Improving Job Launch Rates in the TaPaSCo FPGA Middleware by Hardware/Software-Co-Design

Carsten Heinz, Jaco A. Hofmann, Lukas Sommer, Andreas Koch

Embedded Systems and Applications Group
TU Darmstadt
Darmstadt, Germany
{heinz,hofmann,sommer,koch}@esa.tu-darmstadt.de

***Abstract*—In recent years, FPGAs have established themselves as an important acceleration platform next to GPUs in heterogeneous HPC systems, providing flexibility and high performance for tasks such as machine learning inference or DNA sequencing.**

While the design of the FPGA-based accelerator cores has become accessible to a broader range of users through customized RISC-V soft-cores and the maturity of High-Level Synthesis (HLS), the integration of and interaction with such accelerator cores in the overall heterogeneous system remains a challenging task.

The open-source TaPaSCo framework eases this task by providing a concise software API and middleware for the interaction with FPGA-based accelerator system-on-chips automatically generated from user-provided accelerator cores.

In this work, we present an extension of the TaPaSCo framework which improves the launch rates and latencies of FPGA-accelerated compute jobs, a crucial factor for the performance of the overall system, through hardware/software-co-design of an improved Rust-based software runtime, and a job dispatcher itself accelerated by hardware.

Our evaluation shows that the new dispatchers can provide an improvement of up to 6x in job throughput with only minimal resource overhead.

***Index Terms*—FPGA, Runtime, Task launching**

I. Introduction

In recent years, HPC workloads are becoming increasingly diverse. In the quest to provide each of these applications with the best possible performance, most HPC systems are now *heterogeneous* systems, combining the "classical" multi-core CPU with a whole range of different, specialized accelerators. Next to GPUs, vector processors (e.g., NEC Aurora), and dedicated AI accelerators such as Cerebras' Wafer Scale Engine (WSE), FPGAs have begun to play an important role in these heterogeneous systems. Starting out in low-power embedded use-cases such as [1], they have now advanced into HPC/Cloud scenarios [2], providing the flexibility to implement different application-specific hardware accelerators on a single platform.

While modern hardware-description languages (HDL) such as Bluespec or Chisel3, the availability of open-source RISC-V CPU softcores, and the significant advances in the usability and performance of High-Level Synthesis (HLS) have made it easier to design the FPGA hardware accelerator itself, the *integration* of these accelerators into the overall heterogeneous hardware/software system, specifically the interaction with the host CPU, remain a challenging task.

The open-source framework TaPaSCo [3], [4] was created to facilitate exactly this task by providing an automatic tool-flow to create a complete FPGA-design from individual accelerator cores for a large variety of platforms and a concise C/C++-based runtime API that allows to manage FPGA device execution from the host CPU, including job launches and data-transfers between the host main memory and FPGA memory.

So far, the TaPaSCo execution model has been entirely host-centric, i.e., every device job launch required synchronization with the host. If an application is composed from multiple small tasks that should be executed on the FPGA, but depend on each other's results, the repeated synchronization with the host can add significant overhead to the overall application performance. In order to overcome this limitation, we present an extension of the TaPaSCo SoC framework and middleware that allows for *on-device* job launch offloading and inter-PE synchronization. To this end, we extend TaPaSCo's hardware architecture with a hardware dispatcher and re-design a substantial part of TaPaSCo's software runtime.

In our evaluation, we measure job throughput and launch latency.

II. Related work

The task of compute job invocation has also been addressed by the major FPGA vendors. Xilinx provides the Xilinx Runtime Library (XRT) [5], which is integrated into their OpenCL and HLS workflow. Intel has the Open Programmable Acceleration Engine (OPAE), which is a runtime to access OpenCL and RTL kernels on an Intel FPGA [6]. More specialized solutions of hardware-software co-design have been presented: Tang et al. [7] used a hardware scheduler to assist a RTOS running on a soft-core, and ReconOS [8] orchestrates co-execution using both hardware- and software threads. The framework Connectal [9] provides automation for creating a design consisting of hardware and software components. Asynchronous communication between the components is provided in the form of *Portals*.

GPUs face similar challenges, which have been addressed in many publications, e.g., Chatterjee et al. [10] implemented a

This research was partially funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01 IS 17091 B.

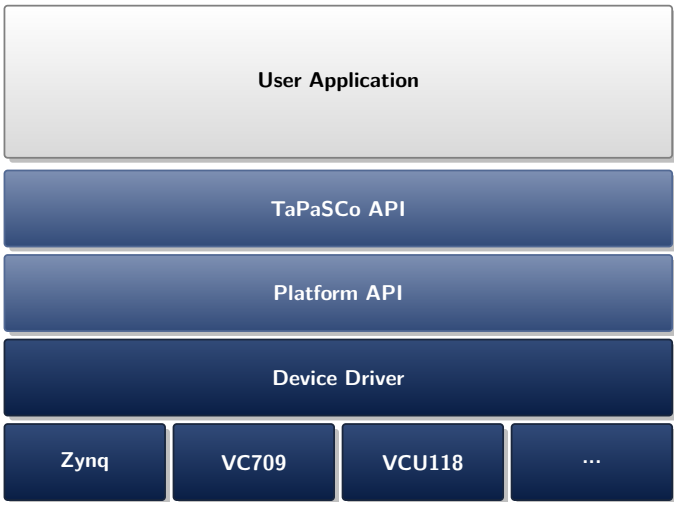


Fig. 1. TaPaSCo runtime stack for interaction with FPGA-based accelerators from host.

```
1  #include <tapasco.hpp>
2  #include <vector>
3  using namespace tapasco;
4
5  ...
6  /* Perform automatic initialization
7     of first device: */
8  Tapasco tapasco;
9  /* data buffer */
10 std::vector<int> v = {0, 1, ...};
11 auto buf = makeWrappedPointer(
12     v.data(), v.size() * sizeof(int));
13 /* Launch a TaPaSCo job */
14 auto myjob = tapasco.launch(
15     KERNEL_ID,
16     buf);
17 /* Wait on Future myjob for completion */
18 myjob();
19 ...
```

