EFCAD – an Embedded FPGA CAD Tool Flow For Enabling On-Chip Self-Compilation

DOI:
10.1109/FCCM.2019.00011

EFCAD – an Embedded FPGA CAD Tool Flow For Enabling On-Chip Self-Compilation

***Abstract*—This paper combines a chain of academic tools to form an FPGA compilation flow for building partially reconfigurable modules on lightweight embedded platforms. Our flow — EFCAD — supports the entire stack from RTL (Verilog) to (partial) bitstream, and we demonstrate early results from the on-chip ARM processor of, and targeting, the latest 16nm generation of a Xilinx UltraScale+ FPGA-SoC device. With this, we complement Xilinx's PYNQ initiative to not only facilitate System-on-Chip research and education entirely within an embedded system, but also to allow building new and specialising existing custom-computing accelerators without needing access to a workstation.**

I. Introduction

Presently, the FPGA tool landscape is dominated by closed source tools provided by FPGA vendors or EDA tool design houses. This is in contrast to software programming with its huge ecosystem that includes a large number of open-source projects for compilers and all kinds of software design tools. One reason for this disconnect is that FPGA vendors do not disclose low-level device-specific information (e.g. the bitstream encoding). While this makes reverse engineering of IP from a bitstream harder (but not entirely impossible [1]), it prevents portability across FPGA vendors and restricts the design of domain-specific implementation tools.

In this paper, we are introducing EFCAD — an Embedded FPGA CAD Tool Flow which can run on a normal PC or directly within an embedded system. Distinct features of EFCAD include 1) a full, FPGA vendor tool independent, open-source flow from Verilog to (partial) bitstreams and 2) support for recent Xilinx Zynq UltraScale+ FPGAs. EFCAD integrates several academic projects into an end-to-end flow. As shown in Figure 1, the Xilinx vendor tools are only used to generate an architecture graph to be used by our run-time flow that then runs entirely on the ARM core of a Zynq UltraScale+ FPGA. In detail, the components include:

- *Yosys* is a popular open-source Verilog logic synthesis tool supporting all common Verilog-2005 features [2]. Consequently, EFCAD accepts the same Verilog features. Yosys also performs technology mapping (into LUTs and flops), which is based on ABC [3]. The tool is very generic and supports for example mapping to Lattice iCE40, Lattice ECP5, and Virtex-7 FPGAs including BRAM inference.
- *nextpnr* is a generic open-source place and route tool that aims at portability across different FPGA vendors and device families [4].
- *GoAhead* is an academic tool for building complex reconfigurable designs for Xilinx FPGAs [5]. GoAhead can parse in XDL and TCL device descriptions that are generated by the Xilinx tools and builds up a full architecture graph for all recent Xilinx FPGAs (S6, V6, entire 7-series and UltraScale devices). Reporting functions in GoAhead are used to export the architecture graph for nextpnr.

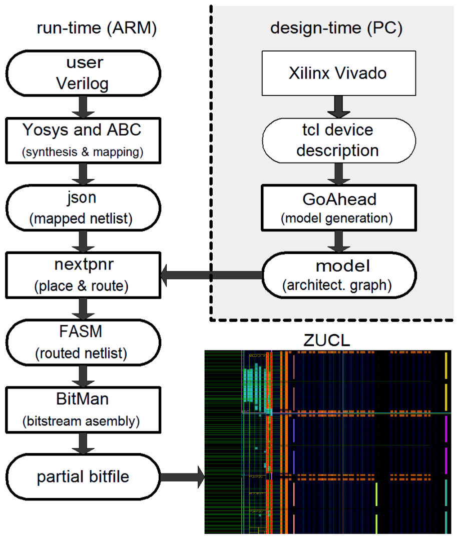


Fig. 1. The EFCAD flow.

- *BitMan* is a tool and library for bitstream manipulations on Xilinx FPGAs [6]. BitMan is designed to operate with designs created with the GoAhead tool. It is used as a tool when implementing reconfigurable systems (the static system/shell as well as for reconfigurable modules). BitMan also performs module relocation in the run-time system and deals with clock-net configurations, if needed.
- *ZUCL* is an open-source shell/static system for Xilinx Zynq UltraScale+ FPGAs for hosting partially reconfigurable modules [7]. It provides a runtime system for configuring (including module relocation) and running modules. It also comes with compile scripts to build the actual partial module bitfiles.

