Software-like Compilation for Data Center FPGA Accelerators

James Thomas
jjthomas@cs.stanford.edu
Stanford University, CA, USA

Chris Lavin
chris.lavin@xilinx.com
Xilinx CTO, Longmont, CO, USA

Alireza Kaviani
alireza.kaviani@xilinx.com
Xilinx CTO, San Jose, CA, USA

ABSTRACT

Compilation times for large Xilinx devices, such as the Amazon F1 instance, are on the order of several hours. However, today's data center designs often have many identical processing units (PUs), meaning that conventional design flows waste time placing and routing the same problem many times. Furthermore, the connectivity infrastructure of a design tends to be finalized before the PUs, resulting in unnecessary recompilation of a large fraction of the design.

We present an open source flow where the connectivity infrastructure logic is implemented ahead of time and routed to many interface blocks that border available slots for PUs. As architects iterate on their PU designs, they only need to perform a single set of parallel, independent compile runs to implement and route the PU alongside each distinct interface block. Our RapidWright-based system stitches the implemented PU into the available slots in the connectivity logic, requiring no additional routing to finalize the design. Our system is able to generate working designs for Amazon F1, and reduces compilation time over the standard monolithic compilation flow by an order of magnitude for designs with up to 180 PUs. Our experiments also show that there is future potential for an additional 4X runtime improvement when relying on emerging open source place and route tools.

ACM Reference Format:
James Thomas, Chris Lavin, and Alireza Kaviani. 2021. Software-like Compilation for Data Center FPGA Accelerators. In *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART '21), June 21–23, 2021, Online, Germany.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3468044.3468047

1 INTRODUCTION

FPGAs are making their way into data centers as network and compute accelerators to improve their overall power efficiency [2][4]. This growth has increased demand for more productive FPGA design development tools that create a more software-like experience similar to that of traditional compute engines such as CPUs. Innovation on the front-end of the hardware development flow has helped raise the abstraction of design entry to software languages. However, the back-end implementation tools have largely been limited to FPGA vendor compilers with long compilation times.

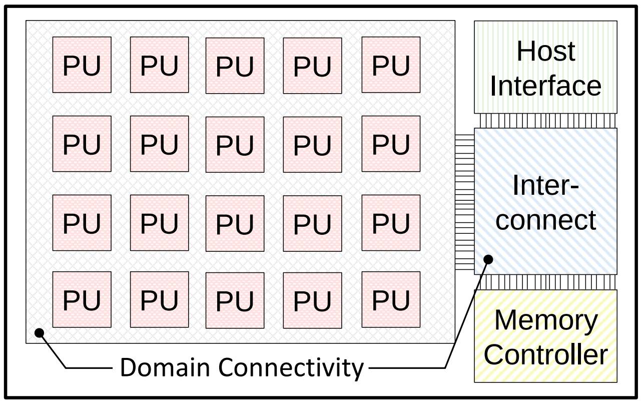

Figure 1: High-level pattern for domain-specific accelerators

This continues to stymie designers looking to prototype their designs on real hardware or software developers with traditional CPU experience.

In order to approach software-like compilation times, back-end FPGA implementation will need a different strategy than what is currently employed in conventional FPGA flows. One of the significant challenges in the new domain-specific age of computer architecture [6] is that FPGA tooling development has largely targeted broad support of customer design scenarios. As FPGA implementation tooling requires heavy investment, vendors—out of necessity—must optimize for generalized solutions that maximize availability across a wide gamut of customer design scenarios. However, as emerging applications such as data center workloads become more prevalent, it becomes clear that a customized approach for back-end implementation is needed to achieve the highest efficiency and performance.

There are two specific attributes of data center FPGA applications that make them amenable to compilation acceleration: 1) replication and 2) reuse. First, designs typically consist of many identical copies of a single processing unit (PU) connected through connectivity infrastructure that handles communication among PUs and retrieval of data from external memory. Conventional FPGA compilation is a flat process; synthesis, placement and routing are performed separately for these identical PUs, increasing compile time. The second commonality for designs is that the connectivity infrastructure does not change when designers are iterating and often remains a constant for a given domain of applications. If the connectivity logic can be reused from a similar application in the same domain, we can avoid recompilation of this static shell. Figure 1 depicts these two high-level design patterns for domain-specific accelerators.

