Automatic Generation of Application-Specific FPGA Overlays with RapidWright

Joel Mandebi Mbongue*, Danielle Tchuinkou Kwadjo*, Christophe Bobda*
*ECE Department, University of Florida, Gainesville Fl, USA
Email: jmandebimbongue@ufl.edu, dtchuinkoukwadjo@ufl.edu, cbobda@ece.ufl.edu

***Abstract*—In this paper, we introduce *Application-Specific FPGA Overlays* (AS-Overlays), a new approach to automatically generate overlays from high-level description language applications that can achieve bare-metal performances. Our approach is based on the automatic extraction of hardware kernels from data flow applications. Extracted kernels are then leveraged for application-specific generation of hardware accelerators. The reconfiguration of the overlay is done with RapidWright which allows to bypass the HDL design flow. Through prototyping, we demonstrated the viability and relevance of our approach. Experiments show a productivity improvement up to 20× compared to the state of the art FPGA overlays, while achieving over 1.33× higher Fmax than direct FPGA implementation and the possibility of lower resource and power consumption compared to bare metal.**

***Index Terms*—FPGA, Overlay, RapidWright, LLVM, Kernel.**

I. Introduction

Over the past decades, FPGAs have continuously matured and now contain millions of logic gates, thousands of DSP blocks, megabytes of BRAMs, and other types of resources. This development opens doors to unprecedented hardware acceleration in several computing domains such as deep learning, image and scientific processing, and cloud computing. Nevertheless, these feature improvements have not translated into widespread use of FPGAs. One reason is that designing for FPGAs remains a challenging endeavour requiring hardware expertise and long compilation time, which limits the use of FPGA accelerators to niche disciplines involving highly skilled hardware engineers.

To help addressing that limitation, High Level Synthesis (HLS) have been proposed . It focuses on high-level functionality rather than low level implementation. However, the needed hardware understanding and the high compilation times (especially placement and routing) still limit productivity and mainstream adoption. FPGA overlays have been developed to promote FPGAs to a wider user community and for increased design productivity. In general, overlays use coarse-grained processors, which can be programmed from a function call, in a 2D intercommunication infrastructure that allows parallel processing and data exchange among the processors [1][2]. The processors are homogeneous and programmable, and not tailored for specific applications. Overlays such as Hoplite [3], FLexiTASK [1], and Quattor [2] have dedicated optimization, mostly focusing on the interconnect and communication infrastructure. In the same line of work, Ishebabi et al. [4] presented methods for automatic synthesis targeting arrays of multiprocessors on chip using exact formulations such as integer linear programming of answer set programming. Unfortunately, these advantages come at the cost of area and performance, limiting overlays to relatively small to moderate applications. Indeed, FPGA overlays are usually an order of magnitude slower than bare metal implementations, and consume way more resource and power. Because the main purpose of FPGAs is hardware acceleration, overlays have therefore not been able to breakthrough.

In this work, we introduce *Application-Specific FPGA Overlays* (AS-Overlays), a novel form of FPGA overlays designed for Data Flow Applications. AS-Overlays provide the flexibility of state-of-the-art overlays on one hand and bare metal performance on the other hand. It leverage application specific architectural components for efficient bare metal implementation of functions needed by run-time applications, effectively eliminating the intermediate layers of conventional overlays. We propose an approach for automatic generation of overlay kernels from applications. The work of Cong et al. [5] is similar to ours in that they applied graph-based techniques to identify frequent patterns by analyzing graph edit distances. That work nevertheless differs from ours as patterns are detected for optimized FPGA resource sharing during the binding in behavioral synthesis. Furthermore, kernels are identified from a set of high-level programming language (HLPL) applications, with no hardware description language (HDL) generation and no usage of domain-specific language (DSL). Specifically, our contribution includes :
(1) An application-specific FPGA overlay generation flow for productivity, performance, and power consumption improvement. (2) An automatic identification of application kernels through intermediate representation inspection using the Low Level Virtual Machine (LLVM) [6], a compilation and code instrumentation framework. (3) Systematic hardware generation from identified kernels using RapidWright to shorten design cycles and generate tailored netlists [7].

