Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-022-01759-2

On-Chip and Distributed Dynamic Parallelism for Task-based Hardware Accelerators

Carsten Heinz[1] · **Andreas Koch**[1]

Received: 2 October 2021 / Revised: 27 February 2022 / Accepted: 6 April 2022

Abstract
The open-source hardware/software framework TaPaSCo aims to make reconfigurable computing on FPGAs more accessible to non-experts. To this end, it provides an easily usable task-based programming abstraction, and combines this with powerful tool support to automatically implement the individual hardware accelerators and integrate them into usable system-on-chips. Currently, TaPaSCo relies on the host to manage task parallelism and perform the actual task launches. However, for more expressive parallel programming patterns, such as pipelines of task farms, the round trips from the hardware accelerators back to the host for launching child tasks, especially when exploiting data-dependent execution times, quickly add up. The major contribution of this work is the addition of on-chip task scheduling and launching capabilities to TaPaSCo. This enables not only low-latency *dynamic* task parallelism, it also encompasses the efficient on-chip exchange of parameter values and task results between parent and child accelerator tasks. For larger distributed systems, the dynamic launch capability can even be extended over the network to span multiple FPGAs. Our solution is able to handle recursive task structures, and is shown to achieve latency reductions of over 35x compared to the prior approaches.

Keywords FPGA · Runtime · Task launching · Parallel computing

1 Introduction

FPGAs have become widely available as accelerators in computing systems. As more and larger applications are being offloaded to FPGAs, the required hardware designs are getting more complex. However, applying typical approaches from software engineering, such as divide-and-conquer, or code-reuse, to reduce complexity, is still a challenge. For example, splitting a large application into multiple cooperating smaller accelerators, such as in the well-known *farm* parallel pattern [1], often results in increased communication overhead between the host and the FPGA.

With highly-compute intensive domains such as AI/ML, not only does the complexity of the individual accelerators grow, but the computing demand is so large that the workload needs to be *distributed* across many compute nodes. Thus, there is a need to efficiently load-balance a high-rate of task launches between the nodes of a distributed system, and quickly communicate task parameters and results among nodes.

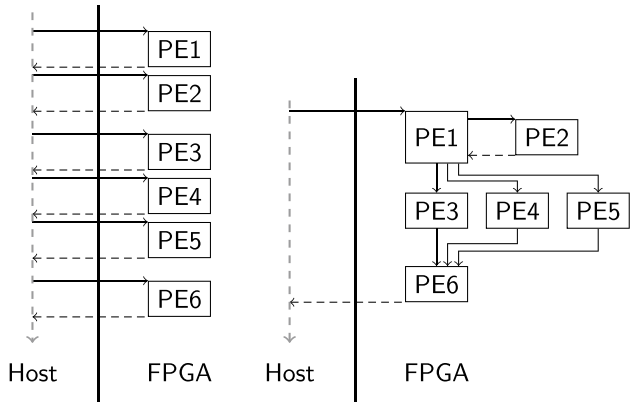
Our work addresses these challenges by adding hardware support for fine-grained task scheduling to the TaPaSCo framework for reconfigurable computing [2]. This new feature enables low-latency interactions directly between processing elements, even across the network in distributed systems, without the need for host involvement. It significantly reduces the number of host/accelerator interactions, as shown in Fig. 1. Furthermore, the new capability reduces development effort and the required time for implementing heterogeneous computing systems without sacrificing performance. Our approach also enables the use of more expressive computing structures, such as recursion, across resource-shared accelerators. The same mechanisms can be used even across a network with multiple FPGAs, with integrated load-balancing facilities ensuring an even use of processing elements across the nodes.

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

Andreas Koch
koch@esa.tu-darmstadt.de

[1] Embedded Systems and Applications Group, Technical University of Darmstadt, Darmstadt, Germany

Published online: 29 April 2022



(a) 12 Transfers **(b)** 2 Host-PE Transfers

Figure 1 Host and PE interactions, with (**a**) the existing host-centric model, and (**b**) the new on-chip dynamic parallelism.

This work extends our previous research described in [3]. The primary contribution here is the support for networking, with secondary contributions being the description of the programming model for PEs, and an extended evaluation on a more realistic near-data processing application.

The rest of this paper is organized as follows: Sect. 2 introduces the heterogeneous computing architecture of the framework TaPaSCo, which is used as a basis for this work. The implementation of our new Cascabel 2 architecture is presented in Sect. 3 and evaluated in Sect. 4. The paper closes with a brief survey of related work in Sect. 5 and concludes in Sect. 6.

2 Heterogeneous Computing Architecture

The open-source TaPaSCo framework [2] is a solution to integrate FPGA-based accelerators into a heterogeneous computing system. It addresses the entire development flow by providing an automated toolflow to generate FPGA bitstreams, and a runtime and API for the interaction of a host application with the accelerators on the FPGA. The resulting SoC design consists of the *Processing Elements* (PE) and the required infrastructure, such as interconnect and off-chip interfaces (e.g., host, memory, network). The PEs are instances of the actual hardware accelerators, and can be provided to the system either in an HDL or as C/C++ code for High-Level Synthesis (HLS). TaPaSCo realizes hardware thread pools, each having a set number of PEs to perform the same task. Thus, a human designer or an automated design-space-exploration tool can optimize how many PEs are to be provided for a specific function, optimizing, e.g., for maximum task throughput.

