FLexiTASK: A Flexible FPGA Overlay for Efficient Multitasking

Joel Mandebi Mbongue
University of Arkansas
Fayetteville, Arkansas
jmmandeb@uark.edu

Danielle Tchuinkou Kwadjo
University of Arkansas
Fayetteville, Arkansas
dtchuink@uark.edu

Christophe Bobda
University of Arkansas
Fayetteville, Arkansas
cbobda@uark.edu

ABSTRACT

One of the major obstacles to the adoption of FPGAs in high-performance computing is their programmability. It requires hardware design skills and long compilation times. Overlays have been proposed as a way to abstract FPGA resources. Unfortunately, most of the time, the topologies they use to connect computing cores impose restrictions on where tasks are placed and how they communicate. In this paper, we propose an overlay architecture designed for efficiency and flexibility. It features a novel Network-on-Chip (NoC) infrastructure making flexible, with no limitation, the placement of hardware tasks. The presented architecture allows tasks to communicate with a low latency and eases the reconfiguration of desired areas on the fabric at runtime. After prototyping the proposed architecture on an Altera Cyclone V FPGA, a maximum frequency of *282 MHz* has been reached and a speedup ranging from 4× to 195× has been observed in some applications compared to the native execution.

KEYWORDS

FPGA; Overlay; Multitasking

ACM Reference Format:
Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, and Christophe Bobda. 2018. FLexiTASK: A Flexible FPGA Overlay for Efficient Multitasking. In *GLSVLSI '18: 2018 Great Lakes Symposium on VLSI, May 23–25, 2018, Chicago, IL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194554.3194644

1 INTRODUCTION

The insatiable demand for innovative, faster, and efficient technologies has led from high-clocked single processors to the use of multicores. In multicores with customized components, computing resources can be tailored to applications' needs, resulting in better system throughput and power consumption compared to homogeneous multicores. Devices like FPGAs can play a big role in such platforms because of their reconfigurable nature and low power consumption compared to general purpose cores [6]. For such an integration to be viable, it must be possible to place, execute and replace tasks on the FPGA fabric at runtime. However, using FPGAs at the raw level raises some hurdles. As a prelude to accelerating a task, a corresponding circuit must be synthesized to specify the functionality on the FPGA. Unfortunately, the process of synthesizing hardware designs from hardware description languages or high-level synthesis is time consuming. Secondly, for upcoming tasks, new circuits will have to be synthesized. These problems make this approach unviable. On the other hand, an abstraction layer offering a hardware virtualization of the FPGA resources could help to tackle those obstacles. This will remove the need for synthesizing hardware circuits and reduce the configuration overhead. In addition, it will provide more flexibility to dynamically place and replace threads on specific regions of the FPGA when needed [3]. The problem addressed in this work is hardware multitasking on coarse-grained reconfigurable architectures (CGRAs). The improvement and development of CGRAs arise from the huge variety of research and industry needs for flexible high performance computing. Our main contribution is to propose a novel virtualization architecture which makes flexible and unrestricted the placement, and communication of hardware tasks at runtime.

Recent works in CGRAs and FPGA overlays mostly present dataflow machine architectures [4, 5, 7]. They are usually added into systems as co-processors for accelerating tasks. They have the advantage to exhibit FPGA's flexibility and break the FPGA-ASIC performance gap. Communications among functional units embedded in these architectures are usually based on direct interconnections among neighbor or distant units. Because of that, those architectures do not address the problem of accessibility and flexibility of communication between computational resources. Bus-based communications (see Figure 1a) present a potential alternative but the overhead of hundreds or thousands of processing elements (PE) trying to communicate using a single bus will drastically reduce system performances, and ultimately obfuscate any parallelism and multitasking gain. A Network-on-Chip architecture (NoC) [1] implemented as an abstraction layer to connect all the cores can be a solution to this communication bottleneck. NoCs use messages for communications between cores, memories and peripherals, all of them connected to the network infrastructure instead of using dedicated wires. Their advantages have been deeply studied and discussed in the literature [9]. Despite their great assets, NoCs present some drawbacks. NoCs are usually provided as a 2-dimensional mesh with PEs or routers at the intersection between lines and columns. There are few problems nevertheless raised:

