Case for Fast FPGA Compilation using Partial Reconfiguration

Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon
Dept. of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA
Email: dopark@seas.upenn.edu, ylxiao@seas.upenn.edu, magnezin@umd.edu, andre@ieee.org

***Abstract*—Despite the FPGA's advantages over other hardware platforms, long compilation time prevents FPGA engineers from efficiently exploring the design space and discourages new users who want to quickly iterate for debugging. To reduce compilation time, this work adopts a divide-and-conquer approach using Partial Reconfiguration with a Packet-Switched Fat-Tree network. Partially reconfigured leaves in the packet-switched network are independent from each other and can be compiled separately in parallel. Also, when a minor fix is required to a bitstream, only the corresponding leaves need to be incrementally compiled. Preliminary experimental evidence from our work-in-progress effort illustrates how a 30 minute full-chip compile time can be reduced to 7 minutes.**

I. Introduction

Compilation time for FPGAs is notoriously long. For large devices, compilation can take hours. For example, post-synthesis place and route of a 50% utilized ZU9EG MPSoC FPGA using Vivado 2016.4 can take half an hour on a 2.7 GHz Intel E5-2680 processor. This is in sharp contrast to compilation for processors and GPUs that typically take seconds to minutes, especially when only making small changes to the application. This makes FPGA use unattractive. It forces a long edit-compile-debug cycle that is inconsistent with modern software development idioms. For novices, the long edit-compile-debug cycle challenges the attention span and frustrates the developer. For experts, the long cycle limits the design-space exploration that can be performed within a limited time-to-market window. Together, *this slow compilation is holding FPGAs and Reconfigurable Computing back form its full potential—limiting the set of developers who will try to use it, limiting the rate at which designs can be developed, and limiting the quality of designs that are deployed.*

We believe the time is right for a methodology and tool chain to support fast compilation of FPGA designs onto today's large-scale FPGA parts. We explore a divide-and-conquer model that allows large FPGA compilation to be decomposed into a collection of smaller compilation tasks that can be run in parallel. The small compiled blocks are assembled using partial reconfiguration and interconnected using a lightweight, packet-switched overlay network (See Fig. 1). This model allows designs to be compiled in minutes, gated only by the time to compile small leaf functions to regions of the chip. For example, a 2700 LUT leaf block on the ZU9EG can be implemented (LUT mapping, placement, and routing) in 7 minutes. During incremental development, a single leaf function can be compiled quickly and independently and reintegrated into the design, just as software compilers can separately compile only the files that changed and link them into a complete application. For larger changes, cloud resources can recompile all the component leaf functions in parallel. For example, with a cluster of 8 compute servers, each with two 2.7 GHz Intel E5-2680 CPUs and 128 GB of RAM (total of $8 \times 2 \times 8$=128 cores), we run 31 parallel Vivado post-synthesis place and route jobs to implement the entire design for the ZU9EG in 7 minutes.

This methodology deliberately sacrifices routed design quality and density for low compilation and recompilation latency. It pays area for an overlay network and sacrifices area to fragmentation in the partial reconfiguration blocks. Separate compilation prevents co-optimization of functions. Connectivity through the overlay network is lower than the raw wiring provided by the FPGA and increases the latency between leaf blocks. Given the high capacity FPGA devices that decades of Moore's Law scaling has produced, we have long emerged from the era of poverty [1] and can now afford to spend some of this capacity to accelerate development, particularly during the early lifetime of a design.

Current FPGAs and vendor tools are not deliberately designed for this purpose. This work explores how we can make the most of the existing designs.