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

runtime for dynamic task parallelism. Originating in the GPU space, the Heterogeneous System Architecture (HSA) specifies communication and memory interfaces to allow job execution on a heterogeneous computing system with different processing units. An implementation of HSA on FPGA is presented in [11]. [12] introduces an integrated compute/memory infrastructure for FPGAs, including support for speculative execution and more fine-grained dynamic scheduling than that defined in the HSA model.

Synchronization of tasks has been addressed by HSA [11] and OpenCL [13]. The common feature are barriers, which delay the processing of subsequent tasks until a condition is met. This can be used to enforce task-dependencies in the execution schedule.

III. TaPaSCo

The main purpose of the open-source Task-Parallel System Composer (TaPaSCo) framework is to facilitate the integration of FPGA-based accelerators into heterogeneous systems. To this end, TaPaSCo provides two main components: A software runtime library to interact with FPGA accelerators and a tool-flow to easily compose complete FPGA-based system-on-chip designs from one or multiple IP-cores. TaPaSCo supports a whole range of different FPGA platforms. Next to embedded FPGA-boards (e.g., Zynq, Zynq UltraScale+ MPSoC), TaPaSCo is available on a variety of data-center scale FPGA extension boards, such as the Xilinx AU280 board or the Bittware XUP-VVH platform. In recent releases, support for the Amazon EC2 F1 FPGA instances available in the AWS cloud has also been added to the TaPaSCo framework [2].

A. Runtime API

A TaPaSCo FPGA-based SoC design can comprise one or more instances of multiple different accelerator cores, called *Processing Elements* (PE). To interact with the FPGA accelerator SoC and the different cores, TaPaSCo provides a runtime API, which can be used from C and C++. It is responsible for data-transfers between host- and FPGA external memory and for launching jobs on the device. TaPaSCo uses a multi-layer approach for the runtime API (depicted in Figure 1), allowing the software interface for an application to be re-used across different target FPGA platforms.

The example code in Listing 1 shows how to launch a TaPaSCo job with the concise C++ API. The first step is to create a TaPaSCo object. As most accelerator jobs require some data, the next step is to prepare any buffer(s) for input and result data. If the data type is not trivial, the `makeWrappedPointer` function is required to associate a data size with the pointer. Unless a buffer is marked as being input/output-only, the default (used here) employs the same buffer for both purposes. Afterwards, a TaPaSCo job can be prepared. This takes a Kernel ID and any used data buffers as arguments. The Kernel ID represents the implemented function of a processing element. Elements with the same function therefore have the same Kernel ID. This call launches the job asynchronously, the Future obtained as result can be checked for the completion of the execution.

The TaPaSCo runtime translates the function calls to low-level driver calls and accesses to the registers on the FPGA.

The driver initialization happens when a `Tapasco` object is created. Then in line 14 of Listing 1, a TaPaSCo job is launched. It uses the given Kernel ID and in the example `buf` as a parameter. In the background, the runtime reserves a processing element, copies the content of `buf` to the memory of the FPGA card and starts the processing element. The call in line 18 is blocking until the processing element has finished. The runtime receives an interrupt, signaling the completion of the processing element and copies back the buffer(s) holding results. The application can then continue its execution.

B. Automatic Toolflow

Next to the runtime API, TaPaSCo also provides an automated tool-flow to compose complete FPGA SoC design from

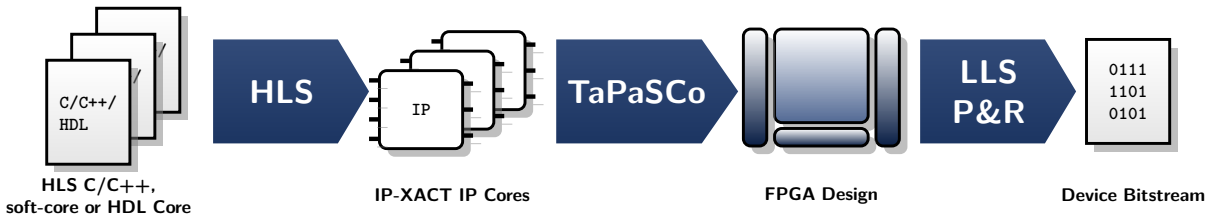


Fig. 2. TaPaSCo tool-flow, starting from the accelerator definition given as HDL-code, softcore IP, or C/C++-based HLS code. If necessary, HLS is automatically performed. After the user-specified composition of the architecture, a complete FPGA design for the requested target platform is automatically constructed, which will then be synthesized to a complete bitstream using the vendor's tool, e.g., Xilinx Vivado.