In this paper we present a flow that aims to reuse back-end compilation work as much as possible. We identify a domain with a compute model similar to the one presented in [12] and conduct an experimental evaluation using a group of designs within that domain. We pre-implement the corresponding domain connectivity grid and replicate the PUs using an open source back-end customization tool, called RapidWright [8]. Each PU implementation is replicated anywhere from 10 to 30 times, resulting in a final design with close to 200 PUs. RapidWright enables PU replication throughout the Fleet connectivity grid in roughly a minute. The PU implementations can be compiled in parallel within about 10 minutes. This is in contrast to a conventional flow where each design change can take hours to complete as all PU instances and connectivity grid are recompiled in their entirety. We target the Amazon F1 device to demonstrate working designs that run in the cloud and will make our flow and experiments available in open-source. Specific contributions of this work include:

(1) An open source flow leveraging an existing FPGA data center platform to compile and run spatial compute accelerators.
(2) Experimental validation and implementation of the proposed flow for an example compute domain, called Fleet, showing an order of magnitude compile time improvement.

The next section summarizes relevant background. Section III describes the details of the customized back-end flow, followed by section IV that validates the flow showing experimental results. Sections V and VI summarize related literature, conclude the paper and point to future work.

2 BACKGROUND

2.1 Front-end Compiler: Fleet

Fleet [12] is a framework that allows designers to specify a single stream processing unit (PU) in a streaming-oriented HDL. The Fleet front-end compiler generates a hardware design with many copies of the PU connected to a connectivity grid that feeds each PU with its own stream of data from external memory. Fleet designs running on Amazon F1 are able to outperform top-of-the-line CPUs and GPUs in performance per watt for several streaming applications, including JSON parsing and integer compression. Fleet-style applications represent a computing domain that lends itself to the generic paradigm explained in the introduction. The PU is replicated hundreds of times, presenting a large opportunity for implementation reuse, and the connectivity infrastructure is identical for all Fleet applications with the same number of PUs. Figure 2 summarizes the connectivity and data movement infrastructure for the Fleet computing domain. Emerging applications such as machine learning or image processing often have a similar conceptual architecture with many identical computational kernels and a domain-specific connectivity infrastructure that feeds those kernels from external memory.

2.2 Back-end Implementation: RapidWright

RapidWright [8] is an open source framework that enables customization of FPGA back-end implementation. It provides a bridge into Xilinx FPGA implementation tools (Vivado) and provides device models that enable customized place and route algorithms to be built. By providing an open framework, it endows developers with the flexibility needed to enable customized solutions that can be tuned for unique domain requirements such as compile time, performance and/or timing predictability.

In addition to enabling customized tooling solutions for commercial FPGAs (such as the Amazon F1 instance), RapidWright also provides unique implementation capabilities such as composability, replication and relocation of previously placed and routed logic. In order to accelerate FPGA compilation, reuse of previously compiled results is a key strategy of this work.

3 CUSTOMIZED BACKEND FLOW

The back-end flow comprises three main steps: building a connectivity shell (off-line), creating a template for fast "on-line" PU compilation, and combining PUs with the pre-implemented shell to build the complete design. In this section we summarize these three steps.