A key feature of TaPaSCo is its support for many different hardware platforms. The first category of platforms are *reconfigurable* system-on-chips with an attached FPGA region. In these architectures, the CPU and the FPGA region share the same address space and both parts have various communication channels for a tight coupling. TaPaSCo supports the older Xilinx Zynq-7000 series and the more recent Zynq UltraScale+ MPSoC (PYNQ-Z1, Ultra96, ...).

The second category are PCIe-based accelerator cards for compute systems (Xilinx VC709, Alveo U280, ...). Direct communication between CPU and FPGA uses the PCIe-bus. The cards have their own off-chip / on-board memory, thus, a DMA engine handles all memory transfers.

A third dimension comes into play when *distributed* systems built from such nodes are considered. A common use-case for distributed FPGA systems is high-performance training in machine learning [4].

This wide range of supported platforms, ranging from small, low-cost FPGAs to high-performance data-center cards, allows a user to select the suitable platform for a given application and enables quickly scaling-up or -down the platform in the development stage or later during deployment. Without any changes to software or the PE implementations, all supported platforms can be utilized. The extension presented in this work also maintains the high portability and thus can be used with all existing platforms.

In its initial version, TaPaSCo employed a software runtime to dispatch a task to a suitable, currently idle PE. Recently, TaPaSCo was sped-up by moving part of this dispatching process from software to hardware. The resulting Cascabel extension [5] employs a hardware queue, which accepts the task requests from the host. The task dispatch (finding a suitable idle PE) and the launch, including the transfer of task parameters and the collection of results to/from the selected PE, is now handled on-chip. This off-loading of the task dispatch decouples the software application on the host side from the PEs on the FPGA. The evaluation has shown that a higher job throughput is achievable, however, with the penalty of an increased latency.

In this work, we present Cascabel 2, which extends the prior version by now allowing the PEs themselves to autonomously launch new tasks without the need for host interaction. This capability is often called *dynamic parallelism*, e.g., in context of GPUs, where threads are able to launch new child threads themselves. The main goal of direct on-chip task launches is to reduce the latency, resulting in task launches with both low latency and high throughput.

Figure 2 shows the Cascabel 2 core in a typical FPGA design. In addition to the connection to a Host, the Network is introduced as a new external connection. On the FPGA itself, we now have stream connections for communication between PEs and the Cascabel core.

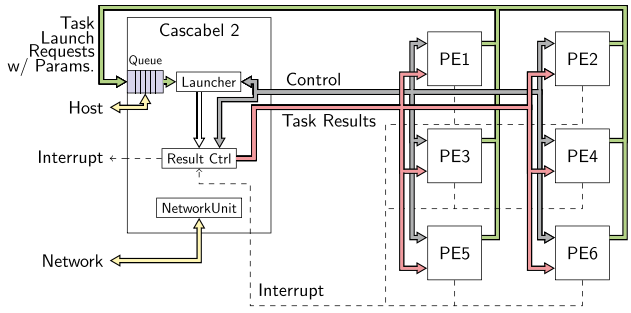


Figure 2 Cascabel 2 hardware dispatcher/launcher and its AXI Stream connections to the PEs for accepting task requests and distributing task results.

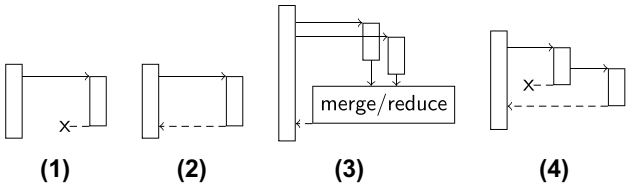


Figure 3 Return value handling: (1) Discard, (2) return-to-parent, (3) merge/reduce-to-parent, (4) return-to-grandparent. **x** indicates a discarded result.

3 Implementation

3.1 Control and Data Flows

The on-chip dispatch/launch functionality should be as powerful as the original software solution. Thus, it encompasses not only the actual dispatch/launching of child tasks, but also the passing of parameters from the parent to the child as well as the retrieval of the child result back to the parent task. We thus require bi-directional communication to perform this exchange.

The architecture is shown in Fig. 2. In addition to the regular TaPaSCo interfaces for PE control and interrupt-based signalling, two AXI4 streams are used to enable dynamic parallelism: A 512bit stream, shown in green, flows from the PEs to the Cascabel 2 unit, and carries new launch requests, including child task parameters. A second 64bit stream, shown in red, flows from the the Cascabel 2 unit back to the PEs and transports the task result, which is generally a single scalar value. Note that these widths are configurable, and can be matched to the application domains, such as a result consisting of a two-element vector of single-precision floats. Also, the Cascabel 2-interface is completely optional. If PEs do not require the dynamic parallelism, no superfluous hardware will be generated. To connect the Cascabel 2 unit with all streaming interfaces on the PEs, AXI4 streaming interconnects are used. As those are limited to 16 ports each, for a larger system, a *hierarchy* of interconnects is automatically created. The interconnects also provide the option to insert register slices for improved timing.

Cascabel 2 supports the two existing methods of transferring data in TaPaSCo: pass-by-value and pass-by-reference. The former is a parameter with a scalar value, the latter is a parameter containing a reference to a memory location for larger data sizes. The software runtime is responsible for memory management.

3.2 On-Chip Dispatch and Launch

Cascabel relies on internal queues for managing incoming tasks and idle/busy PEs and also provides advanced inter-task scheduling operations such as barriers. Adding the dynamic parallelism requires only very few changes here for Cascabel 2. Mainly, the existing memory-mapped interface used by the host to submit tasks for execution into the relevant queues is extended with the stream-based interface used by the PEs to submit task launch requests. For launches, the rest of the operations proceeds as in the initial Cascabel [5].