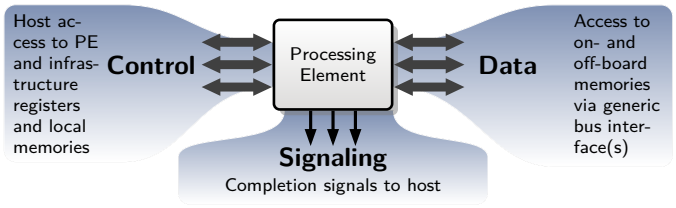


Fig. 3. Standardized interface for TaPaSCo processing elements.

the individual accelerator cores, depicted in Figure 2.

In a TaPaSCo-based FPGA design, the central unit of abstraction is the *Processing Element* (PE). Processing Elements can be defined in a variety of ways, including manual design with a hardware-description language (HDL), through C/C++ based High-Level Synthesis (HLS), which is also automated by TaPaSCo [14], or by importing existing software core CPUs, such as open-source RISC-V cores [15].

Independent of how a PE was defined, for the use with the TaPaSCo framework, it must always expose a standardized interface, referred to as *T-shape*, depicted in Figure 3. The T-shape has three main components: The control interface is used for communicating configuration information with the PE, in the current standard architecture of TaPaSCo this is realized through an AXI4 Lite Slave interface backed by a configuration register file.

The signaling component of the T-shape interface is also used for communication with the surrounding infrastructure and the host, e.g., for signaling completion of a computation job through a simple interrupt wire.

The last component of the T-shape interface, the data component, gives a Processing Element access to important infrastructure components, such as memories. In the current TaPaSCo standard architecture, the data component is realized through one or more AXI 4 Master interfaces that give the PE access to external DRAM or HBM2 memory typically found on data-center scale FPGA extension boards.

Implementing the T-shape interface for a processing element is usually straight-forward: As the interface was heavily inspired by Vivado HLS, IP cores generated by Vivado HLS are directly compatible with TaPaSCo's T-shape interface. For HDL-based PEs and soft-cores, a suitable interface or wrapper can easily be implemented, as demonstrated in [15].

After all PEs have been defined, an *Architecture* can be

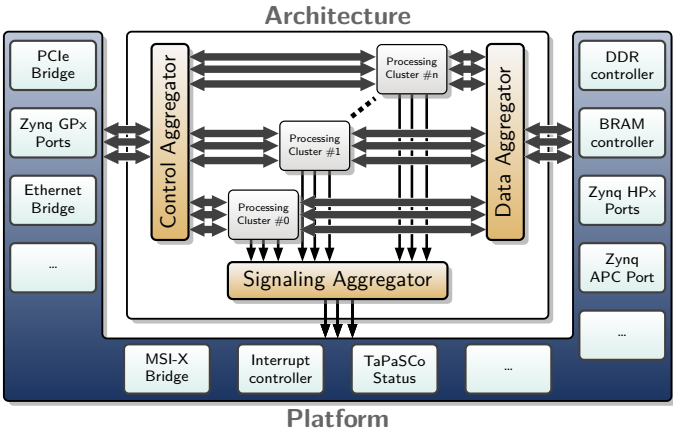


Fig. 4. TaPaSCo system-on-chip template with the platform-independent architecture surrounded by platform-dependent infrastructure.

composed from multiple different PEs and also multiple instances of each PE. Multiple instances of the same PE are subsumed in so-called *Processing Clusters*, and all PEs in all Processing Clusters are then connected to signal aggregators to form an Architecture as shown in Figure 4. The current TaPaSCo standard architecture is host-centric, i.e., PEs work completely independently from each other, cannot be connected directly (data-exchange is still possible via memory) and cannot synchronize with each other. Individual device jobs need to be synchronized by the host, if necessary.

A TaPaSCo architecture is still completely *platform-independent* and can be used on multiple different target platforms without modification. To create a full FPGA SoC-design, the architecture is automatically connected with the infrastructure components of the user-selected target platform (cf. Figure 4). The now-completed FPGA-design can then automatically be synthesized to a ready-to-use FPGA bitstream. To this end, TaPaSCo will pass the design information to the vendor's tools (e.g., Vivado in case of Xilinx platforms), which will eventually generate the bitstream. To further optimize the performance of the FPGA-design, TaPaSCo offers the possibility to run a fully automated *design-space exploration* (DSE), which will optimize the number of accelerator cores within the resource boundaries of the target platform and/or the design frequency.

As stated earlier, the current TaPaSCo architecture and exe-

TABLE I
REGISTER INTERFACE FOR A PROCESSING ELEMENT.

Offset	Function
0x00	Control signals
0x04	Global Interrupt Enable Register (GIER)
0x08	Interrupt Enable Register (IER)
0x0c	Interrupt Status Register (ISR)
0x10	Return value
0x20	Parameter #1
0x30	Parameter #2
...	Parameters

cution model is very host-centric. This can cause inefficiencies, if an application is composed from multiple small device jobs, as the high latency in the communication between host and FPGA (e.g., introduced by the PCIe interface) adds significant overhead to the device job launch. In the evaluation section, it can be seen that the launch latency is longer than 8 µs.

Therefore, in this work, some responsibility for job launch management is moved from the host-side software runtime to the FPGA hardware, allowing for *on-device job launch offloading*. To this end, the original runtime implementation is replaced and extended with a new Rust-based implementation (described in Section IV) and the SoC design is extended with a hardware-accelerated job dispatcher (cf. Section V).