3.1 Domain-specific Connectivity Shell

A connectivity shell is the pre-implemented, domain-specific infrastructure designed such that PUs can be inserted into empty slots later on without the need for routing. The RTL for the shell must have ports for external IO (often an AXI bus) to external memory and a set of internal ports for communication to each PU (PU IO interface). The designer must specify locations and types of PU slots. A PU type that is to be replicated into multiple slots is called

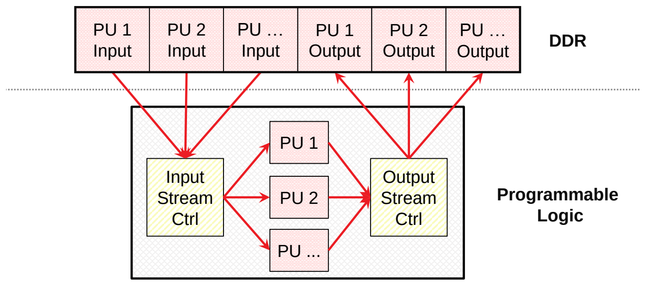

Figure 2: Fleet stream-oriented processing model

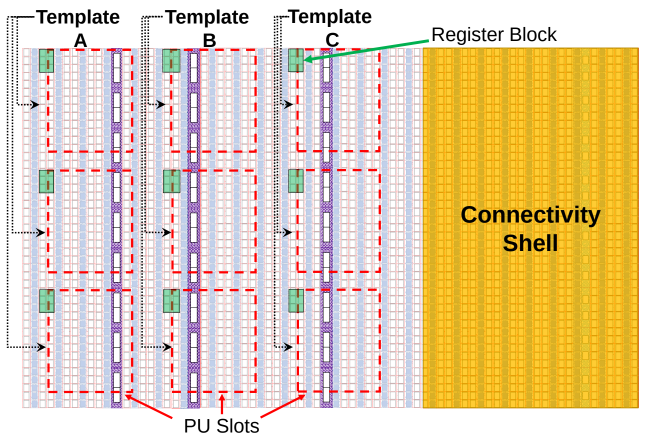

Figure 3: Physical view of PU slots next to the connectivity shell in the fabric. The BRAM column (purple) is located in a different relative position in each slot column, requiring a different implementation template for each column.

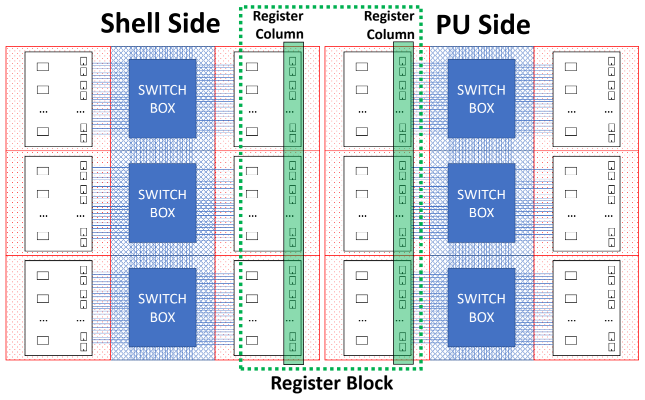

Figure 4: The register block for a particular PU template.