II. Design Flow

The major limitation of FPGA overlays resides in that they most often feature more resources than what is actually needed, resulting in increased power consumption and performance lost. This results in architectures not optimized for specific tasks. In Figure 1, we propose a *Design Flow* to build FPGA overlays that can compete with bare metal

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT47387.2019.00053

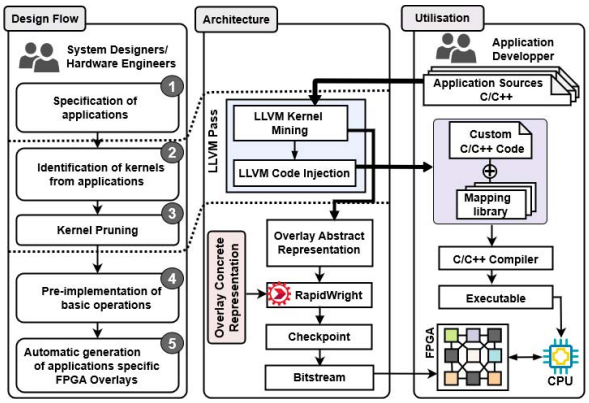


Fig. 1. Design Flow, Utilization Flow and Architecture

implementations. Few steps are necessary to produce an AS-Overlay: (1) Specify the application with a HLPL. (2) Inspect the bytecode or intermediate representation (IR) of the application at compile time to extract compute-intensive code sections that we identify as kernels. (3) Optimize kernels to remove unneeded instructions. (4) Manually pre-synthesize basic operations from the IR instruction set using vendor tools: this step is done exactly once, and the synthesized netlists can be reuse in several other applications. (5) Combine the pre-synthesized basic operations according to kernel descriptions to generate hardware circuits.

Figure 1 also illustrates the overall *Architecture* supporting the proposed design flow. After the identification of kernels, the LLVM Pass generates a new source code equivalent to the input program, in which kernels' instructions are replaced by hardware calls. We use LLVM to search for kernels as it allows transparent optimization on applications written in arbitrary HLPL. Each application is parsed with an LLVM Frontend to output an IR. The produced LLVM IR is then converted into data flow graphs for analysis, and kernels are identified. RapidWright is further leveraged to automatically generate kernel netlists by assembling as in a puzzle, a set of pre-synthesized LLVM IR operations. We use RapidWright because it is designed to quickly stitch together pre-implemented modules with minimal QoR loss. Finally, hardware kernels are embedded within PEs of an arbitrary overlay architecture, and Vivado is used to place and route the AS-Overlay. In the *Utilization* flow, a new optimized C/C++ code alongside the mapping library can now be compiled and run on a SoC. The mapping library is made of a set of functions handling data copy to/from the FPGA, removing the need for hardware expertise. In the rest of the paper, the focus is mainly set on kernel mining and hardware generation.

III. Kernel Mining

A. Background Definitions

Definition 1. A *Control Data Flow Graph* (CDFG) is a directed graph $G = (V, E, L, l)$, where V represents the set of vertices, $E \subseteq V \times V$ the set of edges, and L the set of labels, with $l : V \cup E \rightarrow L$ being the labeling of vertices and edges. In the context of this work, a *graph* is a CDFG defined at a basic block (BB) level, vertexes are operations and edges display the order of precedence between operation.

Definition 2. Given a set of CDFGs $GS = \{g_i | i = 1..n\}$ and a threshold α (in this case the minimum support value), the *kernel mining* consists in finding graphs (i.e kernels) g in GS, such that $support(g) >= \alpha$.

B. Kernel Mining

The kernel mining follows several steps among which: (1) *Generate candidates* using a Depth First Search-based approach, (2) *Prune the candidates* to remove infrequent vertices and edges, (3) *Evaluate the support value* to decide whether a candidate is a kernel or not.

The isomorphism search during candidate pruning stage is known to be NP-complete, and several subgraphs isomorphism techniques as the ones described in [8] lead to high computation overhead. One way to mitigate that high overhead consists in computing the canonical form of graphs [9]: if the canonical form of two graphs is identical, the graphs are considered isomorphic. We therefore construct canonical form of DFS codes as in [10], which the minimum code that can be derived from a graph g. Specifically, the strategy consists in:

(1) Building frequent subgraphs bottom-up, using DFS code as regularized representation.
(2) Eliminating redundancies via minimal canonical DFS code based on lexicographic ordering [10].