1) Architectures with PEs at the intersection of lines and columns like in Figure 1b only provide communication links between neighbor PEs [4, 5, 7]. Though they allow a high throughput between adjacent PEs, they impose a restricted task placement and fixed communication exchanges. They are not best optimized for multitasking in which communication between distant tasks is needed.
2) In architectures with routers at the intersection of lines and columns (see Figure 1c), each PE is attached to a router [2]. This makes unrestricted the communication between distant cores, but

GLSVLSI '18, May 23–25, 2018, Chicago, IL, USA

ACM ISBN 978-1-4503-5724-1/18/05…$15.00
https://doi.org/10.1145/3194554.3194644

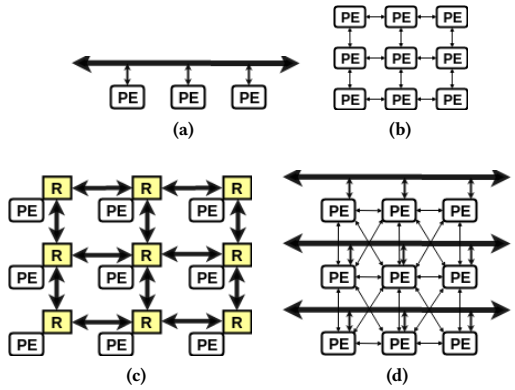

Figure 1: Bus-based Interconnection (1a). PEs at the Intersection of Lines and Columns of the 2-D Mesh (1b). Routers at the Intersection of Lines and Columns (1c), each PE attached to one Router. The Zippy architecture (1d) uses buses for distant communication between PEs belonging to the same row.

introduce a high communication overhead because the number of routers to get packets to their destinations is usually more than what is really needed: more hops between the source and destination of a packet imposes more communication latencies and message blocking probability.

In order to make flexible communications between distant PEs while maintaining direct links between neighbor ones, [8] proposes the Zippy architecture (see Figure 1d). It does not use routers, rather, it features PEs at the intersection of lines and columns of the grid. It employs direct interconnections between adjacent PEs and uses a bus to connect PEs belonging to the same row. The issue with this architecture is the overhead generated by the bus management if the number of PEs is drastically increased. In addition to that, transfers between distant PEs is limited to the same lines.

In this work, we introduce a novel architecture that uses routers for communications between distant cores and direct links between adjacent PEs. It differs from other architectures relying on routers as it decreases the number of routers by increasing the number of PEs attached to each of them.

2 PROPOSED ARCHITECTURE

2.1 General Organization

We propose the general organization of Figure 2. It is a Torus interconnect consisting of a two-dimensional array of routers (R in Figure 2) and programmable processing elements (PE in Figure 2). PEs communicate through a flexible and scalable NoC architecture providing a high path diversity and minimal routes. They are gathered in groups around routers to form a *subnetwork*. The overlay uses data packets for communications. This is beneficial because it makes the overlay scalable, and resources can directly be reused when they complete message transmissions. The Torus topology is implemented to shorten distances between PEs and increase the connectivity: routers at the left edge of the first and last lines are connected to those at the opposite edge, PEs which are all over the edges of the overlay are connected to PEs at their opposite edges.

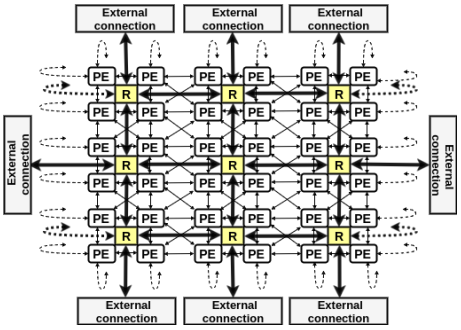

Figure 2: Overlay Architecture

To make it possible to send packets simultaneously to different subnetworks, each router at the edge of the overlay has an external connection. Figure 2 presents an instance of the overlay using 3×3 routers, each surrounded by four PEs. The number of PEs attached to each router, and how many routers to use can be modified according to the needs and available resources on the chip. To be operational, PEs are configured to implement specific operations and send results of their computations to specific targets. There are two configuration modes:**(1) The online mode:** in which configuration packets containing operands are sent. In this case, the next use of the PE will require a new configuration packet. **(2) The offline mode:** configuration packets are sent first, and operands will arrive later from neighbor PEs or through the router of the subnetwork. This mode is interesting for vector processing since configurations can be kept as long as needed.