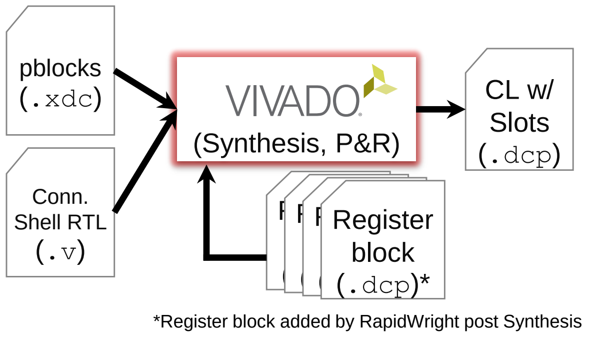

Figure 5: Offline connectivity shell implementation flow

an "implementation template." The PU slots are described using rectangular Xilinx placement constraints ("pblocks"), which constrain logic, DSP, and BRAM resources. Figure 3 shows a physical view of the shell with multiple PU templates. The overall goal is to pre-implement shell routing such that the online compilation runtime for PUs is minimal. To accomplish this, register blocks are inserted across the border of PU slots in the connectivity shell to provide a common physical routing interface. The register blocks consist of two columns of horizontally connected registers, with one connected register pair per bit in the PU IO interface. Figure 4 shows a register block, where the connectivity shell routes to the left column of the register block, and the PU routes to the right column. The same routed register block is used for every PU slot that will be filled by the PU implementation template. Inserting and pre-routing registers at the communication border of a PU and the shell allows PU replication without any rerouting or placement. These register blocks at the PU IO interface completely isolate the timing of the connectivity shell and PUs, enabling higher performance for both components. Of course, this requires the PU IO interface to be latency-insensitive, which is generally a reasonable requirement in throughput-oriented datacenter applications. The pblock used for the connectivity shell consists of the entire available FPGA fabric, excluding the PU slots. Figure 5 summarizes our flow using both Vivado and RapidWright to build the connectivity shell. This flow is performed offline, once per domain. It is possible to create multiple connectivity shells for various size of templates and/or based on external memory needs.

3.2 Processing Units (PUs)

A PU implementation must be compiled for each distinct PU template. Unlike the implementation of the connectivity shell, this step is performed "online" after each design change, and we want to ensure that the compilation is fast. For each implementation template, the PU is connected to the right column of the corresponding register block as shown in Figure 4. The area of a PU template is constrained to a compatible pblock used in shell creation and includes the right column of the register block. All templates are implemented in parallel using Xilinx Vivado. The result is a set of implemented templates contained within PU slot-sized regions that are ready to be inserted into the connectivity shell. This step compiles on the order of ten minutes, with exact runtime depending on the size of the PU.

3.3 Inserting PUs into the Connectivity Shell

In the final step we populate each PU slot in the connectivity shell with an instance of its matching PU implementation template. As this step happens online, RapidWright is used to ensure it is fast. We use RapidWright to load the connectivity shell implementation and remove all register blocks while preserving the routed nets connecting to the left sides of the blocks. We then use the RapidWright Module API to replicate each PU implementation template into all target PU slots, reconnecting the dangling nets from the connectivity shell to the left sides of the PUs' register blocks. This step takes approximately one minute and results in a completed design.

Figure 6 shows the overall flow for compiling a PU into a full implementation. The implementation of our flow for Amazon F1 is available open source [1]. Our flow includes some automation for convenient description of PU slot patterns. The designer specifies "slot columns" that each have their own PU implementation template replicated vertically multiple times.

4 EXPERIMENTAL RESULTS

In this section we describe our evaluation on the Amazon F1 platform followed by a set of experimental results. At the end we summarize some of our challenges and lessons learned.

[1] https://github.com/jjthomas/Fleet-Floorplanning

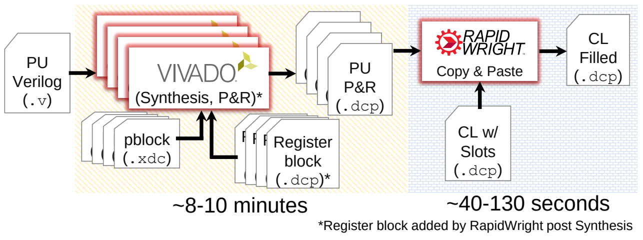

Figure 6: High-level online flow for compiling a PU.

PU	Interface Size (bits)	# Logic Cells	PU Template Implementation Runtime	RapidWright PU Replication Runtime	Our Flow Total Runtime	Standard Flow Runtime	Speedup
Dot	46	71 (incl. DSP)	9m4s	1m1s	10m5s	82m50s	8.2×
Counter	22	109 (incl. BRAM)	10m37s	0m37s	11m14s	87m25s	7.9×
Summer	46	138	8m24s	0m41s	9m5s	82m51s	9.1×
JSON	22	352 (incl. BRAM)	10m57s	0m53s	11m50s	94m30s	8.0×
Time Series Pred.	22	512	8m31s	0m51s	9m22s	95m38s	10.2×
KNN	46	800 (incl. distr. RAM & DSP)	8m25s	1m41s	10m6s	111m15s	11.0×
Integer Coder	46	1119 (incl. distr. RAM)	9m35s	2m7s	11m42s	117m19s	10.0×