3.3 Handling Child-Task Return Values

Since tasks in TaPaSCo generally have return values, Cascabel 2 must be able to handle these as well. Compared to the dispatch/launching mechanisms described in the previous section, this requires greater changes in the Cascabel unit and the SoC architecture, especially since different execution paradigms need to be covered by the mechanisms. As shown in Fig. 3, Cascabel 2 supports four ways of handling child task return values, which will be discussed next. Note that for the methods 2) to 4), the launches can occur synchronously (parent task waits for child result to arrive) *or* asynchronously (parent tasks continues after launching child task).

3.3.1 Discard Child Result

Not all PEs actually make use of the return values of child tasks, or require them for synchronization purposes. An example for this would be a PE whose child tasks provide their results elsewhere (e.g., as outgoing packets on a network port).

3.3.2 Return-to-parent

In general though, parent tasks will be interested in the return values of their child tasks, if only for synchronization purposes ("child task has finished and updated shared state"). As TaPaSCo supports out-of-order completion of

tasks, we want to retain this capability for the dynamic inter-PE parallelism. In this mode, the child task's return value is sent back to the PE executing the parent task. As shown in Fig. 4, the return value can be configured to be sent alone (a), accompanied by the producing child's task ID, either in the same (b), or a separate bus transfer beat (c), to support out-of-order completion of child tasks.

3.3.3 Merge/Reduce-to-parent

For some parallel patterns, such as a task farm, the results of multiple worker PEs must be collected, e.g., in preparation of a reduce operation. To this end, Cascabel 2 provides infrastructure to perform this *merging* in dedicated hardware. When configured, the child task results produced in parallel by multiple worker PEs in the farm will be buffered in BlockRAM, which in turn is then provided to dedicated PEs for performing the reduction/collection operations. Once all merge/reduce tasks have completed, their final result is passed back to the parent task, which in itself may be another merge/reduce PE task.

3.3.4 Return-to-grandparent

For some parallel patterns, results are not required in the parent of a child task, but higher up in the task hierarchy. Cascabel 2 supports this by allowing a child task to *skip* its parent task when returning results, and instead provide its result to its *grandparent* task. Note that if the grandparent task was also launched in this mode, which can be cascaded in Cascabel 2, yet another level in the task hierarchy will be skipped, quickly propagating the child task's result up even further in the task hierarchy. A practical use for this capability will be demonstrated in Sect. 4.2.

The logic for realizing the different return actions is implemented in the *Return Ctrl* block (shown in Fig. 2). As new tasks are launched, the *Launcher* block forwards the associated return action to the *Return Ctrl* block, to be performed later upon task completion. When the Cascabel 2 unit receives interrupts from the PEs indicating the completion of a task, it internally looks-up the associated return action provided earlier. In all cases, except for the *discard child result* action, the first step is to read the result value from the PE. Further processing is dependent on the selected action: it either forwards the result to the (grand-)parent, waits for additional return values, or issues a new merge task using the Cascabel 2 unit. Optionally, a task can specify that an interrupt should be raised and sent to the host. This will generally be done only after an entire set of tasks has been successfully completed in hardware.

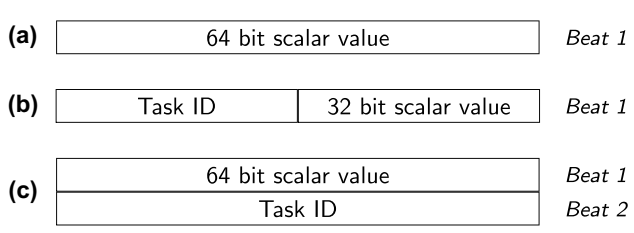


Figure 4 Stream data formats for child task return values.

3.4 Network Launches in Distributed Systems

In addition to on-chip launches, our work provides the option to use a network for launching tasks directly between the FPGAs in distributed systems. For this, an adapted version of the on-chip streaming interface is connected to a *NetworkUnit*. The network protocol is based on Ethernet frames, which are parsed within the *NetworkUnit*. Received frames are filtered for the correct destination MAC address and converted to launch requests analogously to the streaming based interface. On the return trip, PE results are encapsulated within Ethernet frames, and sent back to the launch originator via network.

We chose Ethernet for its low point-to-point latency of just 0.9 µs, and since it suffices to connect typically-sized FPGA clusters, which rarely exceed multiple racks in size. However, for even larger (or more distributed) systems, the *NetworkUnit* could be extended by a TCP/IP stack such as Limago [6]. This would incur longer communication latencies, though. Measurements in such a setup are 1.8 µs for UDP and 2.6 µs for TCP connections [7]. Note that in setups including switches, their latency must also be included in the total calculation.

Even shorter latencies would be possible using direct FPGA-FPGA links based on the Xilinx Aurora protocol [8], which achieves latencies below 0.5 µs. However, with its pure point-to-point nature, it cannot be run over switches, and thus permits only limited scaling of the distributed system.

The required FPGA logic for the physical layer is implemented by a feature already present in TaPaSCo, which provides support for small form-factor pluggable (SFP) transceivers. The feature instantiates the required Ethernet IP cores and provides data streams for sending and receiving data, which are then connected to the *NetworkUnit*. On supported FPGA cards, this allows for the use of 10G (SFP+) or 100G (QSFP28) network interfaces.