2.2 Routing and Processing Elements

Routers (see Figure 3a) control paths taken by data to make sure that each packet reaches its destination. Each router features four channels for communicating with adjacent routers and direct links with PEs. The *scheduler* uses the *First Come First Serve* scheduling algorithm, with the highest priority given to communications coming from PEs, to brings packets to the crossbar matrix. FIFOs store packets waiting to be scheduled. Communications from PEs are first considered to have them available for new computations as soon as possible. The *crossbar matrix* transfers packets to the next direction. A PE is directly connected to at most eight other adjacent PEs. The Broadcast unit (see Figure 3b) forwards channels activated by the configuration to input multiplexers. Input Multiplexers select the origin of operands sent to the Functional Unit (FU), while the Demultiplexer chooses where to send results of computations . The Output Register always contains a copy of the latest result computed in the FU. The Configuration Register stores configuration packets received by the PE. This allows to fix the source of operands, the operation to be performed and the destination where to send results. The FU runs computations on one or two 32 bits operands.

2.3 Programming Flow

Overlays have the advantage that application kernels can be mapped with a programmability similar to software executed into processors. The design, implementation and synthesis of the overlay itself is done off-line using vendor tools. Our custom mapping tool is a

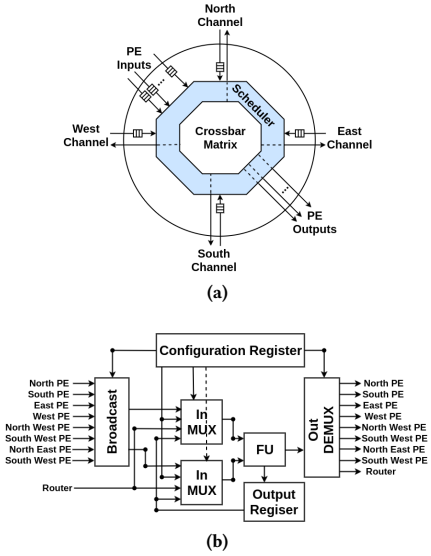

Figure 3: Router Architecture (Figure 3a). Processing Element (Figure 3b).

C/C++ library providing high level functions to access the overlay resources. In its current version, kernels and where computations will take place on the overlay are specified manually. This because the attention of the paper is on the hardware architecture. Future works will focus on having automatic placements. In order to use some hardware acceleration from the overlay, a C/C++ application just needs to call functions from our library. The code can then be compiled using a regular C/C++ compiler like the GNU Compiler Collection (GCC).

3 EVALUATION

Evaluation Infrastructure: the Terasic DE1-SoC board with an Altera Cyclone V FPGA was chosen to verify the operation of the prototyped architecture. To edit and synthesize the overlay, we used Quartus Prime Standard Edition 16.0. In addition, ModelSim-Intel FPGA Edition allowed to run RTL simulations and observe latencies. Throughout our experimental evaluations, we used an instance of the overlay featuring 5×5 routers, each router surrounded by 4 PEs. **Evaluation Metrics:** we look at speedups compared to the native execution. For that purpose, a sytem-on-chip is designed to evaluate execution times on the Dual-Core ARM Cortex A9 Hard Processor burned into the FPGA, as opposed to the same Hard Processor coupled with our overlay. We also consider the f_{MAX} observed at the level of routers under various conditions. Given that the number of PEs around routers is scalable, we trace routing latencies and also study the impact of increasing the number of PEs on FPGA resources. Finally, we discuss some issues arising when comparing our architecture to existing works. **Evaluation Benchmarks:** we tested three image processing operations on 800×600 input images. Afterwards, we ran some matrix/vector operations with matrices of size 100×100, and vectors of 100 entries. For each application, we manually configured a parallelized placement of operations onto the overlay and a sequential one. The parallelized placement