IV. IMPROVED RUNTIME

The originally C-based TaPaSCo runtime was reimplemented in the programming language Rust, in order to improve maintainability, enable safer concurrency, and to reduce typical error sources [16]. As the Rust compiler does more checking at compile time, many sources of bugs in the old runtime could be avoided.

In the following, all required tasks of the runtime are explained:

1) Initialization: The first step when starting up a FPGA accelerator is to initialize all processing elements. As processing elements in TaPaSCo are based on the memory-mapped register interface of Vivado HLS, the control registers for interrupt enable (IER) and global interrupt enable (GIER) have to be set-up (see Table I). IER is used to configure interrupts on the *done* and *ready* state of the processing element. Global interrupts are activated with the GIER register, otherwise the interrupts configured by the IER register are not available for external units. Furthermore, depending on the FPGA platform, the DMA engine and interrupt controllers have to be initialized.

2) Job Launch: Before the launch of a job, all required parameters have to be prepared. Specifically, all data buffers have to be accessible by the processing element during the execution. This means that on a PCIe based platform, which typically has separate host and device memories, a DMA engine has to move data between host and FPGA card. Then, pointers to those memory locations and other parameters are assigned into the hardware parameter registers of the processing element. The actual launch of the processing element is performed by writing to the control register.

3) Job Finalization: The completion of the execution of the processing element is signaled by triggering an interrupt to the host. The interrupt handling on the FPGA is platform specific, but eventually, an interrupt is raised in the host Linux OS. The runtime acknowledges the reception of the interrupt to the processing element by writing into the hardware interrupt status register and can then continue with the execution.

These steps require information about the actual SoC composition. To this end, an FPGA-wide TaPaSCo status core provides an overview of all supported features, a list of the contained processing elements and further platform elements and their addresses on the FPGA. This allows for self-describing hardware without the need to store specific configuration details in the software stack.

The Rust runtime supports both C and C++ APIs, realized as a foreign function interface (FFI). The C++ interface is based on the generated code from the FFI and provides a higher level of abstraction than the C API by implementing wrapper functions. The Rust-provided C++ API is backwards-compatible with applications developed for the original version.

A breaking change on the interface between runtime and kernel driver was introduced for the interrupt handler. To relay the interrupt signal from the Linux kernel driver to a userspace application, the eventfd mechanism is now used. This simplifies the implementation considerably without incurring performance penalties. The userspace runtime just registers an eventfd file descriptor inside the TaPaSCo kernel driver, and can then receive the interrupts issued by the FPGA.

V. HARDWARE JOB DISPATCHER

For even higher job launch rates, the dispatch mechanism itself can be hardware-accelerated. Thus, we examined how to move a useful subset of the dispatcher functionality of the new Rust-based runtime (Section IV) onto the FPGA in the form of the Cascabel hardware dispatcher. The motivation here is to reduce communication latency between a host system and the FPGA accelerators. We aim for maximum performance, even at the expense of some of the additional flexibility provided by the full-scale software dispatcher.

The Cascabel hardware module, designed in Bluespec System Verilog consists of two parts: a queue to store the jobs to be launched, and the dispatcher to invoke those jobs on the processing elements. As seen in Figure 5, it is inserted between the host and the SoC-architecture holding the actual PEs.

A. Hardware Queue

The queue module has a functionality inspired by HSA [11]. It contains the job queue and provides a way for the host to enqueue jobs for execution. The queue is implemented as a BlockRAM and is memory-mapped into the host address space (Figure 5a). To be able to use this block of memory, additional hardware logic in the form of read and write pointers for the queue are required. As concurrent accesses to the registers can occur with a multi-threaded host software, both pointers have to be modified by atomic operations. E.g., When a queue read

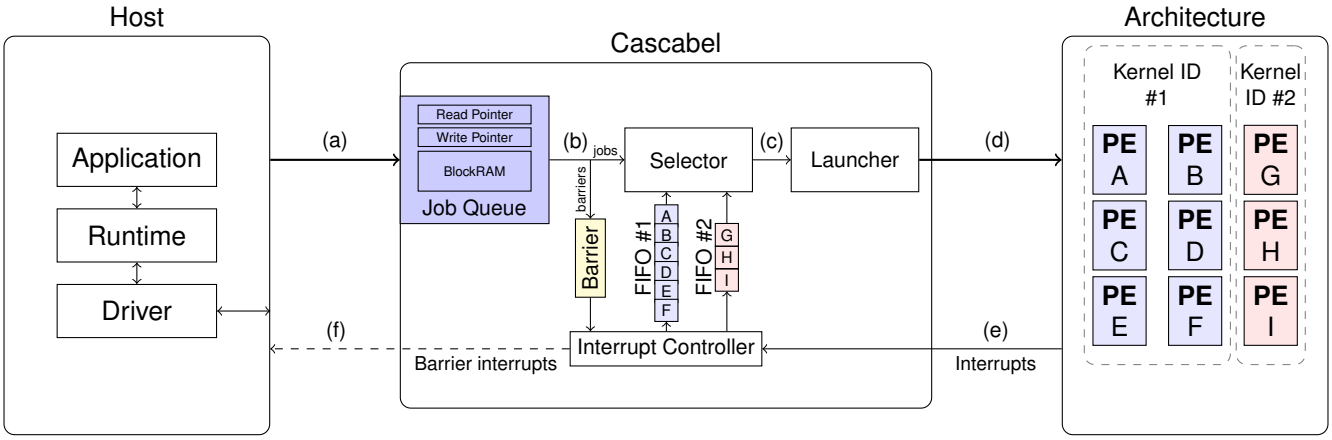


Fig. 5. Structure of Cascabel hardware dispatcher within a TaPaSCo system composition.

is performed, the current queue entry is returned and at the same time, the pointer register value is atomically incremented. This ensures that accesses to the queue are free of conflicts.