This underlying hardware communications layer is leveraged to implement the cross-chip task-launching mechanism. The mechanism re-uses the on-chip streams, but instead of writing the requested task into the local queue, a network request is generated. The Cascabel 2 core then uses a metric to decide whether to submit the request to the network or to the local queue. The current simple metric is just based on the kernel ID: if the local FPGA does not have a processing element matching the kernel ID of the

requested task, then it forwards the request to the network. The reason behind this decision is that a network transfer incurs a large latency overhead. In many cases, that communication overhead will outweigh the penalty of waiting for a local PE to become available on a heavily loaded node. Thus, remote launches are only used for tasks that cannot be performed locally.

For addressing on the network, Cascabel 2 employs Ethernet MAC addresses. Each FPGA has a unique address, which can for example be based on a unique identification of the hardware such as the `DNA_PORTE2` primitive on Xilinx FPGAs. To map the available kernel IDs on remote FPGAs, a data structure is used for fast lookup. Furthermore, this data structure stores the load factor at the remote FPGA. To this end, each FPGA broadcasts the current load factor of all its kernel IDs present on the FPGA at regular time intervals . Based on these broadcasts, the data structure is initialized on startup, and regularly updated afterwards.

The data structure is designed to accommodate up to 1024 MAC addresses, which is sufficient for many practical data center deployments. The structure consists of three parts: (1) a map from kernel ID to list offset, (2) a list of MAC address indices and load factors, grouped by kernel ID (called KML), (3) a list of all available MAC addresses. The KML groups the entries by subdividing the list into parts for each kernel ID. The end of such a list is indicated either by an empty slot or an entry, which is marked as the last element. The length of the individual lists is configurable according to the desired network setup. Therefore, the indices are not directly derivable from the kernel ID and need to be stored in the separate map. If a new task should be launched via network, the first step is to lookup the list offset for this kernel ID. This gives us the start address of the KML list, where all MAC addresses associated with this kernel ID are stored. The hardware iterates over this list until the end marker is reached. Afterwards, the minimum load factor is known and the full MAC address can be looked up by using the MAC address index from the KML list. In the example shown in Fig. 5, kernel ID 1 has an offset of 2. In the KML list, we can now see that the list at position 2 contains indices 1 and 4. Here, one would prefer the MAC located at index 4 over that at index 1 due to lower load.

In most cases, iterating through the KML list is a reasonable choice, as the additional latency for iterating over the entries is negligible compared to the network latency. This might not be the case in a scenario, where a specific kernel ID is available on most or all 1024 FPGAs. Here, the latency to iterate through the list can become as large as the network latency itself, effectively doubling the total latency. A non-optimal performance improvement could be realized by just iterating on a random subset of the list. For this, the length of the list needs to be known. Either it is stored explicitly in the map of list offsets, or the offset to the next kernel ID is used to calculate the end of the list.

With these mechanisms, no central coordinator is needed. For incoming requests, Cascabel 2 stores the source MAC address and uses that, in turn, as the destination address when the task's result frame is sent.

This network communication can be utilized to implement large accelerator farms, which cannot be accommodated using just a single FPGA. From the point-of-view of an individual processing element, this distributed processing is completely transparent.

3.5 Programming Abstraction for Distributed Processing Elements

We allow PEs to interact with the distributed infrastructure using Vitis HLS and Bluespec System Verilog APIs. From the infrastructure viewpoint, a PE has to add the required AXI4 streaming interfaces. Those interfaces are supported in both languages, e.g., in Bluespec via a library such as BlueAXI [9]. Instead of forcing designers to manually fill in the fields of the data structure, Cascabel 2 provides a higher-level API.

In Bluespec, the API is provided through an interface description and the associated module implementation. For the Cascabel 2 functions, it provides methods to launch tasks and to receive results. Figure 6 shows a processing element with the relevant parts of a task launch within a Bluespec module. First, the `mkTaPaSCo` module is instantiated. Then, the new task is launched in the first rule with the single argument `data`. With this, the task gets enqueued on the stream interface. The result of the task is fetched in

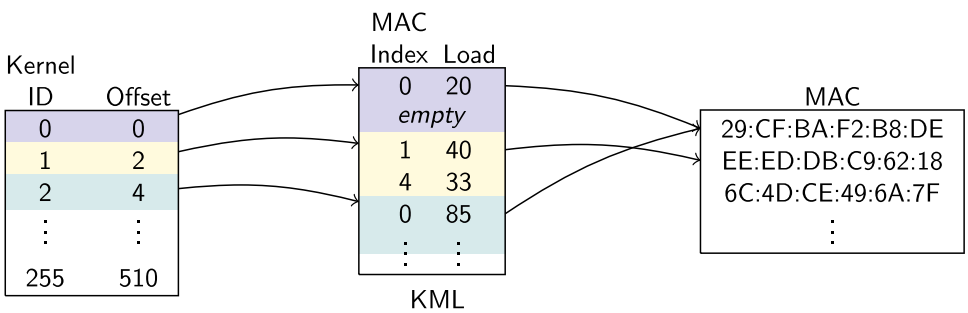


Figure 5 Data structure for MAC address lookup.

Figure 6 Code snippet showing a processing element with an on-chip task launch in Bluespec.

```
/* Processing Element in Bluespec */
module mkMyProcessingElement (PE_Interface);
    /* interface and module declarations omitted */

    let tapasco <- mkTaPaSCo();

    rule launch;
        let data <- alu.get(); // some input from PE-local logic
        tapasco.launch1(PE_ID, data); // on-chip launch with 1 parameter
    endrule

    rule result;
        let r <- tapasco.result(); // get the result
        outFifo.enq(r); // insert into own logic
    endrule

    // pass through AXI streaming interfaces
    interface maxi = tapasco.launch_stream;
    interface saxi = tapasco.result_stream;
endmodule
```

the second rule. Lastly, the AXI stream connections need to be forwarded into the `mkTaPaSCo` module. For the developer, the scheduling behaviour is the same as with regular modules and the Bluespec compiler ensures that rules only execute on valid data. As a consequence, the second rule executes only when a valid result is received. The merge/reduce-to-parent mechanism is supported in a similar way, albeit requiring additional information about the merge/reduce kernel ID.

