The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems

Jens Korinth[0000−0002−5792−2562], Jaco Hofmann[0000−0003−3691−3293], Carsten Heinz[0000−0001−5927−4426], and Andreas Koch[0000−0002−1164−3082]

Technical University of Darmstadt, Darmstadt, Germany
{korinth,hofmann,heinz,koch}@esa.tu-darmstadt.de

Abstract. In this paper we present *TaPaSCo – the Task Parallel Systems Composer*, an open-source, toolflow and software framework for automated construction of System-on-Chip FPGA designs for task parallel computation. TaPaSCo aims to increase the *scalability* and *portability* of FPGA designs by performing the construction of heterogeneous many-core architectures from custom processing elements, and providing a simple, uniform programming interface to utilize spatially parallel computation on FPGAs. A key feature of TaPaSCo's is automated *design space exploration*, which can be performed in parallel on a computing cluster. This greatly simplifies scaling hardware designs, facilitating iterative growth and portability across FPGA devices and families.

Keywords: FPGA · reconfigurable computing · design space exploration · System-on-Chip design · design automation · high-level synthesis · scalability · portability · TaPaSCo · heterogeneous computing · parallel computing

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.

In this paper we present *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. TaPaSCo aims to harness and exponentiate 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*. Using TaPaSCo, existing designs can be more easily re-targeted to new FPGAs and boards without requiring changes to the accelerators themselves.

Furthermore, this allows to postpone the decision for the 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, in Section 3 we give a general overview of TaPaSCo and its primary design abstractions. Specifically, we aim to show how TaPaSCo addresses portability, scalability and extensibility of FPGA hardware designs for systems-on-chip. A practical usage example, including the actual commands for using the tool, is discussed in Section 4. The simple use-case is the creation of a many-core design using MicroBlaze CPUs as processing elements, demonstrating the usage and advantages of TaPaSCo. However, the tool easily allows intermixing of arbitrary kinds of PEs (software-programmable processors, IP blocks, HLS-generated functions etc.) to create truly heterogeneous systems, as well.

2 Related Work

The work presented here is not focused on actual high level synthesis tools such as Vivado HLS [20], Nymble [9], or LegUp [3]. 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 (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 design 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 greatyly reduces the comparability of different implementations. This problem is precisely what motivated the work on ThreadPoolComposer [12], our prior research effort in this area. TaPaSCo is based on ThreadPoolComposer, which is in turn is closely related to previous work on ReconOS [14], hthreads [15], or FUSE [11]. 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) and academic (X10 [4, 5], FastFlow [2]) parallel programming frameworks. Redsharc [17] 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. Similar commercial tools, such as Xilinx SDSoC and SDAccel became publicly available later in late 2015/2016; the former works only on select boards of the Zynq

family of FPGAs, the latter only on select PCIe-based boards for OpenCL computing and does not provide support for job dispatches or custom infrastructure cores. In contrast, TaPaSCo allows much deeper customization, e.g., black-box extension of existing cores with caches, using infrastructure cores to change the interconnection (e.g., by buses, networks-on-chip) of processing elements and interface adapters. TaPaSCo's customizability is key to enable performance for very different computing approaches by not imposing too many restrictions on the design. In [7], 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 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, Xilinx ZC706, Xilinx ZCU102 UltraScale+ MPSoC, Xilinx VC709, Xilinx VCU118 and NetFPGA SUME).

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, as 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. In Section 3.3, Section 3.4 and 3.5, we will argue 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 Figure 2 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, and signaling, leading to the complete system shown in Figure 3. 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 reusable across targets. **A3** finally connects the *Architecture* to the hardware components of the target FPGA board. The scripts currently utilize the Vivado Tcl APIs to automate the wiring of high pin count

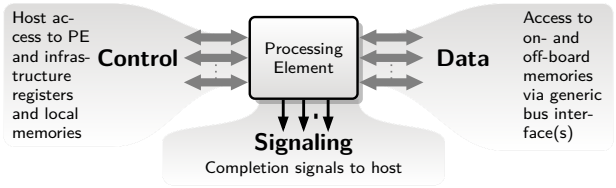

Fig. 1. Basic *T-shape* of processing elements: Each processing element has a *control channel*, a *data channel* and a *signaling* channel, all of which can be implemented by arbitrary means, e.g., AXI4, or Avalon.