Our novel contributions include:

- Strategy for using Partial Reconfiguration to reduce FPGA post-synthesis place and route time, including both incremental compilations that allow recompilation of only the leaf functions that change and parallel compilations that exploit multi-core and cloud capacity to reduce full-chip (re)compilation latency (Sec. III)
- A case study that concretely illustrates the potential benefits and challenges of this strategy when built upon current Xilinx tools and FPGAs (Sec. V)

II. Background and Opportunities

A. Partial Reconfiguration

Partial Reconfiguration (PR) allows portions of the FPGA to be reconfigured independently at some granularity without the need to rewrite the configuration bits of the entire FPGA. This has traditionally allowed portions of the FPGA functionality to be replaced during runtime without disabling complete operation [2], [3]. When only small edits are needed to the logic on the FPGA, this speeds reconfiguration by reducing

the amount of data that must be loaded onto the FPGA through the limited-bandwidth configuration path.

In order to support PR designs, vendors [4], [5] and reconfigurable computing developers [6] provide methodologies and tool flows that allow logic to be constrained to and mapped for a small region of the FPGA. PR designs are decomposed into a static region and multiple reconfigurable regions. The static region refers to the part of the design that is never partially reconfigured, and the reconfigurable regions refer to the part that is recompiled and reconfigured by downloading partial bitstreams. To identify reconfigurable regions, the designer defines a set of physical resources on the FPGA as a *p-block*. Many, independent p-blocks can be defined for the FPGA, providing many different regions that can be independently reconfigured. A portion of the logical design can then be identified as a *reconfigurable partition* that can be assign to a particular p-block.

Since implementation time is influenced both by the size of the logical design (e.g., number of LUTs in the user's design) and the physical substrate (e.g., number of CLBs on the target FPGA or region of the FPGA), to implement a design that is a tiny fraction of the FPGA capacity to a region that is a small fraction of the FPGA capacity should be faster than performing a full-scale FPGA implementation. Nonetheless, the main use of PR to date has been to reduce the area required for a design, by loading only the logic needed at a particular time onto the FPGA, and to reduce the reconfiguration time when new functions must be loaded onto the FPGA. Previous research provides methodologies for using PR regions as slots that are dynamically loaded with hardware modules during execution [7], [8], [9]. However, this work does not address compilation time.

B. Out-of-Context Compilation

Vivado provides *Out-of-Context* (OoC) compilation that allows developers to compile leaf IP blocks without the "context" of the entire enclosing design [10]. This allows groups to divide the work among team members and supports independent synthesis and implementation runs for IP blocks.

III. Strategy

The basic idea is to break the FPGA into a set of separate p-blocks that can be compiled and recompiled separately. We assume the user design is a collection of interconnected components or IP-blocks that we generically call *leaf blocks*. Ideally, each leaf block will be mapped to a p-block.

This raises a number of questions:

- how are leaf blocks interconnected?
- how can we deal with leaf blocks that vary in size?
- how do we prepare designs to fit this framework?

A. Interconnecting Leaf Blocks with Overlay Network

The standard PR model with a fixed-logic static-region would constrain the interconnection between leaf blocks. However, we want to support arbitrary designs for development. We could re-compile the static region, but that would be a long compilation task that would undermine our goal of fast compilation. To avoid this compilation time, we use a lightweight, packet-switched overlay network as the top-level infrastructure for connecting leaf blocks. Once the leaf blocks are loaded and configured, they are all set to communicate. We specifically use a Butterfly Fat Tree (BFT) topology [11], [12] that can be parameterized and tuned to provide different levels of interconnect according to Rent's Rule [13].

B. Leaf p-blocks

The PR regions for leaf blocks are a collection of leaf p-blocks of predefined sizes. As long as a leaf block is smaller than a p-block, it can be placed in the p-block. This may waste some logic resources due to internal fragmentation. Leaf blocks larger than the p-block will need to be decomposed. A future extension could combine multiple primitive p-blocks into a larger p-block that will accommodate the logic.

C. Compute Model

We assume computations are composed of operators, which we call *user modules*, that are connected through streaming dataflow links with data presence handshaking in the style of a Kahn Process Network [14] similar to the Ambric [15] or SCORE [16]. User modules can be arbitrary logic or memory functions and can be written in any language (e.g., Verilog, C, BSV) as long as they follow the stream communication discipline. Threads running on the embedded, hardcore processors can also communicate through stream links. Together, the dataflow streaming model accommodates the variable delay introduced in the pipelined, packet-switched network and the delay changes that arise as the implementation or placement of leaf blocks change.

