Optimizing FPGA-based Streaming Applications for Throughput Using Pipelining

Ali Asghar, Rick van Loo, Timon Kruiper, Daniel Ziener
Computer Architecture for Embedded Systems, University of Twente
{ali.asghar, d.m.ziener}@utwente.nl
{r.vanloo, t.r.kruiper}@student.utwente.nl

***Abstract*—In this paper, we present an automated flow for insertion of pipeline stages in FPGA-based streaming applications in order to increase the throughput. The proposed approach involves the utilization of Xilinx's Automated Pipeline Analysis tool to estimate the number of pipeline stages, while the Rapid-Wright framework incorporate these stages into a synthesized design. The Vivado Design Suite is then used to place and route the modified netlist. Furthermore, a recycling approach has also been proposed to reduce excess registers. The results show a significant improvement in the maximum operating frequency for designs without any sequential loops (~51%) with a moderate resource overhead, while slight gains (~12%) were also observed for designs containing feedback loops.**

***Index Terms*—FPGA Architecture, FPGA CAD, Pipelining**

I. Introduction

Low clock frequencies are one of the drawbacks of the FPGA technology. According to [1], the clock frequency of FPGA designs increased only by 3.8× between 1999 and 2009. In the same time period, the CPU frequencies increased more than 9× from around 400 MHz to 3.6 GHz. For the 90 nm technology, compared to an ASIC, FPGA designs operate at 3 to 4× lower clock frequency [2]. With current technologies (CPU clock frequencies up to 4 – 5 GHz) and current FPGA series, the factor is much higher. This inefficiency results from the excessive flexibility of FPGAs, which requires a lot of reconfigurable multiplexers in the combinational logic.

On the other hand, the logic density inside the FPGAs has increased tremendously, owing to the progressive decrease in transistor size. Since the invention of the FPGA, the logic density has grown by a factor 10,000, whereas the maximum clock frequency has only increased by 100x [3]. This development has prompted researchers to explore the area-delay tradeoffs in FPGAs. Since, logical resources are usually not the limiting factor in the implementation of designs, there is always a possibility to compromise area in favor of higher clock frequencies. The migration of commercial FPGA architectures from 4 input LUT to 6 input LUT was prompted by the better performance of 6 input LUT in terms of higher frequencies [4], despite they introduce a heavy penalty in terms of area efficiency. Furthermore, increase in the number of heterogeneous blocks in commercial architectures, clearly suggests that designers are willing to compromise area in favor of more speed.

The efficiency of FPGA resources (lookup tables) can be increased by using designs with many pipeline stages combined with a very high clock frequency. Pipelining increases the throughput, at the cost of increased register utilization. To achieve better pipelining, Xilinx, e.g., increased the number of flip flops in the FPGA fabric by a factor of two during the introduction the 6 input LUTs [5].

Since, a higher throughput means more processed data per unit time, pipelining is beneficial especially for streaming applications, where the output is independent of the incoming data stream. However, for applications which do not process data continuously, pipelining the design may have an adverse effect of increased latency. Therefore, for efficient pipelining, latency vs. performance trade-off is of critical importance.

In this paper, we introduce a novel automated design flow to increase the throughput of streaming application by automatically inserting pipeline stages after synthesis in the netlist at the logic level. The changes on the design are done without touching the HDL sources. Compared to manual pipelining which involves modifications at the RTL level and a re-run of the flow after every modification, our approach saves a significant amount of time and effort.

Our approach utilizes the *Pipeline Analysis Tool* [6] from Xilinx, to explore the possibility of pipelining designs. In contrast to the Xilinx tool, we optimize the results by merging the newly inserted registers, for the paths which share the same start and end points. This so called register recycling reduces significantly the amount of inserted registers. The optimized results are then implemented by inserting the pipeline registers using *RapidWright* [7]. *RapidWright* is an open-source, Java-based framework from Xilinx, which allows user to access lower level architecture details and make netlist level manipulations, using high-level Java programming.

The remainder of this paper is organized as follows: In Section 2, we provide background details and some prior research work related to pipelining in FPGAs. In Section 3, we introduce our concept. The implementation details are presented in Section 4, and Section 5 provide a discussion of results. Section 6, concludes the paper along with some proposed future work.

