Supporting On-Chip Dynamic Parallelism for Task-based Hardware Accelerators

Carsten Heinz[0000−0001−5927−4426] and Andreas Koch[0000−0002−1164−3082]

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

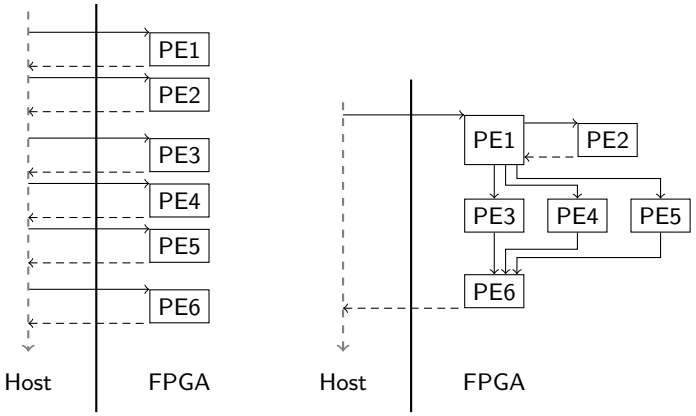
Abstract. The open-source hardware / software framework TaPaSCo aims to make reconfigurable computing on FPGAs more accessible to non-experts. To this end, it provides an easily usable task-based programming abstraction, and combines this with powerful tool support to automatically implement the individual 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.
Our solution is able to handle recursive task structures and is shown to have 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 [4], often results in increased communication overhead between the host and the FPGA.

Our work addresses these challenges by adding fine-grained *on-chip* task scheduling to the TaPaSCo framework for reconfigurable computing [6]. This new feature enables low-latency interactions directly between processing elements, without the need for host involvement. It significantly reduces the number



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

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

of host/accelerator interactions, as shown in Figure 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.

2 Heterogeneous Computing Architecture

The open-source TaPaSCo framework [6] 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 chan-

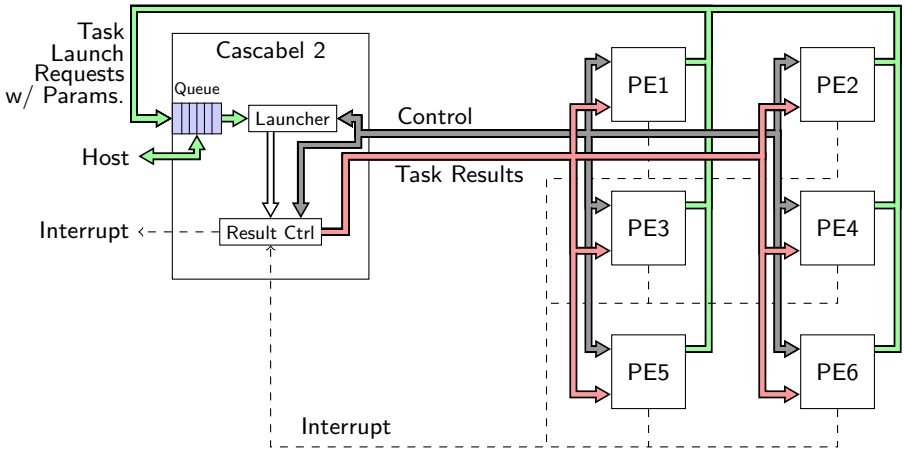


Fig. 2. Cascabel 2 hardware dispatcher/launcher and its AXI Stream connections to the PEs for accepting task requests and distributing task results.

nels 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. 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.

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 Figure 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 512 bit stream, shown in green, flows from the PEs to the Cascabel 2 unit, and carries new launch requests, including child task parameters. A second 64 bit 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.

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 Figure 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).

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).

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 Figure 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.

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.

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,

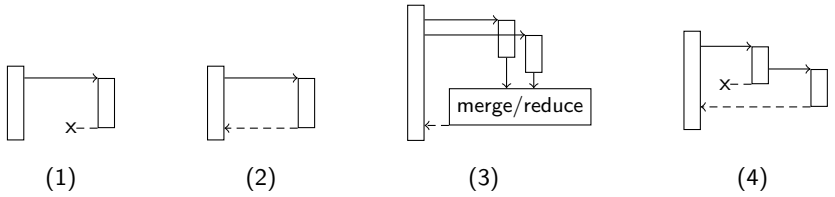


Fig. 3. Return value handling: (1) Discard, (2) return-to-parent, (3) merge/reduce-to-parent, (4) return-to-grandparent. **x** indicates a discarded result.

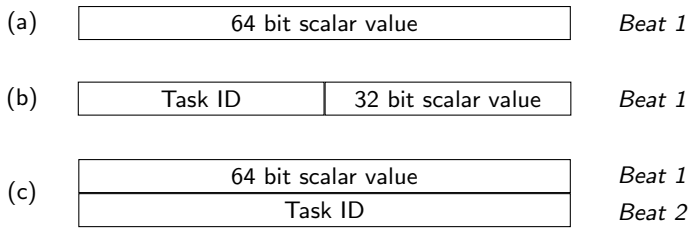


Fig. 4. Stream data formats for child task return values

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 Section 4.2.

