Callahan, T. J., Hauser, J. R., & Wawrzynek, J (2000). The Garp architecture and C compiler. *Computer*, *33*(4), 62–69.
10. Canis, A. (2011). LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proc. of the 19th ACM/SIGDA int. symp. on field programmable gate arrays*.
11. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., & Czajkowski, T (2011). LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays,* (pp. 33–36 ).
12. Charles, P. (2005). X10: an object-oriented approach to non-uniform cluster computing.
13. Chen, Y.-T., Cong, J., & Xiao, B (2015). Aracompiler: a prototyping flow and evaluation framework for accelerator-rich architectures. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (pp. 157–158 ).
14. De La Chevallerie, D., Korinth, J., & Koch, A (2014). Integrating FPGA-based processing elements into a runtime for parallel heterogeneous computing. In *2014 International conference on field-programmable technology (FPT)* (pp. 314–317).
15. Digilent Inc. (2015). ZedBoard, http://zedboard.org/product/zedboard. acc: 05/16/2018.
16. Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., Ngadiuba, J., Pierini, M., Rivera, R., Tran, N., & Wu, Z (2018). Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, *13*(07), P07027–P07027. https://doi.org/10.1088/1748-0221/13/07/p07027.
17. Embedded Systems and Applications Group (2020). TU Darmstadt: TaPaSCo Contributor's Guide. https://github.com/esa-tu-darmstadt/tapasco/wiki/Contributor' s-Guide.
18. Embedded Systems and Applications Group (2020). TU Darmstadt: TaPaSCo Issue Tracker. https://github.com/esa-tu-darmstadt/tapasco/issues.
19. Fleming, K., & Adler, M. (2016). The LEAP FPGA operating system. In *FPGAs for software programmers* (pp. 245–258). Springer.
20. Frigo, J., Gokhale, M., & Lavenier, D (2001). Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays* (pp. 134–140 ).
21. Gädke, H., & Koch, A. (2007). Comrade-A compiler for adaptive systems. In *Design, Automation and Test in Europe (DATE)*.
22. Gerstlauer, A., Haubelt, C., Pimentel, A. D., Stefanov, T. P., Gajski, D. D., & Teich, J (2009). Electronic system-level synthesis methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *28*(10), 1517–1530.
23. Gill, A. (2009). Introducing Kansas lava. In *Int. Symp. on implementation and application of functional languages*.
24. Gokhale, M., Stone, J., Arnold, J., & Kalinowski, M (2000). Stream-oriented FPGA computing in the Streams-C high level language. In *Proceedings 2000 IEEE symposium on field-programmable custom computing machines (Cat. No. PR00871)* (pp. 49–56 ).
25. Guo, Z., Buyukkurt, B., Najjar, W., & Vissers, K (2005). Optimized generation of data-path from C codes for FPGAs. In *Design, automation and test in Europe* (pp. 112–117 ).
26. Heinz, C., Hofmann, J. A., Sommer, L., & Koch, A (2020). Improving job launch rates in the TaPaSCo FPGA middleware by hardware/software-co-design. In *2020 IEEE/ACM International workshop on runtime and operating systems for supercomputers (ROSS)* (pp. 22–30 ). https://doi.org/10.1109/ROSS51935.2020.00008.
27. Heinz, C., Lavan, Y., Hofmann, J., & Koch, A (2019). A catalog and in-hardware evaluation of open-source drop-in compatible RISC-V softcore processors. In *IEEE Proc. International conference on ReConFigurable computing and FPGAs (ReConFig)*.
28. Hofmann, J., Korinth, J., & Koch, A (2016). A scalable high-performance hardware architecture for real-time stereo vision by semi-global matching. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*.
29. Hofmann, J., Korinth, J., & Koch, A (2016). A scalable latency-insensitive architecture for FPGA-accelerated semi-global matching in stereo vision applications. In *Proc. Int. Conference on ReConFigurable Computing and FPGAs (ReConFig)*.
30. Hofmann, J., Thostrup, L., Ziegler, T., Binnig, C., & Koch, A (2019). High-performance in-network data processing. In *International workshop on accelerating analytics and data management systems using modern processor and storage architectures, ADMS@VLDB 2019, Los Angeles, United States*.
31. Huang, S. S. (2008). Liquid metal: Object-oriented programming across the hardware/software boundary. In *European conference on object-oriented programming*.
32. Huthmann, J., Liebig, B., Oppermann, J., & Koch, A (2013). Hardware/software co-compilation with the Nymble system. In *2013 8th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)* (pp. 1–8 ).
33. IEEE Standards Association (2014). IEEE 1685-2014 - IEEE Standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows. acc: 05/16/2018.
34. Intel Corporation: Open programmable acceleration engine, https://opae.github.io/. Visited on 08/20/2020.
35. Intel Inc. (2016). Intel FPGA SDK for OpenCL, https://www.altera.com/products/designsoftware/embedded-software-developers/opencl/overview.html. acc: 05/16/2018.
36. Ismail, A., & Shannon, L. (2011). FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *2011 IEEE 19th annual international symposium on field-programmable custom computing machines* (pp. 170–177 ).
37. Kapre, N., & Gray, J. (2015). Hoplite: Building austere overlay NoCs for FPGAs. In *2015 25th international conference on field programmable logic and applications (FPL)*.
38. Kapre, N., & Gray, J. (2017). Hoplite: A deflection-routed directional torus NoC for FPGAs. ACM Trans. Reconfigurable Technol. Syst. 10(2). https://doi.org/10.1145/3027486.
39. Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., Huang, Q., Kovacs, K., Nikolic, B., Katz, R., Bachrach, J., & Asanovic, K (2018). FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (pp. 29–42 ), https://doi.org/10.1109/ISCA.2018.00014.
40. King, M., Hicks, J., & Ankcorn, J (2015). Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays* (pp. 13–22).
41. Koch, D., Beckhoff, C., & Teich, J (2008). ReCoBus-Builder—A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS. In *2008 International conference on field programmable logic and applications* (pp. 119–124).
42. Korinth, J., de la Chevallerie, D., & Koch, A (2015). An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In *IEEE 23rd Ann. int. symp. on field-programmable custom computing machines (FCCM)*.
43. Korinth, J., Hofmann, J., Heinz, C., & Koch, A (2019). The TaPaSCo open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems. In *International symposium on applied reconfigurable computing (ARC)*.