II. Use Cases

There are many reasons for an open-source tool chain in general and embedded CAD tools in particular. The following scenarios give some ideas and are not meant to be complete:

- **Education** Open-source tools with clean interfaces between all individual processes steps are key for FPGA research. EFCAD supports this over the entire stack and allows therefore experiments and testing on real hardware. In the case of embedded tools, it removes the burden to install, maintain and operate heavy FPGA vendor tools. EFCAD could be used for building lightweight self-contained out-off-the-box education solutions that only needs a screen or terminal. This aligns with the PYNQ initiative from Xilinx that is an educational platform for SoC design under Python [8]. That platform, however, uses pre-built bitstreams and still relies on the Xilinx Vivado suite for building hardware. EFCAD complements this initiative by allowing Pynq users to implement peripherals or other circuits right on the prototype platform.

- **Virtualization** FPGA bitstreams are impossible to be ported between different hardware platforms. Unlike what we know from, for example, smart phone app-stores where binaries can be downloaded and executed on a wide range of different devices, that is infeasible with FPGA accelerators. Even when using the same FPGA device, this will still not easily allow using identical configuration bitstreams. For example, the ULTRA96 and the UltraZed-EG board provide the same FPGA, but I/O pins are connected differently, which in turn requires different static configurations (shells) and consequently different module implementations. However, with EFCAD place and route could be carried out by the system itself taking into account the specific system constraints.
  This approach could be considered further for masking defect or imperfect resources. In this scenario, a system may create a defect mask that will then be incorporated through the model that is used by EFCAD. The idea was proposed in [9] by using a networked workstation that performs the device specific physical implementation. EFCAD would allow such an approach in remote scenarios where transferring data is infeasible or impossible.
- **Partial Reconfiguration** Building partially reconfigurable systems requires obeying some dedicated constraints for the physical implementation. In our case study, for example, we ensure that a module is implemented into a fixed physical bounding box (i.e. a partial region) by only providing a model of the partially reconfigurable region to nextpnr (we call that sandbox). With this, by definition, any completed routing process will obey the bounding box. Similarly, fixing interfaces with the surrounding static system or allowing static system routing to cross reconfigurable regions is very easy to implement in EFCAD but very hard using the Xilinx vendor tools, which holds in particular for supporting module relocation. For instance, in our case study, we allow the static system to use long distance wires (i.e. wires that route a distance of 12 switch matrices on Xilinx UltraScale) for crossing partial regions, if needed. These wires are not included in the sandbox model provided to nextpnr as most of these wires would leave the bounding box constraint anyway. Even when using different long-distance wire paths in different regions, that would not impact relocatability of partial modules as we keep the resource footprint (i.e. the relative positions of logic columns) identical. BitMan supports this as it is possible to merge different netlists into a combined bitstream.
- **Security** Removing wires from the model can be applied as an answer to recent wire tapping [10], [11] security attacks as an attacker would be prohibited to use the required wire resources. For the same reason, BitMan can be used with a truncated bitstream model database so it would then be impossible to create malicious circuits exploiting the mentioned scenarios or malicious bitstream encodings [12].
  Moreover, the possibility of compiling in the field can provide unique security solutions. For example, keys for an encryption algorithm maybe compiled directly into a hardwired netlist without the need to send sensitive information off-site.

III. Related Work

There is a large body of academic and open-source FPGA tools addressing various aspects of FPGA design or FPGA research in general. The following projects are prominent examples that are closely related to EFCAD.

Originally developed for FPGA architecture exploration, the Versatile Place and Route (VPR) tool [13] is the seminal work on place and route tools. This project went through several major revisions and is packaged today as Verilog-to-Routing VTR [14]. VTR, in turn was used in [15] to build a CAD flow targeting Xilinx Virtex-6 FPGAs from Verilog all the way to fully placed and routed netlists using entirely open-soure tools. Only for the final bitstream assembly were the Xilinx vendor tools involved (through the XDL API [16]).

Originally designed for rapid prototyping, the tool HMFlow (developed at Brigham Young University) was designed to compose systems together from (physically implemented) hard-macros [17]. For bulding and manipulating macros and netlists in general, the tool RapidSmith was developed [18].

All previously mentioned tools still rely at least for the final bitstream generation on the FPGA vendor tools (here Xilinx), hence making self-compilation prohibitive. Approaches targeting bitstream manipulation include PARBIT [19] and JBits [20]. They both provide simple netlist/circuit manipulations at bitstream level (which could be carried out by an embedded system, but however, both lack a frontend that allows specifying real-world circuits. This points out that it needs a full compilation flow from a hardware description (e.g., Verilog) all the way to the final bitstream assembly to perform real self-compilation on an FPGA platform in the field.