II. Background and Previous Work

The idea of automatic pipelining was first introduced by Leiserson et al. [8] with an approach called *retiming*. The retiming is an optimization technique, which attempts to replace the existing registers in a design to achieve certain optimization goals (e.g., reducing the delay of the critical path). This optimization can thus improve the throughput of a system while keeping the initial latency constant (as the number of registers around every cycle remains constant). However, if target is to reach a certain frequency, then simply retiming

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT47387.2019.00065

the design may prove insufficient, as the improvements are capped by the number of pipeline stages already present. To address this issue, two modifications have been proposed in [9], *Repipelining* and *C-slow retiming*. The Repipelining technique adds additional pipeline stages to the design at the cost of added latency. The modified design is then retimed to achieve the desired operating frequency. However, lack of accuracy in the delay model used in [9] deteriorates the quality of results.

A. Pipeline Analysis Tool

The Vivado *Pipeline Analysis Tool* [6] builds upon the ideas presented in [8] and [9]. By using the timing models implemented in Vivado, an existing design is analyzed for critical paths, extra pipeline stages are suggested, and a report is generated. However, at the time of writing this document, the automatic pipelining tool included in Vivado can only generate a report, listing the registers which can be added for pipelining the design. For the implementation of pipeline stages, two methods have been proposed in [6]:

1) Using the generated report to implement all the calculated pipeline stages in one step, followed by place and route. The authors note that the approach gives a reasonable estimation, but there is no information on the effect of individual pipeline stages and their placement on timing.
2) An iterative approach, which inserts a specified number of pipeline stages at a time, performs placement, updates timing, and generates a new pipeline report. The procedure repeats till there is no further improvement, after which the design in routed.

The algorithm presented in [6] can only pipeline feed-forward paths. Since, inserting pipeline stages into a sequential loop introduces errors in the functionality, all paths which contain loops are not considered for pipelining. However, designs with feedback loops can also be optimized (to a certain degree) by pipelining the feed-forward path till the loop performance limit is reached.
Unlike Xilinx, other automatic pipeline tools such as [10] and [11], perform pipeline estimations during synthesis which offer less precision at a lower computational cost.

B. RapidWright

As mentioned in the previous section, the registers specified in the pipeline report have to be added manually into the netlist by the designer. To automate this process, we make use of *RapidWright*, an open-source Java-based programming environment, which enables users to make fine grained optimizations to their designs. With *RapidWright*, designers can import a *Design Checkpoint* (.dcp) file from Vivado and make the desired changes, after which the modified design can be exported back to Vivado. *RapidWright* also allows users to implement their own custom place and route tools [12] [13], augmenting/bypassing the place and route stages of Vivado.

III. Concept

Our technique for automatic pipelining is an iterative approach (see Figure 1). For every iteration, we a) place the design using Vivado and b) parse a pipeline analysis report generated by the Vivado built-in *Pipeline Analysis Tool* [6]. We c) optimize the results by using our register recycling approach implemented in *RapidWright*, d) insert the remaining pipeline registers with *RapidWright*, and e) invoke the Vivado place and route. The iteration loop breaks if either the achieved clock frequency $F_{\text{max,act}}$ is equal or higher than the ideal clock frequency $F_{\text{max,ideal}}$ from the pipelining report from b) or the maximum number of iterations N_{iter} is reached.