44. Kurth, A., Vogel, P., Capotondi, A., Marongiu, A., & Benini, L (2017). HERO: Heterogeneous embedded research platform for exploring RISC-V Manycore accelerators on FPGA. arXiv:1712.06497.
45. Lange, H., & Koch, A. (2000). Memory access schemes for configurable processors. In *International workshop on field programmable logic and applications* (pp. 615–625).
46. Lo, C., & Chow, P. (2018). Multi-fidelity optimization for high-level synthesis directives. In *2018 28th International conference on field programmable logic and applications (FPL)* (pp. 272–2727).
47. Lübbers, E., & Platzner, M. (2009). ReconOS: Multithreaded programming for reconfigurable computers. *ACM Transactions on embedded computing systems (TECS)*, *9*(1), 1–33.
48. Mantovani, P., Giri, D., Di Guglielmo, G., Piccolboni, L., Zuckerman, J., Cota, E. G., Petracca, M., Pilato, C., & Carloni, L.P (2020). Agile SoC development with open ESP : Invited Paper. In *2020 IEEE/ACM international conference on computer aided design (ICCAD)* (pp. 1–9 ).
49. Minhas, U. I., Woods, R., & Karakonstantis, G (2019). Evaluation of FPGA partitioning schemes for time and space sharing of heterogeneous tasks. In *International symposium on applied reconfigurable computing* (pp. 334–349 ).
50. Nane, R., Sima, V.-M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y. T., Hsiao, H., Brown, S., & Ferrandi, F (2015). A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *35*(10), 1591–1604.
51. Ober, M., Hofmann, J., Sommer, L., Weber, L., & Koch, A (2019). High-throughput multi-threaded sum-product network inference in the reconfigurable cloud. In *Fifth international workshop on heterogeneous high-performance reconfigurable computing (H2RC)*.
52. Oppermann, J., Reuter-Oppermann, M., Sommer, L., Koch, A., & Sinnen, O (2019). Exact and practical modulo scheduling for high-level synthesis. *TRETS*, *12*(2), 8:1–8:26. https://doi.org/10.1145/3317670.
53. Oppermann, J., Sommer, L., Weber, L., Reuter-Oppermann, M., Koch, A., & Sinnen, O (2019). SkyCastle: a resource-aware multi-loop scheduler for high-level synthesis. In *International conference on field-programmable technology (FPT)*.
54. Peck, W. (2006). Hthreads: A computational model for reconfigurable devices. In *Int. Conf. on Field Programmable Logic and Applications (FPL'06)*.
55. Peck, W., Anderson, E., Agron, J., Stevens, J., Baijot, F., & Andrews, D (2006). Hthreads: A computational model for reconfigurable devices. In *2006 International conference on field programmable logic and applications* (pp. 1–4 ).
56. Pilato, C., & Ferrandi, F. (2013). Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International conference on field programmable logic and applications* (pp. 1–4).
57. REPARA Project Consortium (2016). Work Package 5 deliverables. acc: 05/16/2018.
58. Rodriguez, A., Valverde, J., Portilla, J., Otero, A., Riesgo, T., & De la Torre, E (2018). Fpgabased high-performance embedded systems for adaptive edge computing in cyber-physical systems: The artico3 framework, (Vol. 18 p. 1877).
59. Skalicky, S., Schmidt, A. G., & French, M (2014). High level hardware/software embedded system design with redsharc. arXiv:1408.4725.
60. Slurm Workload Manager, https://slurm.schedmd.com/overview.html. acc: 08/03/2018.
61. So, H. K.-H., & Brodersen, R. W. (2007). BORPH: An operating system for FPGA-based reconfigurable computers. University of California, Berkeley.
