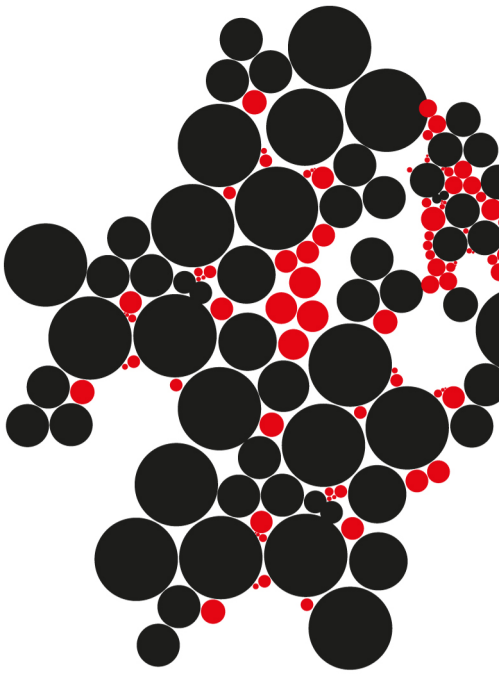
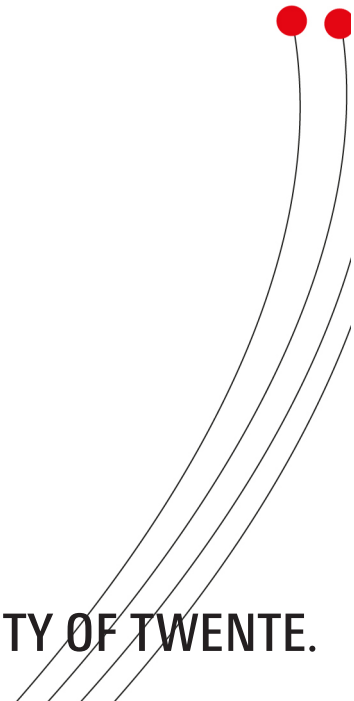
INCREASING THROUGHPUT OF FPGA-BASED STREAMING APPLICATIONS
BY USING PIPELINING



Rick van Loo
B.Sc. Thesis
November 2018

Faculty of Electrical Engineering
Computer Architecture for Embedded Systems

Supervisors:
dr. ing. D.M. Ziener
dr. ir. A.B.J. Kokkeler
dr.ir. M.S. Oude Alink



UNIVERSITY OF TWENTE.

NOVEMBER 19, 2018

Thesis description

The utilization of the used FPGA resources in time is rather low compared to an ASIC due to the lower clock frequency. The efficiency of FPGA resources (lookup tables) can be increased by using designs with many pipeline stages combined with a very high clock frequency.

In this work, the automatic insertion of pipeline stages in data paths after synthesis and the usage of high clock frequencies should be investigated. The overall goal is to increase the efficiency and performance of FPGAs. The data paths of critical parts of a design should be analyzed and additional pipeline stages should be automatic inserted.

The following issues should be solved:

- Get familiar with the FPGA design flow and architecture.
- Literature research about pipelining of FPGA-based implementations.
- Develop a concept for automatically pipelining a given hardware design.
- Implement the above mentioned concept.
- Evaluate the increase in throughput and clock frequency as well as the resource overhead.
- Writing the thesis.

Contents

1 Introduction

In the past years, FPGAs have been increasingly getting more popular due to the lowering of costs. In many applications that require highly parallelizable work and streaming applications, an FPGA is a good choice to improve performance. Designing an FPGA requires good understanding of digital logic and the design aspects that go into implementing a design from start to finish. Various attempts are being made to simplify this process by extra tooling provided by FPGA suppliers that automate processes of the design flow.