In HLS, the API is provided through a header file. The functions defined in it are similar to those of the Bluespec implementation. Instead of managing all streaming operations (like in the Bluespec version), the functions return the values required to transmit. Handling the streaming operations in the HLS kernel itself allows for a tighter integration and may improve results of the HLS compiler. State information, e.g. for the merge/reduce-to-parent mechanism, is stored in a handle and passed on to the functions.

3.6 Limitations

Cascabel 2 is realized as a custom hardware module and optimized for performance. Thus, even with the provided customization options for each specific PE layout, Cascabel 2 does not reach the complete flexibility of the host-side software-only dispatcher. This section discusses the design decisions and the resulting restrictions.

In terms of arguments, Cascabel 2 by default supports values of 64bit. This can either be a scalar value, or a pointer to a memory location. Memory management is handled in the TaPaSCo software API on the host side. At this stage, it is thus not possible to dynamically allocate PE-shared memory for on-chip launched tasks. Instead, memory pre-allocated on the host-side could be used. The number of task arguments is currently limited to up to four arguments, which is sufficient for typical applications. Due to the latency optimization, all four arguments are passed in *parallel* in a single beat over the 512bit-wide launch request interconnect. If more arguments are required, this would either require widening the bus, issuing multiple beats, or passing the arguments via external PE-shared memory, such as on-chip HBM or on-board DDR-SDRAM.

As in all practical implementations (hardware or software), the achievable recursion depth in Cascabel 2 is limited by the capacity of the memory holding the "call stack". Cascabel 2 relies on on-chip BlockRAM to hold the call stack, again aiming for low latencies. The memory capacity used for this purpose can be configured, but will by necessity be much smaller than the DRAM-based main memory call stacks used in software recursion.

In addition, as TaPaSCo PEs are generally not multi-threaded or even re-entrant, a recursive call will always be executed on *another* PE, blocking the calling PE for the duration of the sub-task execution. For example, with recursive task launches following the *Return-to-parent* pattern, each recursion level will lead to one PE becoming blocked, thus limiting the recursion depth to the total number of PEs available on the SoC to execute this task.

4 Evaluation

Our evaluation system is a Xilinx Alveo U280 FPGA card in a server with an AMD Epyc Rome 7302P 16-core CPU with 128 GB of memory. All FPGA bitstreams have a 300MHz design clock and are synthesized in Vivado 2020.1.

4.1 Latency

The key goal when performing hardware-accelerated launches is a low latency. For evaluating this, we use the on-chip launch interface and measure the required clock cycles from writing the task launch command for an immediately returning (NOP) task to the Cascabel 2 launch-command stream, up to when the parent task receives the result value from the child task. This approach follows the conventions established in the HPC community for benchmarking task-scheduling systems, e.g., in [10]. This operation takes 62 clock cycles in total, which at the design frequency yields a time of 207 ns for a complete launch-and-return. When performing the same operation using the host-based software-only scheduler, it takes 7.41µs. Using the hardware-assisted software scheduler [5], which is optimized for task throughput instead of task latency, requires 8.96µs. Thus, Cascabel 2 yields a latency gain of 35x compared to the software-only scheduler, and a gain of 43x compared to the hardware-assisted Cascabel 1 scheduler.

For measuring the network latency, we used a switch-based network setup (see Fig. 7). FPGAs and the server are connected to a 100G switch (Celestica Seastone DX010) and all frames pass through this switch. Direct connections would be possible and should result in lower latencies, however, the switch-based setup is more scalable when increasing the number of FPGAs, as described above. We measure the time from sending out an Ethernet frame until the result frame is received. In the first measurement, we utilize our *server* to send launch requests, and receive back results via network. On average, our sample application achieves a latency of 16.1µs. Most of this time is spent on the network side, specifically on the host server. Hardware measurements have shown that fewer than 100 clock cycles (at the network clock domain running at 322MHz) are actually required for the operation within the FPGA logic to receive a packet and send out the result. This is, as expected, slower than the purely local on-chip launches due to high network overhead in the Linux kernel.

The situation improves considerably, though, when taking the software-based server out of the loop, and performing FPGA-to-FPGA launches across the switch. In that scenario, both the send and receive processing is completely handled in hardware. This results in a reduced latency of just 2.4µs, which is roughly an order of magnitude slower than the purely local on-chip launches on a single FPGA, but already highly competitive with the ≈4µs of round-trip time achievable with hardware-accelerated network adapters for MPI on RoCE via 100G Ethernet [11]. Note that with Cascabel 2, network launches in the distributed system are now roughly 3x *faster* than local software-based launches originally were on the host.

Figure 8 summarizes the measured latencies. Cascabel 1 as the predecessor of Cascabel 2 provides an hardware-offloaded scheduler, which is still connected via PCIe. The hardware module achieves higher throughput at the cost of an increased latency compared to the pure software solution. The on-chip communication of Cascabel 2 skips the PCIe connection and thus can reach those low latencies. Network communication in Cascabel 2 has considerably reduced latency compared to software-based execution. In addition to the reduced latency, the on-chip scheduling of tasks avoids the high jitter of both of the software-in-the-loop solutions, caused by the PCIe connection between the host and the FPGA board. The same applies for switched FPGA-to-FPGA network communication, when the server is not involved.