From DFS code tree built with lexicographic ordering, we can make the following assumptions: (1) If a DFS code γ is frequent, then every ancestor of γ is frequent. (2) If γ is not frequent then every descendant of γ is not frequent. Finally, the custom C/C++ code is generated by replacing kernels' instructions by high-level functions for hardware acceleration. Original names of the variables and their location are retrieved by inspecting the debug metadata attached to each instruction in the IR. LLVM uses the DWARF standardized debugging data format like several other compilers and debuggers to support source layer debugging.

IV. Hardware Generation of Kernels

Initially, the function implemented by a PE in the overlay layout is defined as a black-box. We leverage the pre-implemented design flow of RapidWright [7] to produce netlists from kernels previously identified with LLVM. The first step consists in synthesizing basic operations from LLVM IR out-of-context (OOC) with Vivado to create a *library of Modules*. Modules are built OOC to ensure that I/O buffers and global clocks are not inserted into kernel netlists [11]. This stage implies a manual implementation (through HDL or HLS) of operations to combine into kernels by an engineer, with the advantage that this step is done once. In the following stage, an application built on the RapidWright API stitches pre-implemented modules following the LLVM kernel DFG descriptions. The hardware kernels thus generated are returned as design checkpoint and define the functions to be executed in

PEs. Finally, the RapidWright application opens the netlist of the overlay (EDIF or DCP files), browses through the design cells, and reads-in kernel DCPs into PE black-boxes. The placement and routing are run with Vivado, and a bitstream of the overlay is produced for FPGA deployment.

A. Datapath Regularization

To reduce overall latency and data management overhead, datapaths must be regularized. Each operation within a kernel comes with its own latency in number of clock cycles. We must therefore ensure that operands arrive at the boundary of each module at the same time to expect correct results. This task is done by the RapidWright application which inserts FFs on the path. Inserting FFs do not increase overall latency as the number of FFs is the cumulative latency of operations on the datapath.

B. Processing Elements

We do not discuss interconnection topology between PEs as the focus is not on obtaining improved/flexible communication; rather, we emphasize architectural features supporting the automatic generation of hardware kernels. In addition, the proposed AS-Overlay generation flow is designed and well suited for any interconnect topology (mesh, torus, mixed topology, etc [12]) as only the PE processing core will be changed. We therefore look at the minimum architecture set-up that should be embedded in each PE. Figure 2 illustrates the architecture of PEs. To handle kernels of multiple inputs/outputs, the I/O buses have parameterizable sizes and are split into 32-bits channels. Inputs and outputs are temporarily stored in I/O queues to avoid data lost in case of multiple clock domains crossing. The *Control* module is configured with the latency of the kernel programmed in the PE, allowing to orchestrate when fetching data from input queues, and when writing results into output queues. The *Black-box* is the core of the PE as it implements one or multiple kernels derived from LLVM code inspection.

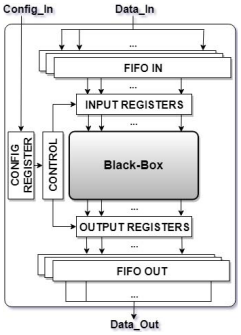


Fig. 2. PE Architecture

V. Experimental Observations

A. Evaluation Platform and Setup

Designs are implemented on a Xilinx Kintex UltraScale+ FPGA (xcku5p-ffvd900-2-i). Hardware generation is conducted with Vivado HLx Editions v2018.2 and RapidWright v2018.2.5-beta. We study: (1) global latency, (2) Fmax and productivity, (3) resource utilization, and (4) power consumption, when comparing AS-overlays to regular overlays and bare metal implementations. For each testing application, we design a 3×3 PEs overlay with three flavors: ***(i) Bare Metal***: functions are implemented in HDL. ***(ii) Regular Overlay***: Each PE implements a dozen arithmetic and logic operations, and is implemented in HDL. ***(iii) AS-Overlay***: Kernels are implemented into PEs using the design flow described in Figure 1.