Various optimizations are being implemented in IDEs that should further improve the requirements that the designer needs. One of these improvements is performance or throughput. With streaming applications that continuously process data, throughput is the most interesting design aspect as it will define the performance of the application. More throughput means that more data can be processed in the same instant of time. Another design aspect is latency. This is the time that is needed to process the data. With a continuous stream of data this is less of an interesting design aspect but for an application that doesn't continuously receive data, this could be a big concern.

One of the most effective ways to improve the performance of FPGA applications is by adding extra pipeline stages. FPGA designs have a synchronous nature consisting of delay elements and logic, which means that it highly benefits from extra pipeline stages. For a designer, this is in practice a very tedious and error-prone manual process where the designer has to find critical paths in a design that can be pipelined. By balancing the stages and simulating the design the designer has to ensure that both performance increase has been reached, but also preserve the functionality of the design.

In this thesis, the various methods of pipelining a design are being looked into. An introduction is made to the various topics that are necessary for understanding the subject. An implementation is being made using the third-party tool RapidSmith2 to automatically pipeline an existing streaming application. In the evaluation is looked into the performance increase and the extra overhead in the design.

2 Theory and background

This chapter will go through the necessary theory and background needed as a basis for understanding the more in-depth methods and implementation described in the following chapters. General concepts such as the FPGA Design Flow, Pipelining and used tools will be described.

2.1 FPGA Design Flow

The FPGA is an integrated circuit that can be programmed after being deployed on a board, using various programming languages called Hardware Description Languages (HDL). The two mainly used HDLs being Verilog and VHDL. These languages are very verbose and strictly typed but give the designer more in-depth and low-level control over their intended design. After the designer is done programming, the programmed blueprint goes into the FPGA Design Flow. This flow is a general step-by-step approach that is not specific to a vendor. The two biggest vendors are Altera and Xilinx with their own IDE tools, Quartus and Vivado respectively, to guide the hardware designer with implementing their design.

The first step, called 'Synthesis', is to convert the HDL code to an implementable digital design. The result is a netlist containing basic elements such as lookup-tables and flip-flops. Afterwards in the 'placing' stage, these elements get an assigned physical location on the FPGA. In the 'routing' stage all these physical elements get properly connected to each other. At last, a bitstream will be generated that can be programmed onto the FPGA. Between different steps of the design flow, the FPGA design can be simulated. Before synthesis, a behavioral simulation can be done to check if the HDL code has the intended functionality. This can also be done Post-Synthesis and after Placing and Routing. At this point timing analysis and simulation can be done as well to check if the design meets the timing constraints. A more general view of the flow can be seen in Figure 1.

2.2 Sequential Logic and Pipelining

After the design went through the Synthesis, the resulting netlist consist of basic FPGA elements. Combinational logic such as the default set of AND- and OR- gates can all be implemented in Lookup tables (LUT). In this case the LUT functions as a truth table to implement multiple digital logic structures depending on the amount of inputs. Sequential logic is implemented using flip-flops (also called registers). This is a basic storage element that can latch the input on basis of a clock. The combination of logic elements and flip-flops make the design synchronous, where every next clock cycle all the latches will keep the state of the input for a full clock cycle. In general, this means how higher the clock frequency, how faster the circuit processes data. This improves the throughput

of the design, so it would be interesting to look through options that can increase clock frequency.

Unfortunately, it is not possible to increment the clock frequency without limit. Both physical and design limits will bound the maximum frequency f_{Max} that is achievable. This thesis will only focus on the frequency limit imposed on by the design.