Table 1: Runtime results for our 180-PU designs in our flow and the standard Vivado flow

4.1 Target Hardware: Amazon F1 Platform

The target hardware for this work is a Xilinx UltraScale+ VU9P device fabricated using a 16nm process. This device became one of the first FPGA devices commercially deployed for FPGA-as-a-Service (FaaS) in the Amazon F1 instance [2]. The Amazon F1 platform provides 4 banks of 16 GB DDR4 memories and a PCIe x16 generation 3 connection to the host. The existing AWS infrastructure and tools readily facilitate deployment of FPGA designs targeting the platform without the need to purchase development boards. The availability of the F1 as a compute acceleration resource makes it appealing for the purposes of this work aimed at improving the productivity of design development.

The F1 has its own shell built using Xilinx Partial Reconfiguration technology. This shell, which must be present in all F1 designs, includes PCIe, one (out of four) DDR channels, and a few other basic functions. The region outside of the AWS shell is available for user logic and is called the Customer Logic (CL) region. The connectivity shell in our flow connects its external IO ports to the F1 AWS shell. Since the F1 shell is encrypted, the combined shell cannot be read into RapidWright. We had to detach the CL region from the F1 shell (using `write_checkpoint -cell`), load that into RapidWright and fill the PU slots, and then reattach the result to the F1 shell (`read_checkpoint -cell`). This reattachment process was particularly slow, taking around 10 minutes. We do not consider this a fundamental obstacle as it can be addressed by creating shells from open source IP or the time can be reduced in future versions of the Xilinx software.

4.2 Evaluation Results

We choose to use the Fleet [12] compute model to demonstrate a proof of concept. Fleet was designed to use all 4 DDRs in the F1 platform, with applications relying on this combined DDR bandwidth to achieve high performance. However, the provided DDR IPs in F1 CL region are encrypted preventing us from creating the connectivity shell as described previously. Therefore, we decided to modify the Fleet model slightly to send the *same* stream of data to all PUs and use just a single DDR in the F1 shell. Each PU also receives unique configuration data before the start of the shared data stream.

We implemented two realistic applications in this model. The first, "KNN," computes the k-nearest vectors (with k=3 in our tests) in the input stream to the configuration vector. Thus, the k-nearest neighbors in a source dataset for each point in a new dataset can be computed by repeatedly executing this design with the source dataset as the input stream and each PU having a different vector from the new dataset as its configuration, until the new dataset vectors are exhausted. The second, "Time Series Prediction," predicts the sign of the next element in the stream based on comparisons of the previous k elements (k=7 in our tests) with k coefficients from the configuration. The comparison results are passed through a k-input lookup table (also part of the configuration) to get the final result. The PU produces the number of correctly predicted points as its output, allowing many different configurations to be tried across the different PUs to see which one gets the best accuracy.

We implemented five additional synthetic PUs in this model, one ("Summer") which simply sums all data it receives (including the configuration), another ("Dot") which computes the dot product of the input data (assuming the vectors to be dotted are interleaved), another ("Counter") which computes a count for each distinct input word it receives, and two others ("JSON" and "Integer Coder") that are taken from the JSON parsing and integer compression examples in [12]. Our example PUs consist of varying amounts of logic as well as distributed RAM, BRAM, and DSP resources.

KNN, Summer, Dot, and Integer Coder applications consume 32-bit input tokens and produce 8-bit output tokens, so they use a connectivity shell with a 46-bit PU interface. (There are 5 ready-valid and other control bits, and one reset bit.) The Time Series Prediction, Counter, and JSON applications consume 8-bit input tokens and produce 8-bit output tokens, so their connectivity shell has a 22-bit PU interface. Implementing each of these connectivity shells takes approximately 1.5 to 2 hours. The motivation behind this approach is that the implementation effort for each shell is amortized over many different PUs that can slot into it.