B. Evaluation Results

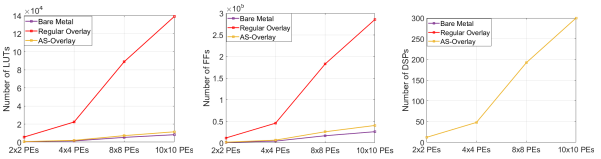
Execution times recorded in Table I come from placing the bare metal, regular overlay, and AS-Overlay implementations of each application alongside a MicroBlaze CPU with a 300MHz global clock. It shows that AS-Overlays can effectively compete with bare metal implementations. The bare metal outperforms AS-Overlays on the image smoothing and matrix multiplication because of the additional clock cycles introduced by the RapidWright application. In fact, to ensure timing closure when integrating kernels within the AS-Overlay fixed sections (PE architecture + interconnect), FFs are injected on the datapath after each operation. Table I also shows that AS-Overlays compute faster that regular overlays when clocked with identical frequency. To carry out Fmax and productivity studies summarized in Table II, we introduced a Phase-Locked Loop generating a 300MHz clock on each flavor of overlay. The idea was to observe the maximum frequency and how long would the compilation take. As first observation, AS-Overlays can achieve up to $1.47\times$ improved Fmax compared to regular overlays on tested applications (the AS-Overlay tops at 447MHz while the regular overlay caps at 304MHz). This is caused by general-purpose ALUs of regular overlays that contain several muxes introducing substantial delays on datapaths. On the other hand, bare metal implementations achieved higher Fmax compared to AS-Overlays on outer product and Robert cross filter. It comes down to an observation made in [7]: vendor tools such as Vivado often produce high performance results for small modules of a design. In this case of figure, outer product and Robert cross are respectively a set of independent multiplications, and subtractions followed by comparisons, which gives to bare metal a $1.09\times$ Fmax advantage over AS-Overlays. That advantage is nevertheless lost on more complex functions such as image smoothing (the AS-Overlay achieved a $1.33\times$ higher Fmax), which computes the average of adjacent pixels, highlighting the benefits of using the RapidWright pre-implemented flow as smaller modules can be pre-implemented to achieve maximum frequency, and later be assembled with minimal QoR loss.

Figure 3 summarizes the utilization of FPGA resources only for the matrix multiplication (because of page limitation, it was not possible to present the same study for each of the tested applications) as an illustration of how the fabric is progressively occupied as the number of PEs is scaled up. In general, the total amount of resources used by AS-Overlays is close to that of the bare metal, and both far below regular overlays. Figure 3a nevertheless displays the same number of DSPs (the purple, red, and yellow lines are superimposed, so only the yellow line is visible) simply because PEs on the three platforms implement only one multiplier using 4 DSP48E2s.

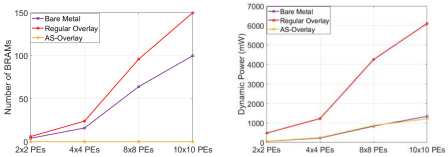
Pre-implementing basic functions from LLVM IR also have the potentiality of reducing resource utilization as illustrated

TABLE I
EXECUTION TIME COMPARISON ON 3×3 PES IN μ S

Matrix Mult				Outer Product				Robert Cross				Smoothing			
Size	**Bare Metal**	**Regular Overlay**	**AS Overlay**	**Size**	**Bare Metal**	**Regular Overlay**	**AS Overlay**	**Image Size**	**Bare Metal**	**Regular Overlay**	**AS Overlay**	**Image Size**	**Bare Metal**	**Regular Overlay**	**AS Overlay**
8×8	0.39	1.71	0.58	8×8	0.043	0.046	0.043	16×16	0.19	0.58	0.19	16×16	3.48	4.35	4.15
16×16	3.05	13.65	4.57	16×16	0.113	0.116	0.113	32×32	0.756	2.28	0.756	32×32	13.68	17.1	16.3
32×32	24.29	109.32	36.42	32×32	0.396	0.400	0.396	64×64	3.03	9.12	3.03	64×64	54.72	68.40	65.36
64×64	194.20	873.84	291.29	64×64	1.53	1.54	1.53	128×128	12.13	36.42	12.13	128×128	218.52	273.15	261.01