D. Design Flow

Standard Vivado compilation consists of three main steps: synthesis, implementation, and bitstream generation. OoC Synthesis (Sec. II-B) can be used to exploit parallelism when synthesizing the user modules. We focus on using PR to accelerate the implementation process so that the full compilation process can be accelerated.

The inputs of the system are synthesized user modules. The user modules are assigned to the appropriate sizes of leaf p-blocks based on their synthesis utilization reports. Our goal is to run implementation on each of the user modules independently, allowing either a single user module to be changed at a time or for a set of user modules to be implemented in parallel. In Fig. 1, six of the configurations are run in parallel, and each configuration implements only one leaf p-block, leaving other leaf p-blocks empty. Since the size of the design with a single user module filled in is smaller than that of the monolithic approach with all the user modules instantiated, the separate compilations should be faster than the monolithic compilation. For the same reason, we put the BFT in its own reconfigurable p-block so that its resources do not contribute to the size of the static region and slow down leaf implementation. One leaf of the BFT is connected to the ARM, and the ARM configures the network informing the leaves where to send the packets.

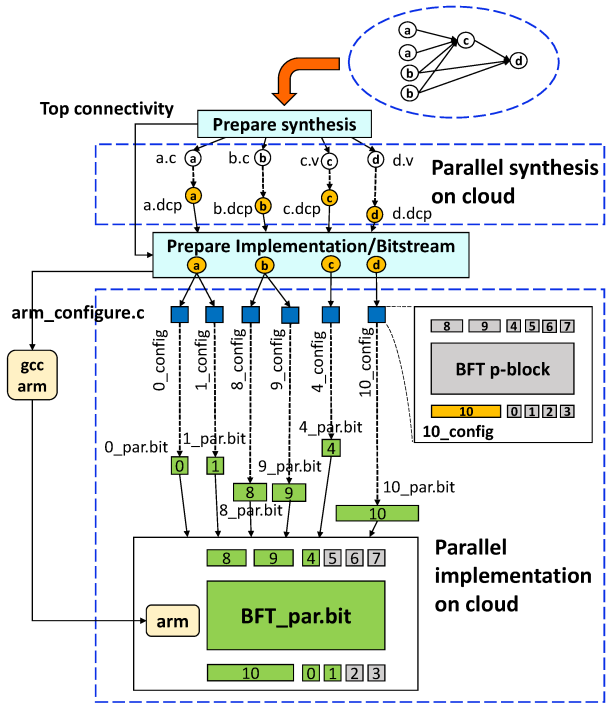

Fig. 1. Separate Compilations of Leaf Blocks

Fig. 2. Floorplanning of BFT-32 and Leaf Blocks on ZU9EG

IV. Design Point

For concrete discussion and evaluation in this and the following section, we specifically consider a Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board which uses an UltraScale+ ZU9EG FPGA (274,080 LUTs, 548,160 FFs, 912 36Kb BRAMs, quad ARM Cortex-A53 processors). Vivado 2016.4 is used for the compilation of the design running on 2.7 GHz Intel E5-2680 CPUs with 128 GB of RAM.

As a base design, we use a p=0.5 BFT (alternating t and π switches, provides the same bandwidth growth as a mesh) with 97b, single-flit packets (64b payload, 32b address, 1b valid). For these parameters, the BFT is roughly 1000 LUTs per leaf supported. If we make each p-block for a leaf about 3900 LUTs and include 11, 36Kb BRAMs, that means we need about 1000+3900=4900 LUTs per leaf. Placing a 31-leaf BFT on an MPSoC ZU9EG uses 151K (55%) of the 274K LUTs. 20% of the logic goes into the BFT overlay network. Leaf interface logic can be as small as 471 LUTs and 3, 36Kb BRAMs and will scale up with the stream links supported. At this size, the leaf interface will consume 10% of the used logic. This leaves about 3400 LUTs or about 70% of the used logic for each user module. At 3900 LUTs, each leaf module is only 1.4% the size of the full 274K FPGA.