Table 1 summarizes our runtime results, with experiments running on a 20-core 2.2 GHz Intel server. We break down the runtime of our flow into the implementation of the PU templates (design optimization, attachment to the register block, placement, and routing) and the replication of the templates into the connectivity shell with RapidWright. The PU templates are assumed to be implemented in parallel. Since we did not have access to a cluster with a job submission system, we simply ran all of the template implementations serially and reported the longest time as the runtime for the step.

Our designs include 180 PU slots arranged in 10 slot columns. Going from left to right across the fabric, the first 4 slot columns contain 30 PU slots apiece, and extend from the top to the bottom of the fabric. The next 6 slot columns only contain 10 PU slots apiece, starting at the top of the fabric and extending to the bottom of the top SLR (the bottom two SLRs in this region are reserved for the F1 shell). Each PU slot is half a clock region (30 logic slices) tall and 4 logic columns wide, and includes one BRAM column and one DSP column.

We compare our flow's runtime to that of the standard end-to-end Vivado flow for the full design, including 180 PUs and the connectivity shell, implemented in the context of the F1 shell. The standard Vivado flow produces only the CL region as output to be on par with our proposed flow. The standard flow is based on the timing-focused implementation flow provided in the F1 development kit, including the `opt_design`, `place_design`, `phys_opt_design`, and `route_design` steps. All designs, in both the standard flow and our flow, were implemented at a clock frequency of 125 MHz as in [12]. This frequency can be increased in our flow by adding more registers into the paths from the connectivity shell to the PU register blocks. We verified that designs produced by our flow worked correctly on the F1 hardware.

The runtime of the proposed flow can be reduced if the PU template implementation step is sped up. As can be seen in Table 1, most of the runtime of this step is Vivado startup overhead. There is fairly little variation in the runtimes for the 7 different PUs, even though they have very different sizes. RapidWright overhead is less, but it lacks a quality placement or routing engine. If PU implementation could be done entirely in RapidWright, our flow could have taken less than three minutes.

4.3 Engineering Guidance and Lessons

In this subsection we describe a number of challenges we encountered and our approach to remedy them. We expect some of these issues will be addressed more efficiently as the RapidWright ecosystem evolves as part of future work.

4.3.1 Routing Global Signals. Our proposal reduces runtime by leveraging PU replication. However, the global signals such as clock, power and ground are not uniformly replicated. Moreover, the PU implementation templates are implemented with the Vivado out-of-context flow, which does not normally perform clock routing. To address this, we instantiate a clock buffer similar to that of the connectivity shell to force clock routing of the PU implementation templates. The clock routing is then saved in a text file and a clock-free design checkpoint is created for the implementation template. When replicating the PU implementation template into the connectivity shell with RapidWright, we read this text file and re-instantiate the clock routing into each copy. A similar procedure is followed for other global signals such as power and ground routing. In the future, this routing will be performed inside RapidWright.

4.3.2 Reduced Utilization. The UltraScale+ architecture is columnar with a heterogeneous column pattern. In order to provide sufficient quantities of contiguous resource types (logic, BRAM, and DSP), our PU slot columns spacing led to reduced utilization because the columns did not contain the right mix. This combined with the area loss due to the hierarchical design flow led to low device utilization. For example, the Integer Coder application with 80% logic utilization in the PU slot, has an overall 16% utilization of the CL region in the final F1 design with 180 PUs. In contrast, we were able to implement an Integer Coder design with 800 PUs using the standard Vivado flow. We believe that there are many memory-bound applications that do not need full fabric utilization to achieve peak performance, as is the case with some of the applications in [12]. Furthermore, applications that do not need all resource types may be able to achieve a PU slot packing with higher utilization; we leave exploration of this design space to future work. Finally, even for applications that need high utilization, this flow may offer a fast means of prototyping and verifying functionality on device before initiating a high-performance implementation run.