Table 1: Execution Times

Applications	Cortex A9	Cortex A9 + Overlay (Sequential Mapping)	Cortex A9 + Overlay (Parallel Mapping)
Roberts Cross	763.69 ms	67 ms	5.15 ms
Corner Detection	488.70 ms	161.77 ms	53.92 ms
Smoothing	116.28 ms	104.98 ms	8.075 ms
Matrix Multiplication	48.77 ms	42 ms	11.4 ms
Outer Product	0.39 ms	0.05 ms	0.002 ms
Dot Product	0.008 ms	0.008 ms	0.001 ms

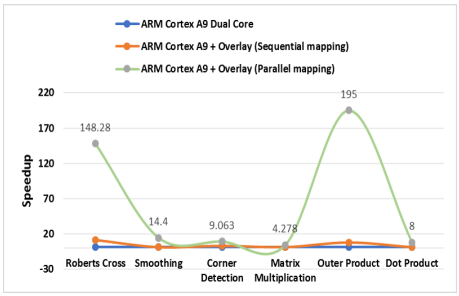

Figure 4: Execution Speedups

provides an idea on how our overlay performs on multithreaded applications, while the sequential placement demonstrates that our architecture also fits pipelined executions. For evaluation purposes, we examine how much acceleration is achieved on multithreaded and sequential environments. **Observations:** table 1 summarizes execution times observed when running all test applications on our three testing infrastructures. It demonstrates that the overlay excels when tasks are executed in parallel. Figure 4 depicts speed up statistics: the parallel placement of the *Outer product* executed 195× faster than the native execution, and 25× faster compared to the sequential execution on the overlay. While the overlay best performs on multithreaded applications, sequential placements still outperform native executions. The *Matrix multiplication* in the parallel placement on the overlay only executes about 5× faster. This is due to the dependencies between operations, and to the size of the overlay. Using 100 PEs to run computation on matrices of size 100x100 restricts the number of tasks that could be run in parallel. Figure 5 studies the resource usage incurred by the scaling of PEs around routers. Everytime we double the number of PEs in a subnetwork, the new circuit synthesized employs about 12 more ALMs and 4 more registers on the FPGA. It demonstrates that PEs are not resource intensive. This observation is particularly interesting as it could pave the way to more PEs for more speed up. However, if increasing the number of PEs provides more computing power, it also influences routing latencies. Figure 6 studies average wait-

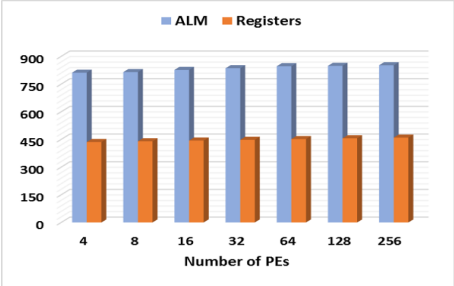

Figure 5: Resource Usage for PE Scaling

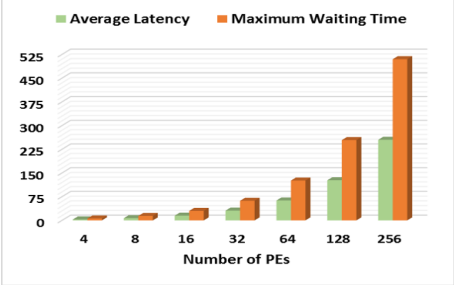

Figure 6: Routing Latencies for PE Scaling

ing times and maximum waiting times for packets simultaneously reaching a router, targeting each PE in the subnetwork. Having 4 PEs, the average routing latency would be 3 clocks cycles and the last packet will wait at most 6 clock cyles. By increasing the number of PEs to 16, the average waiting time could easily increase up to 15 clock cycles and the last packet transferred by the router could wait up to 30 clock cycles. Using more than 32 PEs around a router would become unviable as it will cause an average waiting time of at least 31 clock cycles. The average latency is approximately equal to number of PEs while the maximum waiting time is almost the double. These results indicate that: 1) Though using more PEs looks attractive and not resource hungry, keeping a relative small amount of PEs could drastically decrease the communication overhead, especially when routers are to be crossed. And 2) to best benefit from the architecture, the placement of hardware tasks should avoid saturating routers with communication requests. The f_{MAX} of the overlay is derived from the Quartus Prime Timing Analysis tool. For instance, we observed a maximum frequency of $282MHz$ for 4 PEs around a router and $270MHz$ when having 256 PEs.