interfaces (e.g., AXI4). The *T-model*, named for its T-shape shown in Figure 1, 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 contained individual PEs. The T-shape is repeated here: Each *cluster* itself is T-shaped and can be wired like the PEs themselves (see Figure 2). On the outermost platform-independent level, this process is repeated across

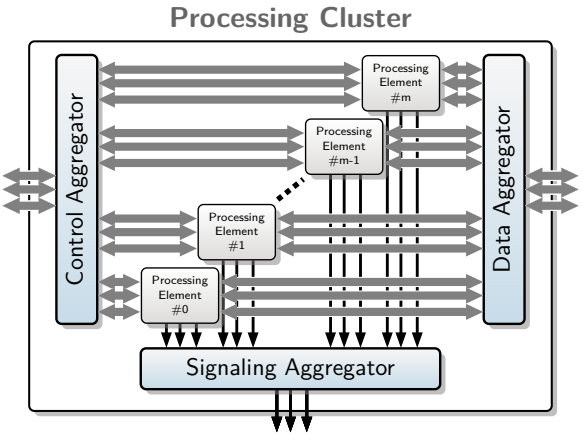

Fig. 2. Processing elements of same kind are grouped into *clusters* using channel aggregators for the three basic channels, e.g., AXI4 Interconnects and interrupt controllers.

clusters. The collective term we use 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 Figure 3. 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

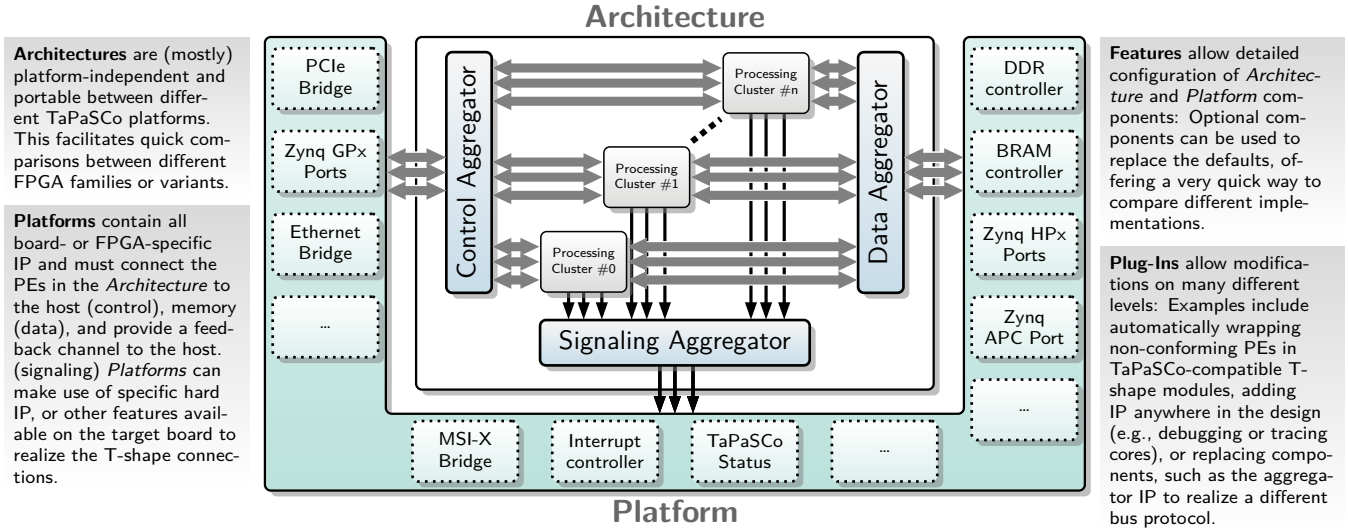

Fig. 3. Using the T-model to hierarchically repeat the automated wiring process to connect the individual clusters into a heterogeneous pool architecture. The availability and presence of components in the *Platform* layer depends on the target FPGA and board; the *Architecture* does not depend on their presence or availability, making the essential design portable.

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 4 for near-data processing, as well as for a complex high-performance Stereovision accelerator based on a systolic array [7]. It would also be entirely possible to use TaPaSCo to connect only a single PE (containing a full architecture inside). TaPaSCo does not impose a model on the PE, it only facilitates easy spatial replication. The last abstraction is called the *Platform*: All parts of the hardware design which are specific to the 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 with numerous injection points for extensions, where *plug-ins* can be inserted to modify the design in flight.