PU	Yosys+nextpnr Runtime	# Logic Cells (Yosys synth.)	Our Flow Runtime
Dot	79s	232	544s (4.3×)
Counter	78s	231	637s (5.9×)
Summer	87s	230	504s (4.3×)
JSON	97s	560	657s (4.7×)
Time Series Pred.	100s	1248	511s (3.7×)
KNN	96s	1199	505s (3.1×)
Integer Coder	108s	2803	575s (3.0×)

Table 2: Runtime improvement potential

4.3.3 Scaling PU Interface Size. Our seven applications use PU interfaces sized from 20 to 50 bits. Larger PU interfaces require additional SLR crossing wires (a scarce architectural resource) causing routing congestion. The top SLR has the most available area for PU slots, but there are a limited number of SLR crossing wires available to route nets from the top SLR to the bottom two SLRs with connectivity shell. We believe that applications can easily be adapted to use smaller PU interfaces (e.g. by using the same bus for addresses and data). Another option is to implement a more efficient data movement across SLRs in the connectivity shell.

4.4 Opportunities for Improving Runtime

There has been significant work in the community on building fast, fully open-source implementation flows for FPGAs, such as Yosys for synthesis and nextpnr for place and route [1]. To assess the potential runtime gains achievable with this flow, we applied it to our seven PU benchmarks. These preliminary results are shown in Table 2 and provide a tentative proof point for an additional 4X runtime improvement. These tools do not support routing pblocks or routing around pre-existing routing (required for interface blocks in our proposed flow). We simply implemented the PU verilog without an interface block or pblock with the intention of understanding the future potential for our proposed flow.

The results show that Yosys clearly has worse QoR than Vivado and nextpnr is unable to resolve all the timing issues. However, the potential runtime gains if these issues are fixed is substantial, specially for domain-specific back-end flows. As these open source flows evolve, this seems like a promising route for another order of magnitude gain, approaching software-like compilation times. This exercise also shows that adding open source place and route capability to RapidWright in the future will significantly improve the advantage of our proposed flow.

5 RELATED WORK

Partial reconfiguration (PR) shares similarities to the way designs are constructed in this work. In modular PR, the design and reconfigurable fabric are partitioned into a static (unchanging) region and one or more dynamic (reconfigurable) regions. Early work in this effort, however, focused on minimizing the size of dynamic regions and maximizing the static regions [5] to reduce overall reconfiguration time. In contrast, this work builds monolithic bitstreams, and minimizing the static region would allow for a greater number of PU slots, increasing parallelism of the workload at hand.

One of the overlapping commonalities of this work with partial reconfiguration is the need to implement a physical negotiation interface between static and dynamic regions of a design. Past research has used routing antennas to form gaskets [7] or bus macros [11]. These implementation constructs establish a physical interface to allow general purpose CAD algorithms (place and route) to implement circuits inside restricted rectangles and still connect in a deterministic way to static regions.

This work builds on the techniques described in [9] and [3] that allow for relocatability of implemented modules but includes a decoupling physical interface and implementation tool techniques needed to construct valid circuits. These related works aim to create partial bitstreams whereas this work improves productivity, the time to generate a valid bitstream from a change in design input. Another differentiating factor is scalability of this work, which goes beyond that of [3] and [10] where the number of reconfigurable slots available is less than 10. Our flow targets scale of over 100 slots, achieving the greater scale by leveraging improvements in fabrication technology. Also our flow involves identifying a number of fabric footprint candidates and building out-of-context implementations in parallel to fit each footprint type (or column).

Recently, Xiao and Park et. al. [13] have demonstrated productivity enhancements by having a pre-implemented packet-switched shell with partial reconfiguration regions that decouple compilation of modules so that implementation of each can proceed in parallel. The proposed methodology, which is called PRflow, divides the design into multiple partitions and uses Vivado Partial Reconfiguration (PR) for the back-end; in contrast, our proposed flow uses RapidWright for the back-end dividing process, leading to a number of key advantages. PRflow relies on today's Vivado PR flow requiring place and route to take place within the static context. This will lead to a runtime increase in the PR back-end portion as the design size increases. Our flow differs by relying on out-of-context place and route partitions, providing better results for larger designs and devices where compile time actually matters.