V. Case Study: Customizing Array of Processors and Accelerators

A. MicroBlaze

To measure the effectiveness of fast compilation using the parallel approach, an array of MicroBlaze processors [17] is chosen as a simple benchmark. The size of MicroBlaze varies from 500 LUTs to more than 4000 LUTs depending on options chosen. The default user modules contain a MicroBlaze with an AXI-lite interface. For this case study, we use a 32-leaf BFT with 31 mostly identical p-blocks with around 3900 LUTs and 11–24 BRAMs.

This array of MicroBlazes might represent the beginning of a design optimization where the user starts with all their computations written in C and running on the MicroBlazes. Since each MicroBlaze is independent of each other, the designer can optimize the design by customizing the MicroBlaze parameters tailored to the operator. Further, the designer can exploit HLS to replace the MicroBlaze with custom hardware IP-block in a leaf block. With this setup, the designer can incrementally migrate, test, and refine each leaf operator with quick implementation turns using our flow.

B. Parallel Compilation

Our primary experiment is to compare the runtime for a leaf block PR build to the runtime for a monolithic, full-chip build with the standard Vivado flow. Our design can be assembled by running these leaf block builds in parallel (Fig. 1) or one at a time, whereas a traditional flow would demand the sequential, monolithic build be run for any changes. We show the maximum time for the single leaf runs. The total time for our scripts to setup these runs and process the result is just a few seconds and hence negligible compared to the Vivado implementation runtimes, so we do not report them here.

Tab. I shows the result of sample experiments with different MicroBlaze configurations. We see the parallel, single leaf implementation and bitstream generation runs are significantly faster than the monolithic run, but not as fast as we might expect based solely on the size of the leaf logic. As stated earlier (Sec. III-D), we put the BFT with ARM Leaf in a seperate p-block, and our approach is much faster than the approach that has BFT with ARM Leaf in the static region ("BFT Static" column).

The synthesis runtime for parallel approach is merely that for synthesizing Microblaze system with the leaf interface since that is all that is included in the leaf block. Synthesis for the monolithic approach takes much more than for parallel

TABLE I
COMPARISON OF COMPILATION RUNTIME

runtime(secs)	μBlaze(v.0)			μBlaze(v.1)		μBlaze(v.2)	
	Mono	BFT Static	Parallel	Mono	Parallel	Mono	Parallel
synthesis	3171	287	287	3118	283	2510	235
impl+bitstream	1797	857	418	1692	413	1283	398

	BFT-32	p-block	μBlaze(v.0)	μBlaze(v.1)	μBlaze(v.2)
LUTs	29774	3840∼3960	2722	2328	1433
BRAM	0	11∼24	8	8	4

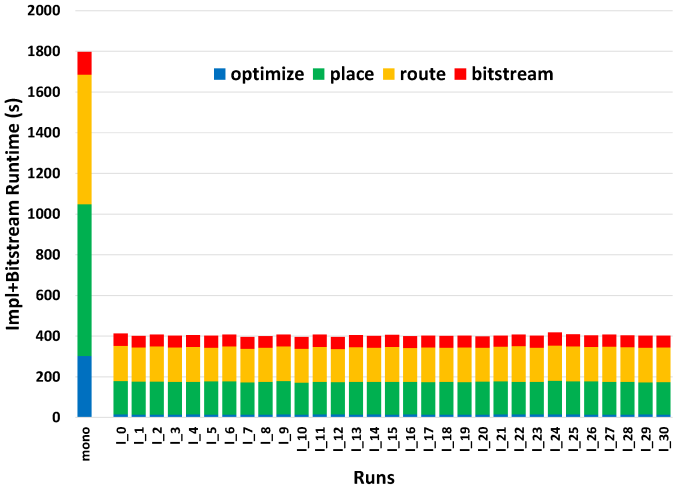

Fig. 3. Monolithic vs. Parallel for μBlaze(v.0)

approach. This can be exploited using OoC synthesis runs (Sec. II-B) for an order of magnitude speedup independent of our strategy. The runtime improvement on the implementation process shows the impact of our parallel compilation flow.