(a) Number of Look-Up Tables, Number of Flip-Flops, Number of DSP Blocks



(b) Number of Block RAMs, Power Consumption

Fig. 3. FPGA Resource Utilization

in Figure 3b: Vivado optimizes individual hardware implementation from LLVM IR without BRAM insertion while adding such resources when compiling bare metal and regular overlays. Figure 3a reports a higher utilization of FFs in AS-Overlays as opposed to bare metal. This is due to FFs insertion during datapath regularization in addition to input, config and output registers from the PE architecture (check Figure 2). While injecting FFs on the datapath as explained in section IV-A do not incur delays in kernel execution, it obviously increases the number of FFs used depending of the structure of kernel DFGs.

TABLE II
PRODUCTIVITY ANALYSIS & MAXIMUM FREQUENCY

3×3 Overlay Generation Flow		Applications			
		Matrix Mult	**Outer Product**	**Robert Cross**	**Smoothing**
Bare Metal	*Synthesis*	26	17	35	35
	Optimization	5	2	6	34
	Placement	28	23	44	85
	Routing	68	59	55	1064
	Total (Seconds)	**127**	**101**	**140**	**1218**
	Fmax (MHz)	**365**	**488**	**348**	**231**
Regular	**Vivado Flow →**	***Synth.***	***Opt.***	***Place.***	***Routing***
		40	27	111	1457
	Total (Seconds)	**1635 (27 minutes 15 seconds)**			
	Fmax (MHz)	**304**			
AS-Overlay	*Kernel Gen.*	3.89	3.48	3.55	4.34
	Kernel Load.	2.16	2.07	2.13	2.05
	Optimization	5	4	3	33
	Placement	46	19	24	83
	Routing	65	54	117	646
	Total (Seconds)	**122.05**	**82.55**	**149.68**	**768.39**
	Fmax (MHz)	**435**	**447**	**318**	**308**

VI. CONCLUSION

In this paper, we presented an approach aiming the automatic generation of Application-Specific FPGA Overlays for data flow applications capable of providing bare metal performances. The approach extracts kernels from applications at compile time, and automatically builds accelerators tailored for the application needs. Experimental evaluations demonstrated the viability of our approach with significant productivity improvement, power consumption reduction, and lower execution time over regular FPGA overlays. Future work will investigate the replicability feature of RapidWright coupled with LLVM code instrumentation to build more efficient FPGA accelerators.

ACKNOWLEDGEMENT

This work was partially supported by the ONR under the Grant CCN 0402-17643-21-0000, and the Air Force Research Lab AFRL/RIGA Cyber Assurance Branch, Rome NY.

REFERENCES

[1] J. Mandebi Mbongue, D. Tchuinkou Kwadjo, and C. Bobda, "Flexitask: A flexible fpga overlay for efficient multitasking," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. ACM, 2018, pp. 483–486.

[2] M. Metzner, J. A. Lizarraga, and C. Bobda, "Architecture virtualization for run-time hardware multithreading on field programmable gate arrays," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Springer International Publishing, 2015, pp. 167–178.

[3] N. Kapre and J. Gray, "Hoplite: Building austere overlay nocs for fpgas," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2015, pp. 1–8.

[4] H. Ishebabi and C. Bobda, "Automated architecture synthesis for parallel programs on fpga multiprocessor systems," *Microprocessors and Microsystems*, vol. 33, no. 1, pp. 63 – 71, 2009.

[5] J. Cong and W. Jiang, "Pattern-based behavior synthesis for fpga resource reduction," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. ACM, 2008, pp. 107–116.

[6] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[7] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 133–140.

[8] N. S. Ketkar, L. B. Holder, and D. J. Cook, "Subdue: Compression-based frequent pattern discovery in graph data," in *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*. ACM, 2005, pp. 71–76.

[9] T. Miyazaki, "The complexity of mckay's canonical labeling algorithm," in *Groups and Computation II*, vol. 28. Aer. Math. Soc.: Providence, RI, 1997, pp. 239–256.

[10] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 721–724.

[11] Xilinx, "Vivado design suite user guide hierarchical design," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug905-vivado-hierarchical-design.pdf, 2017.

[12] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 1, 2006.