The queue is managed by the read and write pointers. Every queue element has a fixed size of 512 bit and can be of the type `Job` or `Barrier`. A Job is the standard TaPaSCo job, which is used to start a processing element. The associated queue entry contains the TaPaSCo Kernel ID, the number of parameters and up to four 64 bit parameter values. The Barrier is used for synchronization purposes, it blocks further execution until all previously launched processing elements have finished. This is required, for example, in the case that results stored in memory should be *reused* in subsequent jobs, and would otherwise result in conflicting memory accesses. The barrier element contains a parameter, which specifies whether the host should be informed of the barrier completion by sending a specific interrupt. This feature enables the hardware dispatcher to autonomously launch entire sequences of jobs and barriers *without* host intervention, notifying the host only once the entire job pipeline has completed. This behavior is achieved by setting the interrupt signal request only in the last barrier of the job sequence.

The job queue can be accessed as a memory-mapped device by the host to submit new jobs. On the dispatcher side, jobs are retrieved from the SoC-side of the queue for further processing.

B. Hardware Dispatcher Operation

The hardware dispatcher dequeues entries from the queue and then processes them (Figure 5b). If it is a `Barrier`, the processing of later entries is paused until all active processing elements have finished. Optionally, the barrier can raise an interrupt to signal completion of the barrier operation to the host (Figure 5f).

Otherwise, if the entry is a `Job`, the selector submodule has to select an idle processing element with the same Kernel ID as the job. This selector is FIFO based: For every Kernel ID, there exists a dedicated FIFO. Idle processing element IDs are stored in a FIFO. At system startup, every FIFO is initialized with all processing elements of the associated Kernel ID (this is the state of the FIFO contents shown in Figure 5). The Kernel ID from an incoming Job is then used to dequeue the ID of an idle processing element from the appropriate queue. If no idle processing element can be found, further execution is blocked until an idle processing element is available again. The dequeued entry, identifying an idle processing element for that Kernel ID, is then forwarded to the launcher submodule (Figure 5c), and is used to start the processing element for the Job: A state machine writes the parameters and the control registers of the processing element (Figure 5d). Execution of the Job is then begun.

On job completion, the finishing processing element issues an interrupt (Figure 5e) to the dispatcher. Without the hardware dispatcher, these completion signals would have been issued directly to the host. With the Cascabel hardware dispatcher, though, they are now evaluated *on the SoC* to re-enqueue the ID of the now idle-again processing element into the correct FIFO. Only if a Barrier entry *explicitly* requests host signaling will the Barrier's completion interrupt be forwarded to the host. As described before, we use this selective signalling to enable synchronization-free autonomous execution of the hardware accelerators as much as possible.

The dispatcher is dependent on the TaPaSCo system composition. As a result, the hardware module has to be generated for every composition. Automated generation is integrated into the TaPaSCo tool-flow.

With its performance-optimization, the hardware dispatcher is less flexible than the software-based one. For example, all memory allocation and DMA transfers still rely on the existing software runtime. In addition, the Cascabel currently has these limitations: (1) Cascabel-launched jobs may have at most four parameters. Supporting a larger number of parameters would require a way to store the variable length of parameter data outside of the dispatch queue. (2) Cascabel itself currently cannot initialize PE-local memories, such as load firmware into the instruction memories of programmable soft-core processing elements. However, Cascabel is able to dispatch jobs to PEs whose memories have been *preloaded* using the software

runtime. (3) Currently, the return value of a processing element is not interpreted in Cascabel. If desired, a return FIFO to the host could be added that would buffer all return values of completed jobs, to be sent all at once after the next Barrier interrupt.

The Cascabel hardware dispatcher is integrated into the platform-wrapper of TaPaSCo. From a logical point of view, it is located between the host and the accelerator architecture parts. The host accesses the job queue via an AXI interface (Figure 5a). Cascabel then invokes processing elements in the architecture with another AXI interface (Figure 5d). For the interrupts, it is vice-versa: interrupts from the processing elements in the architecture (Figure 5e) are handled in Cascabel. As described above, Cascabel signals the host using interrupts only if explicitly requested in a Barrier (Figure 5f).

VI. Evaluation

For the evaluation, the new runtime and the hardware dispatcher are compared to the previously existing runtime. Two different FPGA platforms are used: (1) the Xilinx Alveo U280 as a PCIe based FPGA for datacenter usage and (2) a AVNet Ultra96 as a multiprocessor SoC with embedded FPGA fabric. However, due to the versatility of TaPaSCo, any supported FPGA platform could be used. The host CPU for the U280 is an AMD EPYC 7351P with 16 cores, for the Ultra96 it is the on-chip quad-core ARM Cortex-A53 processor. The first part of the evaluation focuses on synthetic benchmarks to obtain measurements for the maximum possible performance. Then a more realistic application is implemented for the three different implementations (original C-based, new Rust-based and Cascabel hardware-accelerated dispatcher). Lastly, the resource usage of the hardware-based dispatcher is assessed for various configurations.