The logic for realizing the different return actions is implemented in the *Return Ctrl* block (shown in Figure 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.

3.4 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 64 bit. 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 512 bit-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 300 MHz design clock and are synthesized in Vivado 2020.1.

4.1 Latency

The key property when performing on-chip task 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 [3]. 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 with using the host-based software-only scheduler, it takes 7.41 µs. Using the hardware-assisted software scheduler [5], which is optimized for task throughput instead of task latency, requires 8.96 µs. 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. Figure 5 summarizes the measured latencies. 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.

4.2 Recursion

To stress-test the advanced task management capabilities described in Section 3.3 on a simple example, we show a *recursion-intensive* approach of computing the Fibonacci sequence, which is defined as

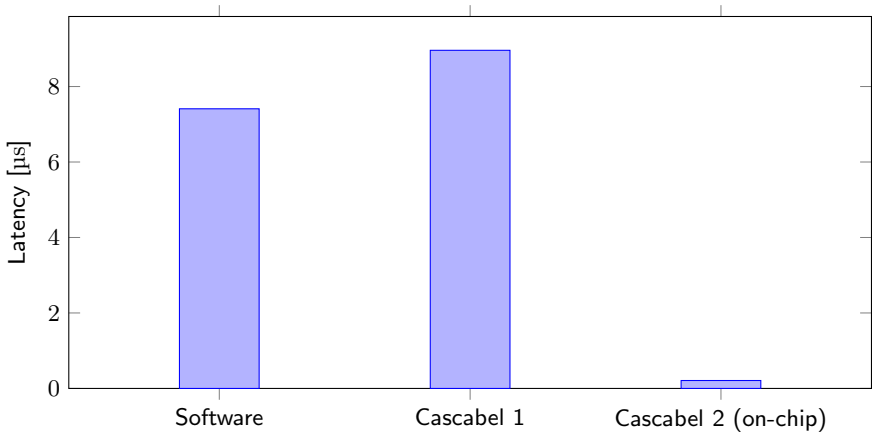


Fig. 5. Comparison of launch latencies: host-side software-only, throughput-oriented hardware-assisted software with Cascabel 1, latency-oriented hardware-only with Cascabel 2

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

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

When implementing this computation naively without the merge/reduce support of Cascabel 2, as shown in Figure 6.a, 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 Section 3.4 for a discussion of this problem.

Using the *Reduce-to-parent* scheme of Section 3.3 in a transitive manner, combined with asynchronous (non-blocking) launches of the child tasks, enables the far more efficient execution sketched in Figure 6.b. 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 Figure 6.b, 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

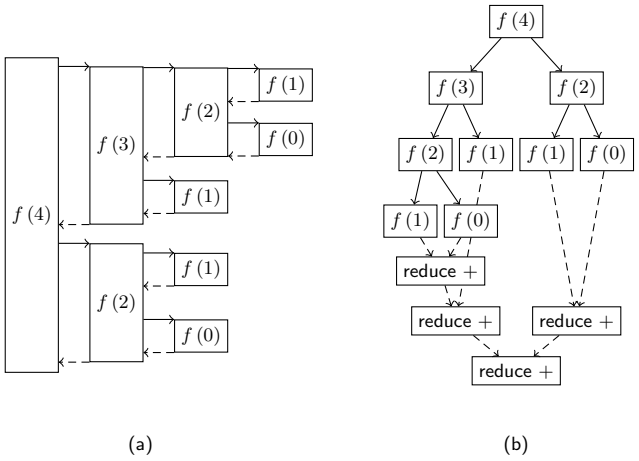


Fig. 6. Recursive Fibonacci computation: (a) naive synchronous execution, (b) asynchronous mode with transitive return-to-grandparent value passing and merge/reduce

stress-test requires just 63.13 µs, 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 [9]. It will examine the performance of Cascabel 2 for less launch-intensive workloads than the previous Fibonacci example. Here, we assume that a database is stored in persistent memory directly attached to the FPGA, and we process aggregation queries on the FPGA near the data (NDP), instead of transferring the data from persistent memory to the host for processing. The sample query we use for this example could be expressed as

```
SELECT avg(age), max(salary), sum(hours)
FROM employees;
```

The database is stored in a format with fixed record size, which allows the use of simple strided offset-based accesses to retrieve the required columns in subsequent rows. On the processing side, we have PEs for the different aggregation tasks (avg, max, sum, etc.) available on the FPGA.