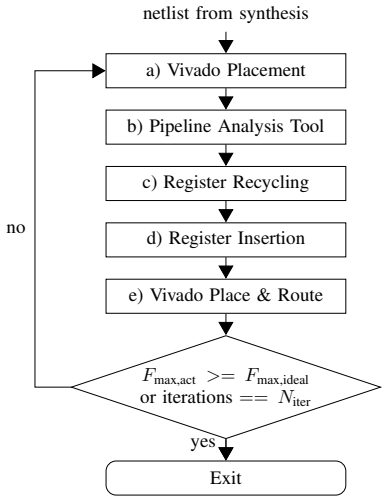


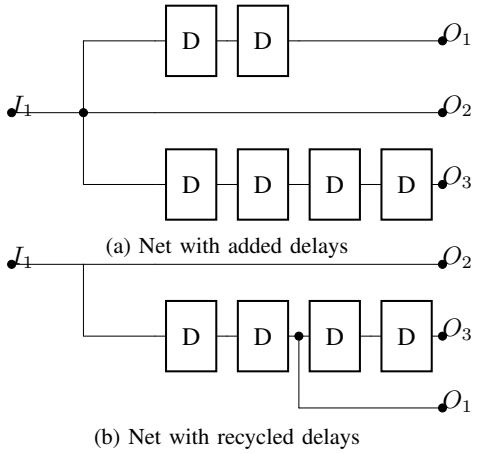
Fig. 1: Iterative approach for automatic insertion of pipeline registers. The loop breaks if the current clock frequency $F_{\text{max,act}}$ is equal or more $F_{\text{max,ideal}}$ or if maximum number of iterations N_{iter} is reached.

A. Pipeline Analysis Tool

The results reported in [6] are from the iterative approach, which according to authors produces better results due to the availability of accurate timing data. Therefore, we have also adopted the idea of iterations for the insertion of pipeline stages. The input to the tool is the currently placed netlist, which could either be coming from the synthesis for the first iteration, or the result of previous iteration. The tool then generates a report which indicates the theoretical maximum clock frequency $F_{\text{max,ideal}}$ and a list with suggestions on how much and where to add additional pipeline registers. $F_{\text{max,ideal}}$ is determined by the lowest maximum frequency of either the loops within the design or DSP slices [6].

B. Recycling

A closer examination of the pipeline report reveals that multiple paths could have the same start and different end points, although such paths are different, the net connecting them would be the same. As a consequence, excess register would be inserted in the design. This situation is depicted in Figure 2b. Two registers would be added before the end point O1 and four before point O3, resulting in six added flip flops. An alternative would be to *recycle* the registers as shown in Figure 2b. The resulting design has same functionality and timing as the one in Figure 2b, but requires lesser registers.



(a) Net with added delays

(b) Net with recycled delays

Fig. 2: Recycling to reduce excess registers

IV. Implementation

Our proposed approach is implemented with a *tcl* script which processes the different steps in Figure 1 by invoking Vivado *tcl* commands and our Java program using *RapidWright*. After generating the checkpoint, Vivado runs this script which takes three arguments as input parameters: design checkpoint, *number of iterations* (N_{iter}), and *maximum latency per loop* (L). The *number of iterations* specify the times the iterative approach will be run, while the *maximum latency per loop* specifies the number of pipeline stages inserted per iteration. The pipeline report for a design can be generated in Vivado using the *tcl* command: *report_pipeline_analysis*, the report contains all the relevant information to add pipeline stages. The report and a synthesized *design checkpoint* (.dcp) are then exported to *RapidWright*. *RapidWright* uses a simple API to import the checkpoint file, named *Design.ReadCheckpoint* and the pipeline report is parsed using the classes: *IntraClockSummary*, *PipelineSummary* and *CriticalLoops*, which corresponds to report sections *Intra-Clock Summary* and *Paths to Pipeline*.

The *Intra-Clock Summary* indicates the changes in $F_{\text{max,ideal}}$ after the insertion of each pipeline stage. For implementation purpose, only the first and the last rows are of interest as shown in Table I. The first row showing the actual $F_{\text{max,ideal}}$, while the last row depicting the theoretical $F_{\text{max,ideal}}$ after pipelining. The *Paths to Pipeline* section (shown in Table II) contains instructions to pipeline the design. For every flip flop there is a start and an end point, corresponding to the path in which the flip flops have to be inserted.

Once all the data is parsed, the Java program executes the following steps:

1) Searches the netlist for the VCC, GND and CLK nets which are used to connect to the flip flop.
2) Checks whether the design contains any loops:
   a) If there are no loops, keep on adding registers.
   b) If there are loops, check whether the maximum frequency is reached and quit the program if the maximum frequency is reached.
3) Insert the registers from the pipeline report into a *HashMap*, where the key is the *StartPoint*, and the value is a list of *EndPoints* and then sort the list on the highest amount of delay.
4) Loop through the *HashMap* and do the following for every entry:
   a) Take the output port of the *StartPoint* and loop through the *EndPoints* list.
   b) Add a flip flop before every *EndPoint* and connect the input to the *StartPoint*.
5) Export the design as a Vivado checkpoint.