A. Job Launch Throughput and Latency

The first metric is the job throughput, which states the number of jobs scheduled within a timeframe. To measure this, a simple `counter` processing element is used. The PE uses a parameter to set the number of clock cycles to wait and then raises an interrupt afterwards. In the benchmark, the bitstream contains 16 counter PEs and the counters count just one clock cycle. The design frequency of this simple setup is $450\,\mathrm{MHz}$.

In Figure 6, the resulting throughput is plotted. It can be seen that all three versions benefit from launching jobs from multiple host-threads. With the reduced communication overhead of the hardware version, the maximum throughput can only be reached with a higher number of threads, as the computation on the host-side is now the limiting factor. This behaviour is even more apparent on the Ultra96, due to its slower host processor. The new software runtime has more than double the throughput of the old one. The Cascabel hardware dispatcher adds another factor of 3x, yielding a maximum throughput of over $6\,\mathrm{MJobs/s}$.

To measure the latency, a counter instance is used again. This time, a single job is launched, and the total time for starting, execution and handling the interrupt is measured. The execution time is subtracted from the final result to obtain only the launch overhead. For the hardware dispatcher, a barrier is inserted after the job to receive an interrupt. Due to variations in single measurements, an average over multiple runs is calculated. The measurements were collected as long as changes in the average are more than $10\,\mathrm{ns}$. The results in Table II show an improvement with the new software runtime, on the U280 the latency is less than half of the original value. In most cases, the hardware version has a slightly worse latency than with the new software runtime. With the barrier and additional on-chip latency, single job execution does not perform as well as with the new software runtime.

TABLE II
Average dispatch latency for single-shot kernels with various execution times.

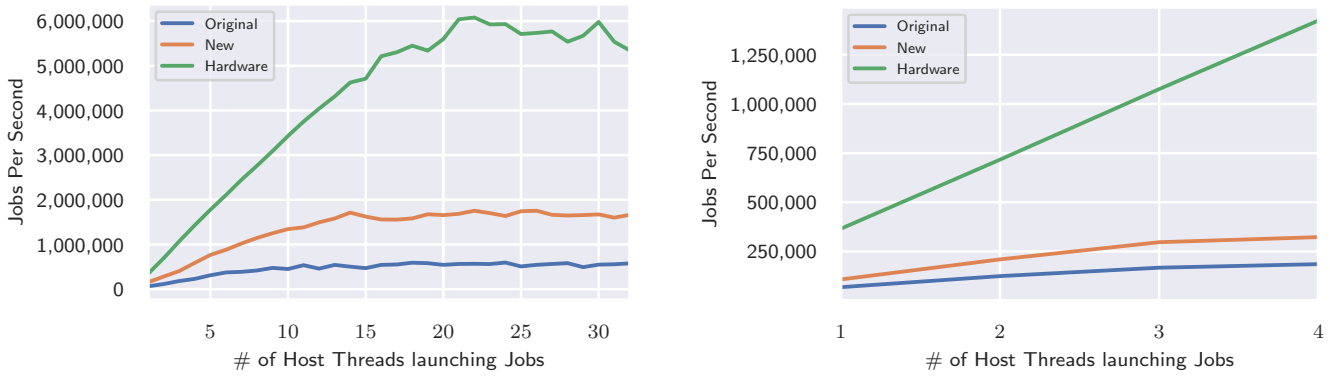
Cycles @ 450 MHz	Alveo U280 Original [µs]	Alveo U280 New [µs]	Alveo U280 Hardware [µs]	Ultra96 Original [µs]	Ultra96 New [µs]	Ultra96 Hardware [µs]
1	19.40	8.54	13.92	19.18	12.45	13.85
16	21.75	8.56	13.94	19.01	12.46	13.91
8192	22.26	10.33	12.64	19.13	12.42	11.88
2^{22}	158.23	45.46	49.15	118.75	91.52	90.30

Note that in all cases, the dispatch latency increases for jobs with longer execution times. We suspect that this slowdown is due to interactions within the host system, specifically the host-side TaPaSCo process being idle for longer periods of time, when waiting for the completion of longer-running compute jobs executing on the FPGA. This idleness might give the *Linux OS-level* scheduler cause to *context-switch* away from the TaPaSCo process to perform other tasks, and then incur another OS-level penalty to context-switch back to the TaPaSCo process once a hardware Barrier signals job completion to the host by interrupt. Similarly, the host CPU might start to *slow-down* the clock frequency for the core executing the waiting TaPaSCo process, incurring another delay to ramp the CPU core clock up again after the FPGA SoC has signalled completion. This misbehavior, though, has only limited impact in practice, as the increased dispatch latency is negligible compared to the longer compute job run-times.

With the latency measurement from the host, the on-chip launch latency cannot be measured directly. A lower bound can be approximated based on the job throughput, as the hardware internally launches all jobs sequentially. At $6\,\mathrm{MJobs/s}$, this results in an on-chip latency of $167\,\mathrm{ns}$.