A full compilation flow could be implemented through Project X-Ray which is an open-source documentation effort for Xilinx 7-series FPGAs [21].

EFCAD is based on established open-source CAD tools from VTR and BitMan for the final bitstream assembly. BitMan has a bitstream database for 7-series, UltraScale, and UltraScale+ devices, and it provides open-source and closed-source parts. The bitstream manipulation library needed for building partially reconfigurable systems is freely available while the BitMan bitstream backend, as used in EFCAD, is only available as an executable binary.

IV. Restrictions

EFCAD is a complete vendor independent CAD tool flow featuring latest FPGA UltraScale devices, and is a strong demonstration of what academic tools can do today. However, UltraScale devices are substantially more complex than previous FPGA architectures and the present EFCAD flow comes with several restrictions. While we could have targeted older and less complex architectures to circumvent such restrictions, we believe it is more important to follow the state-of-the-art. Regardless, even with EFCAD having several restrictions, it has enabled us to identify interesting new research questions;

and due to the open nature of the EFCAD flow, those can now be tackled. The most important restrictions include:

- **CLA Support** In the UltraScale architecture, Xilinx moved for the first time from carry-chain-based arithmetic to a faster carry-look-ahead structure (CLA) which is not currently supported by Yosys. Comparators, counters and adders get consequently mapped entirely to LUTs.
- **Fractual LUTs** 7-Series FPGAs allow a 6-input LUT to be used as a dual 5-input LUT with independent logic functions for the same input signals. UltraScale supports the same feature, but the extra slice outputs are routed differently and are not exported in the report files generated by the vendor tools; meaning that GoAhead cannot include these in the FPGA architecture graph. A practical solution would be to move this routing (that in the previous generations' models belonged inside the logic) into the switch matrices. However this would have to be done manually, which is feasible due to the high regularity of the fabric.
- **Memory Inference** While Yosys can infer BRAMs directly from Verilog, we have not enabled this feature in EFCAD so far. UltraScale devices provide distributed memory in a subset of the Slices, BRAM, and UltraRAM. These memory primitives differ substantially in capacity and modes of operation. An easy solution would be to offload the problem to the designer by instantiating primitives directly, which is not an unusual design practice, and one taken by VTR [14]. In the future, we plan on enabling this feature in Yosys and using GoAhead to deliver the required routing information. However, inferring memory primitives automatically and efficiently is still an open challenge.
- **DSP-Blocks** UltraScale devices have more complex DSP blocks than in prior families, which for example, have 4 input operands and more bits. We therefore decide to omit DSP blocks in this first version of EFCAD. However, as a first step, UltraScale DSP blocks could be used in exactly the same mode as done for 7-series devices in Yosys.
- **Clocks and Timing Closure** While Yosys can deal with multiple clock domains, we omitted this feature in EFCAD because the interface to the ARM SoC in our prototype system provides clock domain crossing and our demonstration circuits are rather baseline. As a simplification, we pre-route all sequential primitives (i.e. all flops in the reconfigurable sandbox region) to a global clock tree. An interesting feature (yet still not enabled in the Vivado tools either) are delay elements in the clock tree [22]. Starting from the same horizontal clock wire, these elements can generate multiple phase shifted vertical clock splines that ultimately connect to the flip-flops. This could be used to allow more advanced implementation techniques like time-borrowing and glitch masking.
- **I/O** EFCAD has no model of any UltraScale I/O primitive and modules communicate through predefined wires only. Therefore, the vendor independent FPGA compilation path is only available for partial modules. Instantiating I/O pads or even more complex peripherals as primitives would be a feasible way to offer full system design in EFCAD, however, BitMan has no understanding of the bitstream information about the internals of I/O cells (e.g., I/O voltage standards, drive strengths, phase shifts). These sections of the bitstream are always kept as generated by the Xilinx tools.

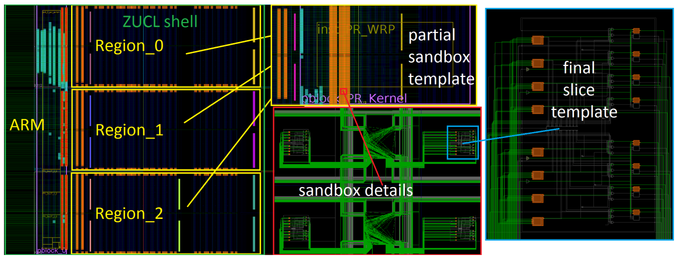


Fig. 2. Sandbox design. left) ZUCL shell hosting up to three sandboxes.