Both the software-scheduled and Cascabel 2 implementations will use these hardware PEs as NDP operations, which compute their results within a *single*

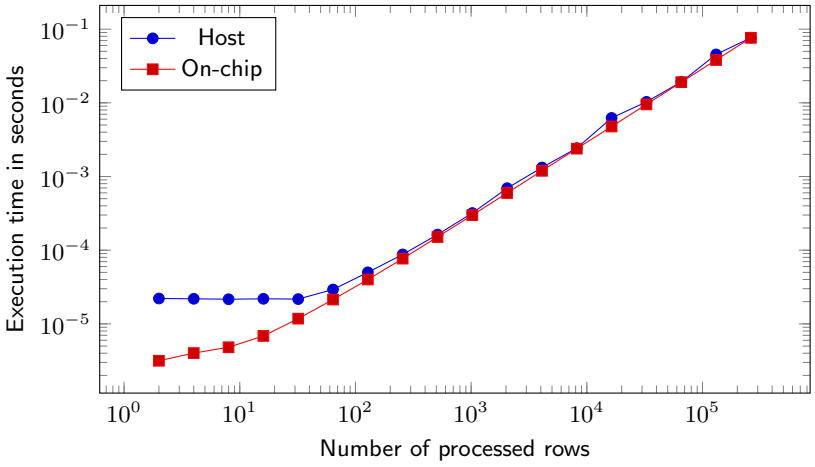


Fig. 7. NDP query runtimes for increasing record numbers

task each over an increasing number of records. For each of the aggregations, a separate task will be launched (three in total, for avg/max/sum), either by the TaPaSCo software scheduler, or using Cascabel 2 on-chip. The results are shown in Figure 7. Obviously, with a larger number of records being processed in each task, the launch overhead per query becomes smaller. However, using Cascabel 2, hardware-accelerated processing remains profitable even for *smaller* numbers of records (e.g., after a narrow selection by a tight WHERE clause). It is faster than software scheduling for fewer than 512 records.

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.

All evaluated designs meet timing closure at the design frequency of 300 MHz. Frequencies of over 500 MHz are achievable [10], but would require increasing the BlockRAM 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 [1] [8], 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 [2], 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).

6 Conclusion and Future Work

Our Cascabel 2 system provides high-performance on-chip 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.

With its short launch latencies, Cascabel 2 could also be employed for hardware-accelerated network processing close to the interface, e.g., as performed in [7] for network security.

Despite 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.

Another interesting addition could be the support for communication *across* single FPGA boundaries to scale to larger applications. Specifically, the currently on-chip-only launch/parameter/result interconnects could be extended using direct FPGA-FPGA links in a single server or rack, or even over switchable/routable network protocols in an entire datacenter setting. While such connections carry an additional latency penalty of $\approx 1\,\mu s$, inter-device launch latency would still remain shorter than in the host-based software-only solution initially used for TaPaSCo.

Acknowledgment

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

References

1. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J.H., Brown, S., Czajkowski, T.: 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 (2011)
2. Chen, T., Srinath, S., Batten, C., Suh, G.E.: 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 (2018)
3. Dubucq, T., Forlini, T., Dos Reis, V.L., Santos, I.: Matrix: Bench - benchmarking the state-of-the-art task execution frameworks of many-task computing (2015)
4. Ernsting, S., Kuchen, H.: A scalable farm skeleton for hybrid parallel and distributed programming. Int. J. Parallel Program. **42**(6), 968–987 (Dec 2014), https://doi.org/10.1007/s10766-013-0269-2
5. Heinz, C., Hofmann, J.A., Sommer, L., Koch, A.: 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 (2020). https://doi.org/10.1109/ROSS51935.2020.00008
6. Heinz, C., Hofmann, J., Korinth, J., Sommer, L., Weber, L., Koch, A.: The tapasco open-source toolflow. Journal of Signal Processing Systems (May 2021), https://doi.org/10.1007/s11265-021-01640-8
7. Mühlbach, S., Brunner, M., Roblee, C., Koch, A.: MalCoBox: Designing a 10 Gb/s Malware Collection Honeypot Using Reconfigurable Technology. In: 2010 International Conference on Field Programmable Logic and Applications. pp. 592–595 (2010). https://doi.org/10.1109/FPL.2010.116
8. Prabhakar, R., Koeplinger, D., Brown, K.J., Lee, H., De Sa, C., Kozyrakis, C., Olukotun, K.: Generating configurable hardware from parallel patterns. Acm Sigplan Notices **51**(4), 651–665 (2016)
9. Vinçon, T., Weber, L., Bernhardt, A., Riegger, C., Hardock, S., Knoedler, C., Stock, F., Solis-Vasquez, L., Tamimi, S., Koch, A.: nKV in action: Accelerating KV-stores on native computation storage with near-data processing. In: Proceedings of the VLDB Endowment, Volume 13 (2020)
10. Xilinx, Inc.: Performance and resource utilization for axi4-stream interconnect rtl v1.1, https://www.xilinx.com/support/documentation/ip_documentation/ru/axis-interconnect.html#virtexuplus