B. Performance of Multi-Job Pipelines

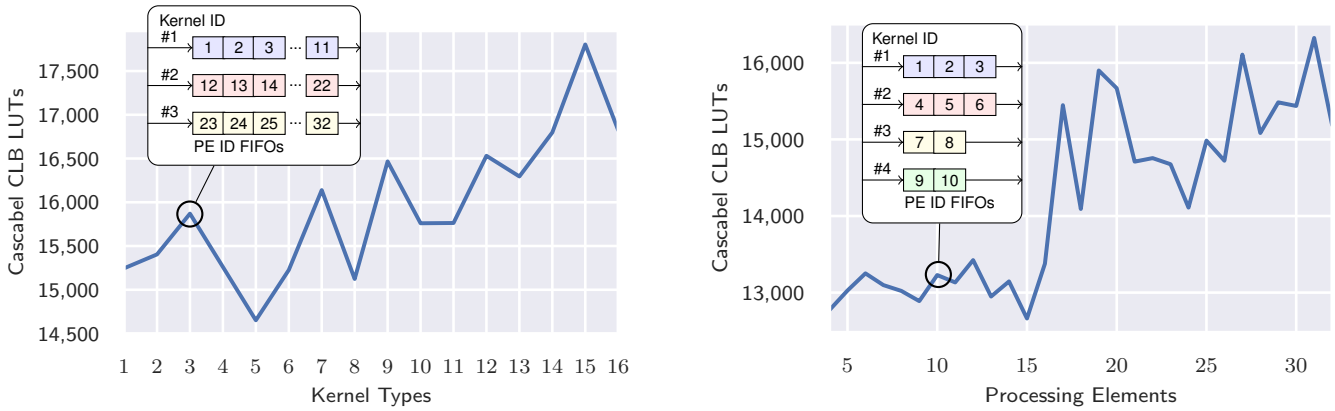
In this subsection, we examine a more complex application with three different kernels. The kernels used are arrayinit, arrayupdate and arraysum. The three kernels are implementing operations on arrays: arrayinit creates an array in memory and initializes it with values, arrayupdate updates the values in the array and arraysum calculates the sum over all array values, with array sizes of $32\,\mathrm{kB}$. With those three kernels, we demonstrate a simple pipeline application having data dependencies between the kernels.



(a) Alveo U280

(b) Ultra96

Fig. 6. Job throughput measured in jobs per second.



(a) Increasing number of different kernels, 32 PEs in total

(b) Increasing total number of PEs, four different kernels

Fig. 7. Scaling CLB LUTs overhead of Cascabel hardware dispatch for various configurations.

With the C and Rust software runtimes, a dependent job cannot be scheduled before the previous job has finished execution and returned to the host. With Cascabel, it is now possible to schedule dependent jobs *in advance*. By using barriers, the data dependencies will be maintained on the SoC without host interaction. Figure 8 shows a simplified sample schedule with just two processing elements per kernel, and inserted barriers. The different colors are representing batches of tasks. Within one batch, the exact binding of a job to the processing element is irrelevant. However, it has to be assured that arrayupdate is only operating on data completely initialized by arrayinit. To this end, the barrier feature is used. After two batches, complete utilization of all six processing elements is achieved.

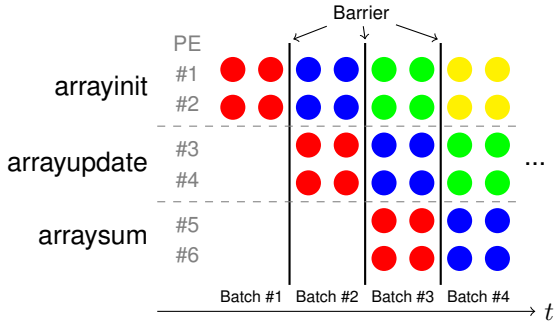


Fig. 8. Pipeline application utilizing Cascabel's barrier feature to maintain inter-kernel data dependencies on-chip.

As the hardware queue does not yet support reordering, the starting sequence of the jobs is important. If we were to issue all arrayinit jobs first, then all jobs for arrayupdate, and finally all jobs for arraysum, this would leave many processing elements unused. Thus, we currently have to rely on the software-side to ensure a suitable ordering when enqueuing the jobs into Cascabel for dispatch.

For our measurements here, we use an SoC having eight instances of each kernel, in contrast to just the two shown in Figure 8. These processing elements are clocked at a frequency of 400 MHz. Running 100,000 iterations of this pipeline on the Rust-based runtime takes 16.1 s with a single thread and 5.7 s with 8 threads. Using the Cascabel hardware dispatcher, this is reduced to 6.25 s for a single thread, and 4.42 s with 8 threads.

C. Hardware Resources and Performance

The resource overhead for Cascabel is analyzed here for a fixed clock frequency of $300\,\mathrm{MHz}$ and compared to a TaPaSCo composition without the hardware dispatcher. Figures are given for the U280. No major differences are expected for the Ultra96, as it has a similar FPGA fabric and the hardware module is the same.

In Figure 7, the additional CLB LUTs utilization of the Cascabel hardware core is shown. For all configurations, the numbers are below $2\,\%$ of the available resources (1,303,680 CLBs). All configurations require 20.5 RAMB36 BlockRAMs for the job/barrier queue and the dequeuing FIFOs (Figure 5b). As the hardware job dispatcher is automatically generated and customized for each SoC architecture, the resource overhead is kept to the necessary minimum and no resources are wasted.