62. Sommer, L., Korinth, J., & Koch, A (2017). OpenMP device offloading to FPGA accelerators. In *2017 IEEE 28th Int. conf. on application-specific systems, architectures and processors (ASAP)*.
63. Sommer, L., Oppermann, J., Hofmann, J., & Koch, A (2017). Synthesis of interleaved multithreaded accelerators from OpenMP Loops. In *2017 international conference on reconfigurable computing and FPGAs (ReConFig'17)*.
64. Sommer, L., Oppermann, J., Molina, A., Binnig, C., Kersting, K., & Koch, A (2018). Automatic mapping of the sum-product network inference problem to FPGA-based accelerators. In *IEEE International conference on computer design (ICCD)*.
65. Sommer, L., Oppermann, J., Molina, A., Binnig, C., Kersting, K., & Koch, A (2018). Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem. In *ICML 2018 Workshop on tractable probabilistic models (TPM)*.
66. Sommer, L., Weber, L., Kumm, M., & Koch, A (2020). Comparison of arithmetic number formats for inference in sum-product networks on FPGAs. In *2020 IEEE 28th Annual international symposium on field-programmable custom computing machines (FCCM)*.
67. Sotiriou-Xanthopoulos, E., Xydis, S., Siozios, K., Economakos, G., & Soudris, D (2016). An integrated exploration and virtual platform framework for many-accelerator heterogeneous systems. *ACM Transactions on embedded computing systems (TECS)*, *15*(3), 1–26.
68. TaPaSCo (2017). https://github.com/esa-tu-darmstadt/tapasco. acc: 04/09/2020.
69. TaPaSCo RISC-V (2019). https://github.com/esa-tu-darmstadt/tapasco-riscv. acc: 04/09/2020.
70. Weber, L., Sommer, L., Oppermann, J., Molina, A., Kersting, K., & Koch, A (2019). Resource-efficient logarithmic number scale arithmetic for SPN inference on FPGAs. In *International conference on field-programmable technology (FPT)*.
71. Weber, S. J., Paul, J. M., & Thomas, D.E (2001). Co-RAM: combinational logic synthesis applied to software partitions for mapping to a novel memory device, (Vol. 9 pp. 805–812).
72. Wenzel, J., & Hochberger, C. (2016). RapidSoC: short turnaround creation of FPGA based SoCs. In *Proceedings of the 27th international symposium on rapid system prototyping: shortening the path from specification to prototype* (pp. 86–92 ).
73. Xilinx Inc. (2018). Vivado high level synthesis, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. acc: 05/16/2018.
74. Xilinx Inc. (2020). Vitis Platform, https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (visited on 09/10/2020).
75. Xu, C., Liu, G., Zhao, R., Yang, S., Luo, G., & Zhang, Z (2017). A parallel bandit-based approach for autotuning fpga compilation. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays* (pp. 157–166).
76. Yang, H. J., Fleming, K., Adler, M., & Emer, J (2014). LEAP shared memories: Automating the construction of FPGA coherent memories. In *2014 IEEE 22nd annual international symposium on field-programmable custom computing machines* (pp. 117–124 ).
77. Zaruba, F., & Benini, L. (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 (VLSI) Systems.
78. Zhang, P., Huang, M., Xiao, B., Huang, H., & Cong, J (2015). CMOST: a system-level FPGA compilation framework. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (pp. 1–6).

## Affiliations

**Carsten Heinz**[1] · **Jaco Hofmann**[1] · **Jens Korinth**[1] · **Lukas Sommer**[1] · **Lukas Weber**[1] · **Andreas Koch**[1]

Jaco Hofmann
hofmann@esa.tu-darmstadt.de

Jens Korinth
korinth@esa.tu-darmstadt.de

Lukas Sommer
sommer@esa.tu-darmstadt.de

Lukas Weber
weber@esa.tu-darmstadt.de

Andreas Koch
koch@esa.tu-darmstadt.de

[1] Technical University of Darmstadt, Darmstadt, Germany