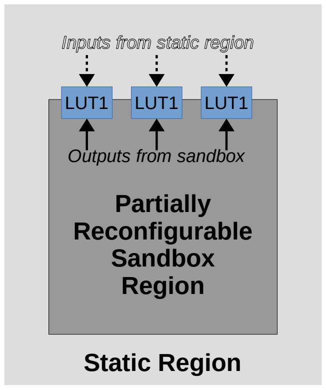
V. System Sandbox Creation

For testing EFCAD implemented modules, we are using the ZUCL open-source shell for the UltraZed board [7]. As shown in Figure 2, the XCZU3EG FPGA in that system uses a small static system to connect three larger regions to the ARM SoC. The static system is in charge of decoupling the partial regions during reconfiguration and to provide a static clock. Each of the three large reconfigurable regions provides a dedicated AXI master and AXI-Lite connection through a standardized interface that is routed identically in each region so as to permit module relocation.

In one of these regions, using the ZUCL partial module compilation scripts, we implemented a sandbox containing one dummy module (essentially a set of 6-input XOR gates with combinatorial and sequential outputs enabled) that occupies all available LUTs and corresponding adjacent flops. Within this sandbox, all LUTs are configured in their 6-input/1-output mode (where the output is provided directly from the LUT and passing through a flop), all flip-flops are connected to the clock tree, and the routing of these primitive outputs were constrained to specific wires. By doing this, we create a "superset" configuration template for use by EFCAD, from which BitMan can then manipulate to implement the required configuration only by adjusting LUT tables and switch matrix settings. After this, a partial configuration bitstream of the sandbox connected to the AXI Lite interface is generated and duplicated to all three slots. We also connect a PMOD with each region. In this system, we can load up to three individual sandboxes. However reconfigurable slots can host other modules, like OpenCL accelerators, as intended by ZUCL. The previous steps are all carried out mostly with the help of the Xilinx vendor tools.

VI. On-Chip Self Compilation

We demonstrate the feasibility of on-chip self-compilation on the Ultra96 development board containing a Xilinx UltraScale+ MPSoC ZU3EG device, with a 1.3GHz ARM Cortex-A53 processor, 2GB of RAM, running Ubuntu 16.04. Since Yosys and nextpnr are open-source tools, we were able to recompile these for the ARM architecture, which coupled with a version of BitMan that the authors of [6] have kindly provided, allows us to execute EFCAD entirely within this embedded platform.


Inputs from static region
LUT1
LUT1
LUT1
Outputs from sandbox
Partially Reconfigurable Sandbox Region
Static Region

Example Top-Level Verilog:

```
module top;
  wire reset, [31:0] odata, ...;

     // Do not optimise away
  (* keep,
     // Placement constraint
     LOC="X14Y60.A" *)
  LUT1 resetn_in (.O(reset));

  (* keep, LOC="X16Y100.A" *)
  LUT1 odata0_out (.I0(odata[0]));
  (* keep, LOC="X16Y100.B" *)
  LUT1 odata1_out (.I0(odata[1]));

  ...

endmodule
```

Fig. 3. Use of location-constrained 1-input proxy LUTs to enter/exit reconfigurable sandbox region, as well as preprocessed Verilog for EFCAD.

As shown in Figure 1, GoAhead first extracts a device description from the vendor tools offline, and prepares a flattened architecture model containing a uniquified list of all available primitives (in this preliminary version of EFCAD, only the LUT+FF primitive is supported), and their (X,Y) location as well as the associated pins on each (e.g. `A1`,`B2`,...,`G`,`HQ`). Also in this architecture model is the list of programmable routing switches available from each pin or wire, as well as a crude delay estimate – for example, the `FQ` output pin has a switch connecting it to a `LOGIC_OUTS_W22` wire, and from that wire a switch exists to `INT_NODE_SDQ_78_INT_OUT0` and so on.

EFCAD starts with the invocation of Yosys to synthesise a Verilog design into a JSON netlist format accepted by nextpnr. nextpnr accepts the architecture model produced earlier by GoAhead in a CSV format, and uses the parsed information to generate a compact in-memory data-structure, before proceeding to timing-driven place and route of the design. Since EFCAD does not currently support I/O primitives, entry into and exit from each partially-reconfigurable sandbox region requires "proxy LUTs" — where a one 1-input LUT is used per signal that crosses the sandbox — and is accomplished by preprocessing the Yosys design as in Fig. 3. The output of nextpnr is a FASM file, which contains a list of placed primitives with their LUT init values (truth table for each LUT), as well as the list of routing switches to be enabled.