The Cascabel hardware module is located between the platform-dependent host-interface and the architecture. In TaPaSCo, both of these parts run in different clock domains. The Cascabel hardware crosses into both clock domains: the queue submodule resides in the host clock domain, whereas the selector, launcher and interrupt controller submodules are in the clock domain of the architecture holding the PEs. The host is clocked at $250\,\mathrm{MHz}$ and meets timing closure. The maximum frequency for the architecture is only slightly impacted by the Cascabel module. All tested configurations could be clocked faster than $400\,\mathrm{MHz}$.

Comparing the results with related work is difficult, as published evaluation results are scarce. Connectal [9] only gives latency figures for one-directional transfers. The documentation of Xilinx XRT [17] states an overhead between $30\,\mu\mathrm{s}$ and $60\,\mu\mathrm{s}$, which can be interpreted as a measurement similar to the latency in this evaluation. HSA on FPGA [11] measured $58.1\,\mu\mathrm{s}$ for all dispatch steps, thus without handling the job completion. In both cases, the measurements are for PCIe based cards. Our best case latency is less than $9\,\mu\mathrm{s}$ for the new Rust based runtime.

VII. Conclusion and Future work

This work presented an improved job launch interface for TaPaSCo, an open-source framework that greatly simplifies the task of integrating FPGA-based accelerators into heterogeneous systems. By careful hardware/software-co-design of a new Rust-based software runtime and a hardware-accelerated dispatcher, job launch latencies and throughputs could be improved significantly over the previous software-only implementation. Even though the FPGA footprint of hardware dispatcher is very small, Cascabel can still provide a 3x improvement in job throughput over the highly optimized new Rust-based software dispatcher.

The hardware based dispatcher is currently a proof-of-concept implementation. Beyond lifting the limitations described above, two additional features seem highly desirable. First, some form of scheduling better than "FIFO" should be implemented, e.g., by allowing job re-ordering in the queue. At the moment, performance may be lost when inserting the jobs in an unsuitable ordering. At the cost of higher resource utilization, a better scheduler could both improve performance and reduce the need for manual tuning of the job ordering.

A second improvement could be the support for *on-chip* job creation. The hardware dispatcher could provide an SoC-side enqueuing port, where the processing elements themselves could issue new jobs. This would again reduce the number of high-latency interactions with the host, and allow more flexible pipelines to be realized.

For another improvement, *Inter-PE communication* could be implemented. Data could then flow directly between processing elements, eliminating even more transfers between the host and processing elements.

The new software runtime is already publicly available for testing on Github https://github.com/esa-tu-darmstadt/tapasco/tree/feature/NewRuntime. The hardware dispatcher will be available as an optional feature in the next release of the open-source framework TaPaSCo.

References

[1] A. Engel, B. Liebig, and A. Koch, "Energy-efficient heterogeneous reconfigurable sensor node for distributed structural health monitoring," in *IEEE Proc. Conference on Design & Architectures for Signal & Image Processing, 10-2012*, IEEE, 2012.

[2] M. Ober, J. Hofmann, L. Sommer, L. Weber, and A. Koch, "High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud," in *Fifth International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2019.

[3] J. Korinth, D. de la Chevallerie, and A. Koch, "An open-source tool flow for the composition of reconfigurable hardware thread pool architectures," in *The 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2015.

[4] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems," in *International Symposium on Applied Reconfigurable Computing (ARC)*, 2019.

[5] Xilinx, Inc., *Xilinx Runtime Library (XRT)*. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/xrt.html (visited on 08/20/2020).

[6] Intel Corporation, *Open Programmable Acceleration Engine*. [Online]. Available: https://opae.github.io/ (visited on 08/20/2020).

[7] Y. Tang and N. W. Bergmann, "A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1254–1267, 2015.

[8] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[9] M. King, J. Hicks, and J. Ankcorn, "Software-driven hardware development," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15, Monterey, California, USA: Association for Computing Machinery, 2015, pp. 13–22, ISBN: 9781450333153. DOI: 10.1145/2684746.2689064. [Online]. Available: https://doi.org/10.1145/2684746.2689064.

[10] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar, "Dynamic Task Parallelism with a GPU Work-Stealing Runtime System," in *Languages and Compilers for Parallel Computing*, S. Rajopadhye and M. Mills Strout, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 203–217, ISBN: 978-3-642-36036-7.

[11] M. Reichenbach, P. Holzinger, K. Häublein, T. Lieske, P. Blinzer, and D. Fey, "Heterogeneous Computing Utilizing FPGAs," *J. Signal Process. Syst.*, vol. 91, no. 7, pp. 745–757, Jul. 2019, ISSN: 1939-8018. DOI: 10.1007/s11265-018-1382-7.

[12] H. Gädke-Lütjens, B. Thielmann, and A. Koch, "A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation," in *IEEE Intl. Conf. on Field Programmable Logic and Applications (FPL), Milano (I)*, IEEE, 2010.

[13] Khronos Group, *The OpenCL Specification. Version 2.0*.

[14] L. Sommer, J. Korinth, and A. Koch, "OpenMP Device Offloading to FPGA Accelerators," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017.

[15] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch, "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)*, IEEE, 2019.

[16] Rust Team, *Rust Programming Language*. [Online]. Available: https://www.rust-lang.org/ (visited on 08/20/2020).

[17] Xilinx, Inc., *Vitis Unified Software Development Platform 2020.1 Documentation - Optimizing the Performance*. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/optimizingperformance.html (visited on 08/20/2020).