V. Experimentation Setup and Results

To evaluate the performance of our automated pipeline insertion flow, we have performed experimentation on five benchmark circuits, namely, SHA1, DES, AES (pipelined), and FIR which were obtained from [15], [16], [17], and [18] respectively. For AES (sequential), we used our own round-based implementation. The designs AES (pipelined) and FIR filter are without any feedback loops, while, AES (sequential), DES, and SHA1 are all sequential. According to [6], highly pipelined designs map well on UltraScale and UltraScale+ architectures [14]. Therefore, all the designs have been implemented on a ZCU102 board, featuring a Zynq UltraScale+ device. The synthesized design checkpoint files for *RapidWright* have been generated using Vivado 2018.3.

To explore the effect of pipelining on the benchmark circuits, we set the number of iterations $N_{\text{iter}} = 10$ and $L = 1$. From our observations, setting $N_{\text{iter}} = 10$, results in designs with $F_{\text{max,ideal}}$ very close to the theoretical value. Furthermore, with a lower value of N_{iter}, the runtime overhead is also reasonable.

Table III, shows the effect of pipelining on the throughput of the benchmark circuits. As expected, the designs with feedback loops (AES and DES) showed little improvement (~2%) in throughput, with the exception of SHA1, which showed a significant increase of ~12%. The run-time penalty for all the designs was modest, with the highest value recorded for DES (292 seconds). However, an interesting observation was made for the designs without feedback loops. The report generated by the pipeline tool, showed that both AES (pipelined) and FIR filter are already operating at or very close to $F_{\text{max,ideal}}$. Since, $F_{\text{max,act}} >= F_{\text{max,ideal}}$ results in the iterative approach to terminate, no pipeline stages were inserted in the AES (pipelined) design, while, for FIR filter throughput was improved by only ~11%. This effect could be attributed to the size mismatch between the benchmarks and the target architecture. The abundance of resources in UltraScale+ devices makes it easier to achieve $F_{\text{max,ideal}}$ for designs like AES or FIR filter with a relatively low (~4.24%, and ~0.04% respectively) utilization of resources. This idea was validated by mapping the same designs on a smaller Zedboard device [19]. When mapped on a Zedboard, the throughput for AES (pipelined) and FIR filter improved by (~28%) and (~51%) respectively.

The results for the impact of pipelining on the resource utilization are also depicted in Table III. The terms *'original'* and *'pipe'* refer to the resource usage before and after the

Clock	Added Latency	Ideal Fmax	Ideal Delay	Requirement	WNS	Added Pipeline Reg	Total Pipeline Reg	Pipeline Insertion Startpoint	Pipeline Insertion Endpoint
CLK	0	325.99 MHz	3.068 ns	10 ns	6.932 ns	n/a	0	state_out[0]_i_1/O	r1/state_out_reg[0]/D
CLK	13	651.98 MHz	1.534 ns	10 ns	8.133 ns	958	15060	out_1[0]_i_1/O	state_out[0]_i_1/I3

TABLE I: Example first and last line of the Intra-Clock Summary

Clock/PathGroup	Path Cut	Added Pipeline Registers	Startpoint	Endpoint
CLK/CLK	0	4	out_1[88]_i_1__3/O	state_out[88]_i_1__3/I4
CLK/CLK	1	6	out_1[88]_i_1__4/O	state_out[88]_i_1__4/I4

TABLE II: Example of two 'instructions' of 'Paths to pipeline'

Design	$F_{max,ideal}$ (MHz)	$F_{max,act}$ (MHz)	$F_{max,ideal,pipe}$ (MHz)	Improvement (%)	Runtime (sec)	#CLBs (original)	#CLBs (pipe)	#Registers (original)	#Registers (pipe)	#Recycled Registers
SHA1	475.66	416.26	466.12	11.97	290	172	205	1001	1453	6422
AES (seq)	686.17	675.51	686.17	1.57	170	51	58	261	294	1103
DES	787.8	770.21	787.8	2.28	292	1156	1350	6008	7221	964
AES (pipe)	795.32	795.32	N/A	N/A	N/A	2184	2184	7882	7882	0
FIR Filter	594.05	534.09	594.05	11.22	281	24	28	42	79	154

TABLE III: Runtime and Percentage Improvement in $F_{max,ideal}$ after pipelining

pipelining. The highest increase in the register utilization was observed for DES (~20.2%), with an extra 1213 registers used for pipelining. However due to the increased flip-flop to LUT ratio of newer FPGA families, the increase of used CLBs is only marginal. Also, from the results in Table 2, the benefits of the recycling approach can be clearly seen with a considerable amount of registers saved. For SHA1, the pipeline report suggests inserting 6874 registers, however, with recycling only 452 registers were required to pipeline the design. The recycling technique shows a lot of improvement in terms of resource usage. While the throughput stays the same, the recycled version uses fewe registers and slices.