Example 1. If a PE does not have a TaPaSCo-compatible register interface (see [16] for a more detailed description of the register conventions used by TaPaSCo), a plug-in can automatically instantiate a suitable wrapper and TaPaSCo continue with the automated wiring. A different example can be found in the `zedboard` *Platform*: The Digilent ZedBoard [6] 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.

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. This specifically 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 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*, without having to change the host code, cf. [16]). In TaPaSCo's software framework, a task corresponds to a *cluster* and 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 4). To implement this rather abstract API, TaPaSCo internally mirrors the abstractions of the hardware design (see Figure 4): The *TaPaSCo library* is concerned with the *Architecture*. It manages

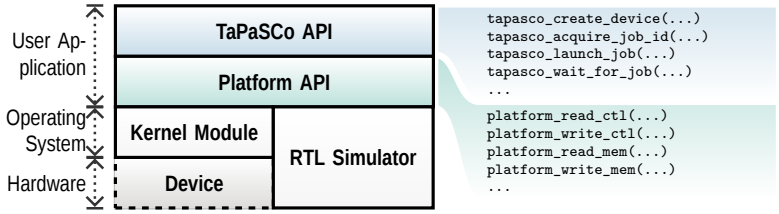

Fig. 4. Software Layer Hierarchy in TaPaSCo: The top-level API provides task-parallel abstraction, the *Platform API* provides a thin user-space layer above either a) the operating system primitives implemented in the *TaPaSCo Loadable Kernel Module (TLKM)*, which in turn interacts directly with the device(s), or, alternatively, b) interfaces with a RTL simulator of the hardware design.

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 are themselves 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 only be implemented at most once for each *Platform* (often code can even be shared among families of devices). They 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 our documentation [13] for more details on the internal APIs.

3.3 Portability

The overall approach outlined in Section 3.1 and Section 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.

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: At the time of writing, TaPaSCo supports seven different platforms, ranging from small embedded boards using Zynq devices, up to high-performance PCIe-based expansion cards with large 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 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 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 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. The remaining interval is then divided evenly in 5 MHz steps by default, each step being the frequency component of a coordinate in the 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 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 cutting off viable designs. 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* 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. Therefore TaPaSCo supports different heuristic functions to score each element in the design space and then explore batches of elements ordered by their score; at each step, the design space is pruned, e.g., of the elements which have a lesser score

than the best element found so far. Since such explorations still require a lot of computing power, TaPaSCo supports the use of the Slurm Workload Manager [1] to parallelize the DSE across entire high-performance computing clusters.

Example 2. Assume the user specifies an initial composition consisting of three different PE kinds, called `A`, `B` and `C`, 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 $[A \times 2, B \times 4, C \times 6]$; in the following, we will call such a configuration a *composition*. When scaling linearly, the smallest composition with the same ratios containing all PEs is thus $[A \times 1, B \times 2, C \times 3]$. Also assume that TaPaSCo has determined via OOC that the largest composition fitting on the target FPGA is $[A \times 3, B \times 6, C \times 9]$. 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 [8].

Table 1. Initial design space for TaPaSCo DSE run. F = Target Design Frequency, R = Replication Factor.

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	$[A0 \times 1, B \times 2, C \times 3]$	$[A0 \times 2, B \times 4, C \times 6]$	$[A0 \times 3, B \times 6, C \times 9]$
	$[A1 \times 1, B \times 2, C \times 3]$	$[A1 \times 2, B \times 4, C \times 6]$	$[A1 \times 3, B \times 6, C \times 9]$

3.5 Extensibility

Given the vast variety of scenarios in which FPGAs are often used, it is impossible for a generic toolchain like TaPaSCo to anticipate and support every use case out of the box. Instead of a one-size-fits-all approach we opted for a high degree of modularity and extensibility in all parts and aspects of TaPaSCo. Using plug-ins and features to modify the hardware design generated by TaPaSCo has already been discussed in Section 3.1. Adding new *Platform*s or *Architecture*s is very easy, too. 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 scientist. We therefore also aimed at making most parts of TaPaSCo modular 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. [18]), or OpenCL.