Lastly, BitMan reads this file and translates its contents into the corresponding configuration bit settings. The process is completed in memory and only after the full partial bitstream assembly, the changed configuration frames are written to the fabric (using standard PCAP mode). We can therefore say that our nextpnr modules are partially reconfigurable modules inside partially reconfigurable modules (i.e. the sandbox infrastructure), a feature not supported by the Xilinx vendor tools.

A. Experimental Results

The feasibility of running Verilog synthesis and place-and-route entirely within the embedded platform, for a 3840 LUT sandbox region, is shown in Table I. As can be seen, even for our larger test case, peak memory consumption was less than 200MB which is well within the 2GB RAM available on the ARM SoC. Execution times scales poorer than we expect, but as the toolchain continues to mature and we continue to optimise EFCAD, we believe that we can improve this significantly. Future work would also include evaluating on a greater, more varied, set of benchmarks.

Benchmark	LUT+FF primitives	Routing arcs	Yosys time (s)	nextpnr time (s)	Peak Mem (MB)
32bit "blinky"	59	165	13	14	148
256bit "blinky"	492	1280	83	63	148
16-tap 23-bit FIR *(Fixed Coefficients)*	1175	4347	113	614	159
picorv32 *(RISC-V CPU)*	2223	8474	266	4163	170

TABLE I
EFCAD Metrics From Running on Xilinx UltraScale+ MPSoC.

VII. Conclusion

The EFCAD flow presented here enables in-system compilation from Verilog to bitstream without using Xilinx vendor tools. While the flow needs more testing and more features (as highlighted in Section IV, our first experiments demonstrate that the flow is very feasible to run an ARM SoC of a Zynq UltraScale+ FPGA using only moderate memory requirements and processing time. With this, we enable new directions in research, education and FPGA embedded system design in general.

References

[1] J.-B. Note and E. Rannaud, "From the Bitstream to the Netlist," in *FPGA*, 2008, pp. 264–264.
[2] C. Wolf, "Yosys Open SYnthesis Suite," http://www.clifford.at/yosys/.
[3] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification," http://www.eecs.berkeley.edu/ alanmi/abc/.
[4] C. Wolf et al., "nextpnr – a portable FPGA place and route tool," https://github.com/YosysHQ/nextpnr.
[5] C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A Partial Reconfiguration Framework," in *IEEE FCCM*, 29 2012-may 1 2012, pp. 37–44.
[6] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A Tool and API for FPGA Bitstream Manipulations," in *DATE*, 2017.
[7] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch, "ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications," in *FSP*, 2018.
[8] PYNQ, "http://www.pynq.io/community.html," http://www.pynq.io.
[9] W. Xu, R. Ramanarayanan, and R. Tessier, "Adaptive Fault Recovery for Networked Reconfigurable Systems," in *FCCM*, 2003, p. 143.
[10] I. Giechaskiel et al., "Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires," in *ASIACCS*, 2018.
[11] C. Ramesh et al., "FPGA Side Channel Attacks without Physical Access," in *FCCM*, 2018.
[12] C. Beckhoff, D. Koch, and J. Torresen, "Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration," in *FPL*, Aug 2010, pp. 596–601.
[13] V. Betz and J. Rose, "VPR: a New Packing, Placement and Routing Tool for FPGA Research," in *FPL*, 1997, pp. 213–222.
[14] J. Luu et al., "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM TRETS*, vol. 7, no. 2, pp. 6:1–6:30, Jul. 2014.
[15] E. Hung, F. Eslami, and S. J. E. Wilton, "Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices," in *FCCM*, 2013, pp. 45–52.
[16] Xilinx Inc., "The Xilinx Design Language," july 2000, HTML documentation file supplied with ISE Verion 6.3.
[17] C. Lavin et al., "HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping," in *FCCM*, 2011, pp. 117–124.
[18] ——, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," in *FPL*, Sep. 2011, pp. 349–355.
[19] E. L. Horta, J. W. Lockwood, and S. T. Kofuji, "Using PARBIT to Implement Partial Run-Time Reconfigurable Systems," in *FPL*, 2002.
[20] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java Based Interface for Reconfigurable Computing," in *MAPLD*, 1999.
[21] SymbiFlow, "Project X-Ray," https://github.com/SymbiFlow/prjxray.
[22] S. Chandrakar, D. Gaitonde, and T. Bauer, "Enhancements in UltraScale CLB Architecture," in *FPGA*, 2015, pp. 108–116.