Another key difference for our flow is reducing place and route compute time by replicating PUs. Our approach requires only 10 separate PU implementation runs for the 180-PU designs. In comparison, PRflow requires a separate compile for each leaf block, even if the PR regions and logic are identical. PRflow partitions are confined to the reconfiguration boundaries (necessary for PR). Our RapidWright flow, however, provides a granularity of control for partitions (or PUs) that is 50-60X finer than PR regions. This will provide more opportunities to scale and adapt our flow in the future work. Finally, the best results reported by PRflow heavily rely on open source flows such as nextpnr [1]. As we described in the previous section, while these tools present a promising direction for the future, they are not yet able to provide the timing and performance guarantee that realistic applications require.

6 CONCLUDING REMARKS

We have presented a fast, open source implementation flow for data center FPGA applications where infrequently changing connectivity logic is implemented ahead of time and routed to unoccupied processing unit (PU) slots. This approach allows for a faster development cycle that is up to an order of magnitude faster than conventional flows as only the PU must be re-implemented. We have demonstrated this benefit by leveraging domain-specific attributes and the replication and relocation technology in RapidWright.

We are excited about the possibility that future FPGA architectures with more homogeneous patterns of resource columns will lend themselves to our proposed flow. A more homogeneous columnar pattern implies fewer PU implementation templates and a denser packing of PUs. We expect compile times will improve as the RapidWright ecosystem grows in its capacity to allow more of the placement and routing to be done outside Vivado. In that case, not only will the proposed flow be simpler, but PUs would be compiled solely by RapidWright, further reducing the compile time. We believe the caliber of productivity our flow offers is necessary to expand the use of FPGAs to the wider market of software developers and support acceleration in the post-Moore's Law era.

Author's Note: The opinions expressed by the authors are theirs alone and do not represent future Xilinx policies. To download code from this paper, please visit https://github.com/jjthomas/Fleet-Floorplanning.

REFERENCES

[1] [n. d.]. nextpnr. https://github.com/daveshah1/nextpnr-xilinx.

[2] AWS. [n. d.]. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[3] C. Beckhoff, D. Koch, and J. Torresen. 2012. Go Ahead: A Partial Reconfiguration Framework. In *FCCM'2012*. 37–44. https://doi.org/10.1109/FCCM.2012.17

[4] Alibaba Cloud. [n. d.]. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057

[5] J. D. Hadley and Brad L. Hutchings. 1995. Designing a partially reconfigured system. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, John Schewel (Ed.), Vol. 2607. International Society for Optics and Photonics, SPIE, 210 – 220. https://doi.org/10.1117/12.221341

[6] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[7] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. 2002. Dynamic Hardware Plugins in an FPGA with Partial Run-Time Reconfiguration. In *DAC '02*. Association for Computing Machinery, New York, NY, USA, 343–348. https://doi.org/10.1145/513918.514007

[8] C. Lavin and A. Kaviani. 2018. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *FCCM'2018*. 133–140. https://doi.org/10.1109/FCCM.2018.00030

[9] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. 2011. HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping. In *FCCM'2011*. 117–124. https://doi.org/10.1109/FCCM.2011.17

[10] D. P. Montminy, R. O. Baldwin, P. D. Williams, and B. E. Mullins. 2007. Using Relocatable Bitstreams for Fault Tolerance. In *AHS'2007*. 701–708. https://doi.org/10.1109/AHS.2007.108

[11] Pete Sedcole, Brandon Blodget, Tobias Becker, James Anderson, and Patrick Lysaght. 2006. Modular dynamic reconfiguration in Virtex FPGAs. *IEE Proceedings-Computers and Digital Techniques* 153, 3 (2006), 157–164.

[12] J. Thomas, P. Hanrahan, and M. Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *ASPLOS'2020*. ACM, New York, NY, USA.

[13] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, and A. DeHon. 2019. Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks. In *ICFPT'2019*. 153–161. https://doi.org/10.1109/ICFPT47387.2019.00026