4 Case Study: MicroBlaze-based many-core architecture

TaPaSCo was recently used in a study of *near data processing (NDP)*, where an FPGA is inserted in between storage elements and the host, and simple data processing tasks (which require no inter-task synchronization facilities) are offloaded to be performed by the FPGA instead of the host. This approach can free the main CPU from trivial, but data-intensive tasks, such as summing or calculating averages, and avoid expensive data transfers. To simplify the programming of the system, the first prototype of the NDP system, called *Shishito*, consists of MicroBlaze soft-core processors [19] with small local BRAM storage and direct access to the memory controller. Each core runs independently of and asynchronously to the others without synchronization across tasks. The following sections will discuss the design of the Shishito processing elements and the overall architecture, then proceed to illustrate how TaPaSCo accelerated the whole design and implementation process, showing the actual commands required to assemble the SoC. The NDP use-case also employs capabilities just recently added to TaPaSCo to describe and manage more complex memory systems (e.g., distinguishing between PE-global and PE-local memories).

4.1 Shishito Processing Elements

To allow TaPaSCo to automatically construct the SoC design for us, the first step was to design a TaPaSCo-compatible MicroBlaze PE. The MicroBlaze processor has a multitude of configuration options, from minor changes such as enabled/disabled exception support, over support for optional instructions, up to different instruction pipeline architectures. In the NDP scenario, the programs running in the MicroBlazes are relatively simple. We thus deactivated most instruction set extensions in favor of larger data caches. A common headache in this scenario is to find a good size for the caches, so we decided to explore different sizes, where both data and instruction cache share the same BRAM-backed storage. To make this design work well with TaPaSCo, we needed to wrap it into the T-shape, previously discussed in Section 3.1, as follows:

While the AXI4 memory interface can simply be turned on using a configuration parameter for the MicroBlaze, the signaling and control interfaces require additional modules. The control interface is implemented as an AXI4Lite register file module written in Chisel, called `MBCtrl`. This module uses the direct wire interface of the MicroBlaze to hold the processor in reset until the start register

is written. The processor will then start to execute its program, which should end with triggering an interrupt at a local interrupt controller (see Figure 5). `MBCtrl` receives and acknowledges the interrupt immediately, then puts the MicroBlaze back in reset. Finally, it raises the interrupt on the external line to signal completion to the host.

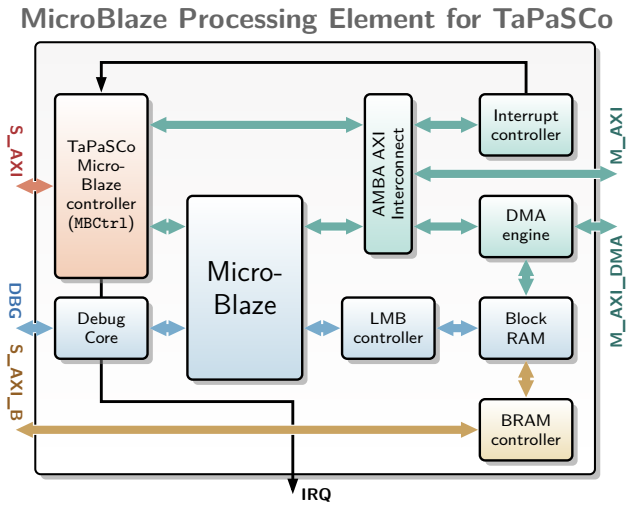

Fig. 5. Shishito processing element: `MBCtrl` provides an AXI4Lite slave interface for TaPaSCo; BRAM is accessible via LMB from the MicroBlaze, as well as via an AXI4 controller. An optional DMA engine can transfer data between local and device memory.

In order to be able to communicate with the host via BRAM, we attached an AXI4 controller to the local BRAM. This allows us to directly transfer the MicroBlaze programs using the standard mechanisms of TaPaSCo (see Section 4.3). The diagram in Figure 5 shows the final PE design for the prototype; this design is assembled by a Tcl script for Vivado Design Suite. It could have been generated by BlueSpec, or Chisel, or directly from Verilog/VHDL, just as well, but since we are using several components from the Xilinx IP catalog, this approach was the fastest. Note, however, that any of these approaches to define the PE would have worked with TaPaSCo.