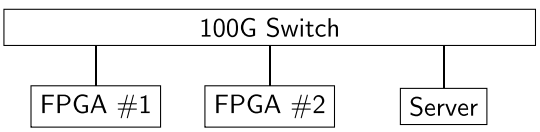


Figure 7 Network topology used for Server-to-FPGA and FPGA-to-FPGA communication.

4.2 Recursion

To stress-test the advanced task management capabilities described in Sect. 3.3.3 on a simple example, we show a *recursion-intensive* approach of computing the Fibonacci sequence, which is defined as

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = f(2) = 1$$

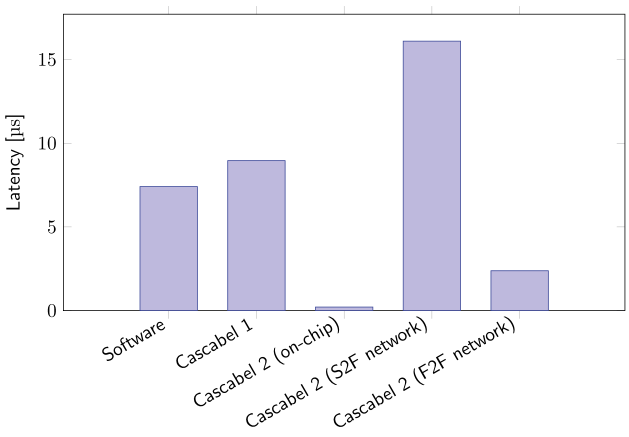


Figure 8 Comparison of launch latencies: host-side software-only, throughput-oriented hardware-assisted software with Cascabel 1, latency-oriented hardware-only with Cascabel 2, switched network-based launches from server-to-FPGA (S2F) and from FPGA-to-FPGA (F2F).

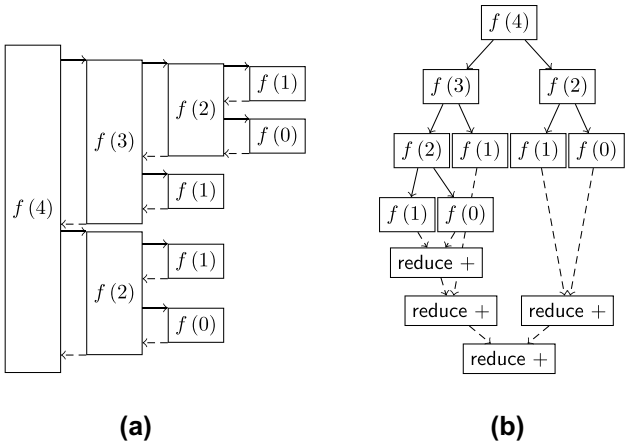


Figure 9 Recursive Fibonacci computation: (**a**) naive synchronous execution, (**b**) asynchronous mode with transitive return-to-grandparent value passing and merge/reduce.

When implementing this computation naively without the merge/reduce support of Cascabel 2, as shown in Fig. 9a, the performance and area efficiency will be very poor, as each of the recursive tasks would wait for a result from their child tasks, which in turn would lead to many occupied, but waiting hardware PEs, and will not scale beyond very small values for n. See Sect. 3.6 for a discussion of this problem.

Using the *Reduce-to-parent* scheme of Sect. 3.3.3 in a transitive manner, combined with asynchronous (non-blocking) launches of the child tasks, enables the far more efficient execution sketched in Fig. 9b. Here, each task completes immediately after spawning its child tasks with the updated parameters $n-1$ and $n-2$. Note that a parent task does not wait for the child tasks' results. Instead, by having *all* of these tasks execute in *Return-to-grandparent* mode, the results of all of the child tasks will propagate up to the outer *reduce* tasks, which actually perform the summing over all of the partial results. That computation has been moved out of the inner nodes of the call graph of Fig. 9b, to the outer reduce nodes. In this manner, the recursion depth is not limited by the PEs available on the chip. The implementation can scale from a single Fibonacci PE, and a single reduce PE for summing, up to many PEs running in parallel. The recursion depth is only limited by the size of the BlockRAM storage used for buffering the recursion results in a call-stack-like manner.

When using two PEs for executing Fibonacci computation tasks, and four PEs for the merge/reduce tasks, computing $f(11)$ as a highly task-intensive stress-test requires just 63.13µs, with the bulk of the execution time required for task dispatching/launching (as the computation itself is trivial). When performing host-side scheduling, instead, managing the same parallel structure would require 1.29 ms, more than 20x longer. Note again that we have chosen this example to demonstrate the *scheduling* capabilities and speed of the Cascabel 2 system, it is *not* intended to show high-performance computations of Fibonacci numbers.

4.3 Near-Data Processing for Databases

Our second use-case realizes an accelerator for near-data processing, e.g., for use in computational storage [12]. It will examine the performance of Cascabel 2 for less launch-intensive workloads than the previous Fibonacci example. Here, we assume that a key-value (KV) store is located in persistent memory directly attached to the FPGA, and we perform operations on the FPGA near the data (NDP), instead of transferring the data from persistent memory to the host for processing. Our simplified KV store uses a log-structured block format with data blocks of variable length. On insertion of a new block, it is appended to the last block. To update an existing block, the new version of that block is appended as well, and the invalidation timestamp of the previous (now outdated) version is written. Memory addresses pointing to the newer and older version ease the lookup of different versions. The header format of each block is shown in Fig. 10.