Fig. 3 shows the runtime composition of implementation and bitstream generation process for monolithic approach and all 31 runs that are executed in parallel. Because 31 parallel runs implement the identical designs in the identical sizes of the leaf p-block, the runtimes are almost the same. While the bitstream generation is not accelerated much, the implementation process decreases due to the smaller problem size.

VI. CONCLUSIONS

We show how to use partial reconfiguration capabilities of modern FPGAs coupled with lightweight, packet-switched overlay networks to decompose the normally monolithic and slow compilation tasks into separate, smaller compilation problems that can be run independently. This allows both faster incremental compilation when only small parts of the design change and parallel compilation when the design is new or has significant changes. Modern FPGAs and vendor flows are not necessarily tuned to this usage, limiting the magnitude of the benefit. Nonetheless, we show how to use Vivado's existing facilities to achieve a 4× speedup and get implementation time down to 7 minutes when decomposing the design into leaf blocks with a few thousands LUTs.

ACKNOWLEDGMENTS

This work is funded in part by the Office of Naval Research under grant N000141512006. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. Xilinx donated Vivado tools for use in this work.

REFERENCES

[1] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, March 2015.

[2] J. D. Hadley and B. Hutchings, "Design methodologies for partially reconfigured systems," in *FCCM*, April 1995, pp. 78–84.

[3] *Virtex Series Configuration Architecture User Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, October 2004, XAPP 151. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp151.pdf

[4] D. Lim and M. Peattie, *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, May 2002, xAPP 290 <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>.

[5] *UG909: Vivado Design Suite User Guide: Partial Reconfiguration*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, December 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug909-vivado-partial-reconfiguration.pdf

[6] C. Beckhoff, D. Koch, and J. Torresen, "Go ahead: A partial reconfiguration framework," in *FCCM*, April 2012, pp. 37–44.

[7] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. B. W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Run-time support for heterogeneous multitasking on reconfigurable socs," *INTEGRATION, The VLSI Journal*, vol. 38, no. 1, pp. 107–130, 2004.

[8] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The Erlangen slot machine: A dynamically reconfigurable FPGA-based computer," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 47, no. 1, pp. 15–31, 2007.

[9] K. Vipin and S. A. Fahmy, "Automated partial reconfiguration design for adaptive systems with CoPR for Zynq," in *FCCM*, May 2014.

[10] *UG946: Vivado Design Suite Tutorial: Hierarchical Design*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, Aprial 2015. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug946-vivado-hierarchical-design-tutorial.pdf

[11] C. E. Leiserson, "VLSI theory and parallel supercomputing," MIT, 545 Technology Sq., Cambridge, MA 02139, MIT/LCS/TM 402, May 1989, also appears as an invited presentation at the 1989 Caltech Decennial VLSI Conference. [Online]. Available: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA214035

[12] N. Kapre, "Deflection-routed butterfly fat trees on FPGAs," in *FPL*, Sept 2017, pp. 1–8.

[13] B. S. Landman and R. L. Russo, "On pin versus block relationship for partitions of logic circuits," *IEEE Trans. Comput.*, vol. 20, pp. 1469–1479, 1971.

[14] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP CONGRESS 74*. North-Holland Publishing Company, 1974, pp. 471–475.

[15] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM*, April 2007, pp. 55–64.

[16] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE): Extended abstract," in *FPL*, ser. LNCS. Springer-Verlag, August 28–30 2000, pp. 605–614.

[17] *UG984: MicroBlaze Processor Reference Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, November 2016. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug984-vivado-microblaze-ref.pdf