TaPaSCo only requires an IP-XACT [10] description of the module and the T-shape of interfaces to perform the wiring of any PE automatically. In our case, we used the Vivado Design Suite to generate an IP-XACT description of the PE and packaged it into a `.zip` file, which can be directly imported into TaPaSCo:

```
tapasco import shishito.zip as 1337
```

The `import` command performs several actions: In general, it makes the PE contained in `shishito.zip` available to TaPaSCo using the *kind ID* 1337. This kind ID will later be used in the application to identify the kind of PE a job requires. Note that this only identifies the abstract algorithm; different implementations or algorithms performing the same computation will usually share the same kind

ID, since this knowledge should be hidden from the user. Unless given with the `--skipEvaluation` command argument, the `import` command will perform OOC synthesis and place-and-route for all targeted Platforms, in this case all known Platforms (this could be restricted, e.g., to the ZedBoard Platform using `-p zedboard`). OOC will yield estimations for both area utilization A per instance and maximal operating frequency F_{max}.

4.2 Shishito Architecture

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 our case this means that we will let TaPaSCo construct the entire on-chip architecture for us, leaving us free to concentrate on the MicroBlaze PEs and their application code. To get the software engineers up and running, we can generate a fully working bitstream with two MicroBlazes for them with a single command:

```
tapasco -v compose [shishito x 2] @ 100MHz -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 MicroBlaze PE running at 100 MHz and a bitstream will be generated for the ZedBoard. The low operating frequency and number of PEs ensures that the synthesis time will be reasonably short, so we can use frequent iterations while working in tandem with the software engineers on the application side. For the final evaluation of the prototype, we will use the design space exploration feature of TaPaSCo to find a good trade-off between number of instances and operating frequency. By default, TaPaSCo will optimize job throughput, i.e., the number of computation jobs per second. However, to estimate job throughput, we need a good approximation of the average computation time required for each job. Luckily, this is very simple: Once the MicroBlaze program is assembled, the number of clock cycles for any given input can be determined by offline simulation. A number of ways can be used to provide this data to TaPaSCo's DSE, the simplest being re-importing the PE with the additional data:

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

Now we're ready to harness the power of TaPaSCo's automated design space exploration:

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

The `explore` command will take an *initial composition* and a list of design space dimensions; the initial composition determines the ratio of PE kinds to each other, e.g., for an initial composition $[A \times 1, B \times 2]$, TaPaSCo will only use compositions where there are twice as many instances of B as of A when varying the area utilization. The design space dimensions `area` and `freq` activate the exploration along the area utilization and operating frequency axes, respectively. Note that we used a `-p zedboard` Platform filter this time, to restrict the exploration to a single Platform. By default, `explore` will spawn one thread for each active CPU