In our use-case, we want to apply a Number Theoretic Transform (NTT) as a batched operation in NDP-fashion on all blocks of a given snapshot. NTT is a special case of the Discrete Fourier Transform applied over finite fields. NTT is the basic function required in many cryptographic applications, such as homomorphic encryption or post-quantum cryptography [13, 14]

Blocks of our KV store belong to the given snapshot, if the timestamp is older than the snapshot and the block is still valid or the invalidation happened at a point later than the snapshot. In our implementation tailored to Cascabel 2, we structure the application as two parts: (1) a database parser, and (2) the actual NTT operation. This way, we have two kernels with low complexity each, but which together can perform a complex operation.

For the batch operation, the database parser iterates over all blocks by fetching all headers sequentially. As block sizes are variable and potentially large, the read accesses to the block headers are bounded by the random access performance of the memory. If a block is *visible* at the timestamp of the snapshot, a new on-chip task launch of the NTT kernel is performed. The NTT hardware module is based on an open-source implementation [15]. We

Figure 10 Block header used in the KV store.

ID	Length	Timestamp (TS)	Invalidation TS	Ptr new	Ptr prev

use the smallest configuration of $(n, K) = (256, 13)$. To integrate this module into the database use-case, a custom load and store unit is implemented for handling all memory transfers. Furthermore, a control unit converts the PE interface of TaPaSCo to the control signals of the NTT module. A single NTT operation requires 1,056 clock cycles by itself. But when including all memory transfers (load and store), the execution time increases to around 2,000 clock cycles (at 300 MHz).

The final design consists of a single database parser and eight instances of the NTT operator. Our sample database has a 50 % visibility of blocks for the chosen snapshot timestamp. When using Cascabel 2, we measure a time of 42.8 ms to process 100,000 blocks. For a comparison, we create a similar setup with the software runtime instead of on-chip launches. The host software fetches the header from the memory on the FPGA and, if the header signals a visible block, it launches the execution on the NTT hardware module. Software execution completes in 120.9 ms, which shows a speed-up of 2.8x when employing Cascabel 2.

4.4 Resource Utilization and Frequency

The dynamic parallelism features of Cascabel 2 require additional chip resources compared to the initial Cascabel version of [5]. As Cascabel 2 is highly configurable for the needs of a specific application, the actual hardware costs depend on the features enabled. However we can describe some design points here: For the two examples, the merge/reduce buffer was configured to use an extra 32 RAMB36 blocks for buffering intermediate child task results. Also, the Cascabel 2 task launch and result interconnects required just 0.27% extra CLBs compared to the original. In all of our experiments, the absolute resource cost of the Cascabel 2 system was below 2% of the available resources, across all resource types on the Alveo U280 board.

To get a better assessment of the required hardware for the on-chip stream connections, we created designs with a variable number of PEs connected to the on-chip streams. Figure 11 shows the utilized resources in terms of CLB LUTs (the Alveo U280 FPGA has a total of 1,303,680 available). Even for the largest design, this is below 1 %. The numbers show that scaling is close to linear and thus should not limit designs with an even larger number of PEs. This highlights the advantage of using streams for communication. The data flows only in a single direction, and is easy to pipeline. For designs larger than those evaluated here, additional pipeline stages can be easily inserted. This, of course, will increase the latency slightly, but may compensate for an otherwise reduced clock frequency.

Another aspect is the number of resources required when adding an *on-chip* connection. As this is typically tightly integrated into a PE, obtaining the precise resource

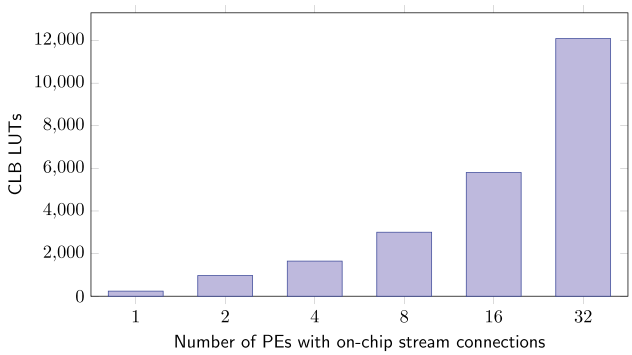


Figure 11 Required hardware resources (CLB LUTs) for the on-chip task launch and result interconnects.

requirements just for the launch logic is difficult. Instead, we analyze a `NOP` PE that just performs task launches bu no other operations. This minimal PE requires 1,274 CLB LUTs, which is just 0.1 % of the total number available. Thus, compared to the much larger PE sizes of real applications, the resource overhead of the launch interface is negligible.

All evaluated designs meet timing closure at the design frequency of 300 MHz. Frequencies of over 500 MHz are achievable [16], but would require increasing the Block-RAM access latency, which we have not done for the examples shown here.

5 Related Work

In contrast to much prior work on HLS that focuses on parallelism *within* a PE [17, 18], our focus is parallelism across PEs and the required infrastructure to support this. Here, the field of related work is much narrower. One recent example is ParallelXL [19], which also aims for dynamic on-chip parallelism: Their PEs are grouped into tiles, which are attached to two NoCs to perform work-stealing scheduling and argument/task routing. The result is a more distributed system, compared to our centralized Cascabel 2 unit. However, the evaluation of ParallelXL was limited to just a small-scale prototype on a Zynq-7000 device and gem5-based simulations. We believe that our simple n-to-1 and 1-to-n streaming interconnects will scale better than ParallelXL's more expensive NoCs, and still allow performance gains even for highly scheduling intensive workloads, as demonstrated by our task-excessive Fibonacci example. In addition, ParallelXL lacks advanced features such as hardware support for merge/reduce operations and has more limited customizability (e.g., omitting the result interconnect on PEs for void child tasks).