VI. Conclusions

This work presents a novel automated pipeline insertion flow for streaming-based applications. The run time of several seconds to few minutes of our approach is negligible compared to manual pipelining. Furthermore, the HDL sources do not have to be modified. Our technique builds upon the work presented in [6], extended with a sophisticated recycling approach and using *RapidWright* to implement the design modifications. The use of *RapidWright* provides access to lower-level architecture details, making it possible to add pipeline stages. The results show slight improvement in throughput (upto ~12%) only for sequential designs on an UltraScale+ device, while significant gains (upto ~51%) were observed for designs without feedback loop, when mapped to a low-end Zedboard device. Hence, to validate our approach, a comprehensive experimentation needs to be performed with large scale designs without any feedback loops, exhibiting greater potential for pipelining. Furthermore, compared to [6], the introduced recycling approach significantly reduces the number of registers required to pipeline the designs.

VII. Acknowledgement

This work has been supported by the German Federal Ministry for Eduction and Research (BMBF) within the collaborative research project SecRec (16KIS0609).

References

[1] P. Sundararajan, High performance computing using fpgas, Xilinx white paper: FPGAs, pp. 115, 2010.
[2] I. Kuon and J. Rose, Measuring the gap between fpgas and asics, IEEE Transactions on computer-aided design of integrated circuits and systems, vol. 26, no. 2, pp. 203215, 2007.
[3] Steven Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," Proceedings of the IEEE 103 (2015): 318-331.
[4] Elias Ahmed, and Jonathan Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," FPGA (2000).
[5] Xilinx Inc., Virtex-5 Family Overview, August, 2001.
[6] Ilya Ganusov, et al. "Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs," 2016 26th International Conference on Field Programmable Logic and Applications (FPL) (2016): 1-10.
[7] Chris Lavin, and Alireza Kaviani. "RapidWright: Enabling Custom Crafted Implementations for FPGAs," 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (2018): 133-140.
[8] Charles E Leiserson and James B Saxe. "Retiming synchronous circuitry," Algorithmica, 6(1-6):535, 1991.
[9] Nicholas Weaver. "Retiming, repipelining and c-slow retiming," In Reconfigurable Computing, pages 383399. Elsevier, 2008.
[10] A. J. Chung K. Cobden M. Jervis M. Langhammer B. Pasca "Tools and techniques for efficient high-level system design on FPGAs," First International Workshop on FPGAs for Software Programmers 2014.
[11] Matei Istoan, and Florent De Dinechin. Automating the pipeline of arithmetic datapaths, Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017 (2017): 704-709.
[12] Leo Liu, Jay Weng, Nachiket Kapre, "RapidRoute: Fast Assembly of Communication Structures for FPGA Overlays," Field-Programmable Custom Computing Machines (FCCM) 2019 IEEE 27th Annual International Symposium on, pp. 61-64, 2019.
[13] Matthew J. Cannon et al., "Strategies for Removing Common Mode Failures From TMR Designs Deployed on SRAM FPGAs," Nuclear Science IEEE Transactions on, vol. 66, no. 1, pp. 207-215, 2019.
[14] Xilinx. UltraScale Architecture and Product Overview. Xilinx, ds890 (v3.9) edition, June 27, 2019.
[15] Available Online at: https://opencores.org/projects/sha_core
[16] Available Online at: https://github.com/freecores/des
[17] Subhasis Das. Fully pipelined AES core. Available Online at: https://github.com/freecores/aes_pipe,2009.
[18] Scott Larson. Digi-key: Fir filter (vhdl). Available Online at: www.digikey.com/eewiki/pages/viewpage.action?pageId=78086825
[19] Avnet. ZedBoard Product Briefs, 2018.