In [4], authors propose an architecture for speeding up data flow graph executions using an Altera Stratix III platform. They achieved a maximum frequency of $172MHz$ and a speed up of 9× compared to an execution on a NIOS II soft processor. Sen Ma and al [7] designed an overlay for accelerating custom instructions built on a Xilinx Kintex-7 FPGA for testing purposes. They claimed a speedup of 25× on a 3x3 matrix multiplication. [5] proposes an architecture for pipelined execution of data flow graphs. After prototyping their overlay on an Altera Stratix IV FPGA, they reported a f_{MAX} reaching up to $355MHz$ and throughputs obtained in GFLOPS. It could be tempting to claim that our architecture outperforms results presented in [4] as we obtain a greater speedup and higher maximum frequency. Following the same logic, it might also appear like the architecture described in [5] is more efficient than ours since authors claim to achieve a better maximum frequency. We could even speculate which of our architecture or the one portrayed in [7] offers a better speedup. But this form of comparison is misleading since it lacks context. For instance, [4], [7], and [5] do not use similar testing applications (i.e. lack shared benchmarks). Even when they try to, the size of the dataset is generally different and sometimes not even disclosed. This is on top of the fact that evaluation platforms generally rely on FPGAs from different manufacturers with completely different hardware structures. Finally, even when FPGAs from the same brand are used, using various series produce different results. To illustrate that point, when testing our architecture on an Altera Stratix IV FPGA, we observe a maximum frequency rising up to $359MHz$ compared to the $282MHz$ on the Cyclone V. This new frequency observed even surpasses the one pointed out in [5]. Those three factors make it difficult to fairly provide performance comparisons between architectures as it is particularly challenging to reproduce the exact same testing environment.

4 CONCLUSION

We presented a novel virtualization architecture for FPGAs which is suitable for multithreaded applications but also for sequential execution. The architecture allows to modify the functionality of any region of the layout at runtime. Our evaluation showed that adding our overlay as a co-processor allowed significantly decreased execution time compared to the native execution. Future researches will focus on automatically identifying kernels from application specified with high-level programming languages.

ACKNOWLEDGMENTS

This work was supported in part by the NSF under Grant CNS-1618606 & 1302596.

REFERENCES

[1] Luca Benini and Giovanni De Micheli. 2002. Network on chips: A new soc paradigm. *computer* 35, 1 (Aug. 2002), 70–78. https://doi.org/10.1109/2.976921
[2] Tobias Bjerregaard and Shankar Mahadevan. 2006. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)* 38, 1 (June 2006), 1. https://doi.org/10.1145/1132952.1132953
[3] Christophe Bobda. 2007. *Introduction to reconfigurable computing architectures, algorithms and applications.* Springer Science & Business Media.
[4] Davor Capalija and T Abdelrahman. 2012. A coarse-grain fpga overlay for executing data flow graphs. *The Second Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2012)* (2012).
[5] Davor Capalija and Tarek S Abdelrahman. 2013. A high-performance overlay architecture for pipelined execution of data flow graphs. *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on* (2013), 1–8. https://doi.org/10.1109/FPL.2013.6645515
[6] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/2145694.2145704
[7] Sen Ma, Zeyad Aklah, and David Andrews. 2015. A run time interpretation approach for creating custom accelerators. *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on* (2015), 1–4. https://doi.org/10.1109/FPL.2015.7293996
[8] Christian Plessl and Marco Platzner. 2011. Hardware virtualization on dynamically reconfigurable processors. (2011).
[9] Muhammad Athar Javed Sethi, Fawnizu Azmadi Hussin, and Nor Hisham Hamid. 2015. Survey of network on chip architectures. *Sci. Int.(Lahore)* 5, 27 (2015), 4133–4144.