Distributing processing elements across multiple network-attached FPGAs has been well described in much prior work.

E.g., the framework presented in [20] provides an infrastructure to generate FPGA clusters from an abstract cluster description, and allows for flexibly replicating kernels. Our work does not provide an automated way to generate full clusters, but by incorporating the TaPaSCo framework, the kernel composition on individual FPGAs can easily be replicated.

6 Conclusion and Future Work

Our Cascabel 2 system provides high-performance hardware-accelerated dynamic parallelism at low resource costs. It extends the scheduling capabilities of prior work (e.g., barriers) with new mechanisms for performing inter-task reduction operations and optimized result passing.

Cascabel 2 enables low-latency task launches both locally on-chip, as well as in a distributed system over the network. With the choice of Ethernet as a communications medium, we achieve both the desired low-latency, as well as scalability by employing switched networks.

With all its proven advantages, for some applications, the task-based programming model currently at the heart of TaPaSCo is not the optimal one. We are currently working to *combine* task-based reconfigurable computing with self-scheduling *streaming* operations for use in data-flow applications.

Future performance improvements can be achieved by further enhancing the underlying scheduling method used in Cascabel 2. In particular, for systems with many *different* PE types, the current "FIFO" scheduling may in many cases not reach optimal PE utilization.

Acknowledgements This research was funded by the German Federal Ministry for Education and Research (BMBF) in project 01 IS 17091 B.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Ernsting, S., & Kuchen, H. (2014). A Scalable Farm Skeleton for Hybrid Parallel and Distributed Programming. *International Journal of Parallel Programming, 42*(6), 968–987. https://doi.org/10.1007/s10766-013-0269-2
2. Heinz, C., Hofmann, J., Korinth, J., Sommer, L., Weber, L., & Koch, A. (2021). The TaPaSCo Open-Source Toolflow. *Journal of Signal Processing Systems*. https://doi.org/10.1007/s11265-021-01640-8
3. Heinz, C., & Koch, A. (2021). Supporting on-chip dynamic parallelism for task-based hardware accelerators. In S. Derrien, F. Hannig, P. C. Diniz, D. Chillet (eds), *Applied reconfigurable computing. Architectures, tools, and applications* (pp. 81–92). Springer: Cham.
4. Wang, T., Geng, T., Li, A., Jin, X., & Herbordt, M. (2020). FPDeep: Scalable Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters. *IEEE Transactions on Computers, 69*(08), 1143–1158. https://doi.org/10.1109/TC.2020.3000118
5. 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
6. Ruiz, M., Sidler, D., Sutter, G., Alonso, G., & López-Buedo, S. (2019). Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 286–292). IEEE. https://doi.org/10.1109/FPL.2019.00053
7. Hartmann, M., Weber, L., Wirth, J., Sommer, L., & Koch, A. (2021). Optimizing a hardware network stack to realize an in-network ML inference application. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*.
8. Aurora 64B/66B Protocol Specification (SP011 (v1.3) October 1, 2014). https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b_protocol_spec_sp011.pdf
9. BlueAXI. https://github.com/esa-tu-darmstadt/BlueAXI
10. Dubucq, T., Forlini, T., Dos Reis, V. L., & Santos, I. (2015). Matrix: Bench - benchmarking the state-of-the-art task execution frameworks of many-task computing.
11. Shah, H., Voloshin, M., & Sharma, D. (2020). MVAPICH2 on Thor: High performance MPI meets mainstream ethernet controller. 9th Annual MVAPICH User Group (MUG) Meeting.
12. Vinçon, T., Weber, L., Bernhardt, A., Riegger, C., Hardock, S., Knoedler, C., Stock, F., Solis-Vasquez, L., Tamimi, S., & Koch, A. (2020). nKV in Action: Accelerating KV-Stores on native computation storage with near-data processing. In *Proceedings of the VLDB Endowment*, Volume 13.
13. Kim, S., Lee, K., Cho, W., Nam, Y., Cheon, J. H., & Rutenbar, R. A. (2020). Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines* (FCCM) (pp. 56–64). IEEE.
14. Scott, M. (2017). A note on the implementation of the number theoretic transform. In *IMA International Conference on Cryptography and Coding* (pp. 247–258). Springer.
15. Mert, A. C., Karabulut, E., Ozturk, E., Savas, E., & Aysu, A. (2020). An Extensive Study of Flexible Design Methods for the Number Theoretic Transform. *IEEE Transactions on Computers*. https://doi.org/10.1109/TC.2020.3017930

16. Xilinx, Inc. Performance and Resource Utilization for AXI4-Stream Interconnect RTL v1.1. URL https://www.xilinx.com/support/documentation/ip_documentation/ru/axis-interconnect.html#virtexuplus
17. 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).
18. Prabhakar, R., Koeplinger, D., Brown, K. J., Lee, H., De Sa, C., Kozyrakis, C., & Olukotun, K. (2016). Generating configurable hardware from parallel patterns. *Acm Sigplan Notices, 51*(4), 651–665.
19. Chen, T., Srinath, S., Batten, C., & Suh, G. E. (2018). An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 55–67). IEEE.
20. Tarafdar, N., Lin, T., Fukuda, E., Bannazadeh, H., Leon-Garcia, A., & Chow, P. (2017). Enabling flexible network FPGA clusters in a heterogeneous cloud data center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Association for Computing Machinery. FPGA '17* (pp. 237–246). New York, NY: USA. https://doi.org/10.1145/3020078.3021742