```
1  #include<vector>
2  #include<tapasco.hpp>
3
4  /* Perform automatic initialization of first device: */
5  Tapasco tapasco;
6  auto prog { /* MicroBlaze program */ };
7  std::vector<JobData> data { /* data for each job */ };
8  std::vector<JobFuture> threads;
9  /* Launch jobs asynchronously. */
10 for(JobData& jd : data)
11   threads.push_back(
12     tapasco.launch(1337, Local(InOnly(prog)), jd);
13   );
14 /* Wait for all jobs to finish. */
15 for(auto& t : threads)
16   t.wait();
17 /* do something with the result */
```

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

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 until it finds a design that achieves timing closure automatically (see [7] for a more thorough discussion of the algorithm itself). This will usually take a lot of time and computing resources, but does not require any interaction. After a few hours, or days, depending on the complexity of the design, TaPaSCo will generate a working bitstream with close to ideal operating frequency and number of PEs.

4.3 Application Development with TaPaSCo

The last missing piece for our prototype is the application software: To be precise, we need the MicroBlaze programs to execute on our 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 will be examined briefly to give an idea of software development with TaPaSCo. Listing 1 contains excerpts from the host program focusing on the main offloading loop. Assume that the executable binary code of the target MicroBlaze 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`. Note that this assumption is not unrealistic. In many simple cases, 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. In Line 14, we launch a TaPaSCo job for each data element. This line looks intentionally, but deceptively, simple. In fact, there is an enormous amount of work being performed under the hood, which we can only briefly gloss over: In the `launch` call, the kind ID 1337 is used to identify the target PE kind, as expected; the program `prog` is wrapped in class constructors called `Local` and `InOnly`, which simply serve as a type annotation for C++ template expansion in `tapasco.hpp`. Seeing a `Local` argument, TaPaSCo will allocate PE-local memory for the data block (as opposed to device-global memory) at the PE where the execution will take place. It will then copy the executable code `prog` to the PE memory and

pass the handle returned by the allocation to the MicroBlaze program. For `jd`, TaPaSCo performs much the same procedure, only that allocation takes place 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. The bottom half consists of 1. waiting for the corresponding completion signal, then 2. copying back data from the device-global memory for `jd` to the CPU's memory location for `jd`, 3. releasing of the allocated memory for `prog` and `jd`, and 4. finally releasing the PE. Note that the bottom half does not launch a separate thread, but is instead executed only at the call to its `wait` method in the loop below; each bottom half thus executes on the main thread of the application. This approach allows us to hide the fact, that a PE for the kind 1337 may not be available when calling `launch`; 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 (e.g., data generated on the device).

4.4 Scaling out with TaPaSCo

Assuming our initial prototype on the ZedBoard was satisfactory, we can now easily scale up to larger boards using TaPaSCo: E.g., we can simply target a much larger ZC706 by running our DSE again with `-p zc706`, which will generate a new bitstream, likely with significantly more PEs than on the ZedBoard, likely even running at a higher frequency. The application code shown in Listing 1 does *not* need to be changed at all to make use of the new PEs. In fact, since the CPU architecture on ZC706 and ZedBoard is the same, it does not even need to be recompiled!

5 Conclusion

We have shown how TaPaSCo can reduce 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. Furthermore, we argue that TaPaSCo's *design space exploration* facilities can remove guesswork and manual design iterations, while improving upon the final result (cf. [8]). Last but not least, TaPaSCo is freely available as open-source. 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 our public GitLab website [13].

References

1. Slurm Workload Manager. `https://slurm.schedmd.com/overview.html`. acc: 08/03/2018.
2. M. Aldinucci et al. Fastflow: high-level and efficient streaming on multi-core. *Programming Multi-core and Many-core Computing Systems*, 2017.
3. A. Canis et al. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proc. of the 19th ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*. ACM, 2011.
4. P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40. ACM, 2005.
5. D. de La Chevallerie, J. Korinth, and A. Koch. Integrating FPGA-based processing elements into a runtime for parallel heterogeneous computing. In *Field-Programmable Technology (FPT), 2014 Int. Conf. on*. IEEE, 2014.
6. Digilent Inc. ZedBoard. `http://zedboard.org/product/zedboard`, 2015. acc: 05/16/2018.
7. J. Hofmann, J. Korinth, and A. Koch. A scalable high-performance hardware architecture for real-time stereo vision by semi-global matching. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition Workshops*, 2016.
8. J. Hofmann, J. Korinth, and A. Koch. 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)*, 2016.
9. J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. Hardware/software co-compilation with the nymble system. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th Int. Workshop on*. IEEE, 2013.
10. IEEE Standards Association. *IEEE 1685-2014 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*. 2014. acc: 05/16/2018.
11. A. Ismail and L. Shannon. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Ann. Int. Symp. on*. IEEE, 2011.
12. J. Korinth, D. de la Chevallerie, and A. Koch. An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In *Field-Programmable Custom Computing Machines (FCCM), IEEE 23rd Ann. Int. Symp. on*. IEEE, 2015.
13. J. Korinth and A. Koch. TaPaSCo. `https://git.esa.informatik.tu-darmstadt.de/tapasco/tapasco`, 2017. acc: 05/16/2018.
14. E. Lübbers and M. Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1), 2009.
15. W. Peck et al. Hthreads: A computational model for reconfigurable devices. In *Field Programmable Logic and Applications (FPL'06), Int. Conf. on*. IEEE, 2006.
16. REPARA Project Consortium. *Work Package 5 Deliverables*. 2016. acc: 05/16/2018.
17. Sam Skalicky, Andrew G. Schmidt, and Matthew French. High level hardware/-software embedded system design with redsharc. *CoRR*, abs/1408.4725, 2014.
18. L. Sommer, J. Korinth, and A. Koch. OpenMP device offloading to FPGA accelerators. In *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th Int. Conf. on*. IEEE, 2017.
19. Xilinx Inc. *UG984 - MicroBlaze Processor Reference Guide*. 2018. acc: 05/16/2018.
20. Xilinx Inc. Vivado High Level Synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2018. acc: 05/16/2018.