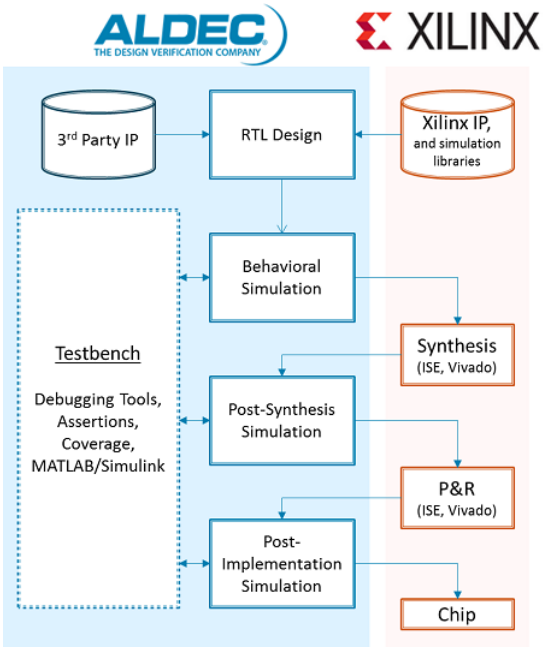


Figure 1: The Xilinx FPGA Design flow [1]

2.2.1 Design limits

All non-sequential elements such as the LUTs, combined with the propagation delay due to the routing between the elements, will introduce a delay. In a sequential design, the delay between two latches will ultimately limit f_{Max}. An example of such a model can be seen in Figure 2. A LUT, for example, needs time before it can produce an output from its input signals, this is the logic delay. The route delay is also of interest; short routes will produce the shortest delay. If the next latch would already take the input before the logic settles the output would be wrong. In this example, the maximum path delay shown is $4ns$. Since this is the only path, this is the critical path delay. To produce a correct output, the flip-flop has to 'wait' $4ns$, this means that $f_{Max} = 250MHz$.

This sets the definition that f_{Max} is directly set by the delay in the circuit, and while this is true, it is not very often used in an environment where FPGA designers have to work to a set requirement instead of the maximum possible achievable. This means that

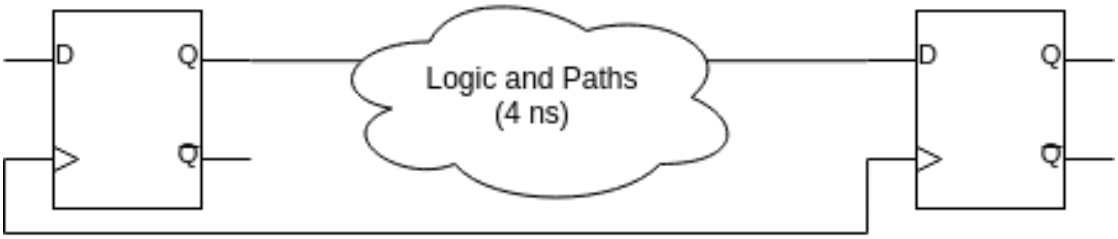


Figure 2: Example of a sequential circuit

the designer has to first set 'constraints' that defines the required frequency. During the 'Max-Delay Analysis' the following definition is used:

Definition 1. ***Slack*** *(max-delay) = Required Time - Data Arrival Time [11]*

A positive slack in this case means that the data has enough time to arrive at the next register. A negative means it fails the requirements. This sets the definition that the lowest or the worst slack will be the limiting slack in a design:

Definition 2. ***Worst Negative Slack (WNS)*** *This value corresponds to the worst slack of all the timing paths for max delay analysis. It can be positive or negative [11]*

The WNS is the limiting factor for the maximum possible achievable frequency in a design. F_{Max} can be calculated on basis of these numbers:

$$\omega_{min} = \omega_{required} - WNS \tag{1}$$

$$f_{Max} = (\omega_{min})^{-1} \tag{2}$$

Further referencing to Slack will be the Max-Delay Slack as described in Definition 1 or noted otherwise. The path containing the WNS, is often also called the 'critical path'.

2.2.2 Pipelining

One of the most known examples of where pipelining is applied to a design is a microprocessor [5]. In these microchips, the whole process of computing an instruction is cut up in smaller pieces with memory elements in between. This makes it possible to have multiple instructions being executed at the same time and opens up a whole world of different parallelism techniques to improve the speed of a processor. Pipelining can also be applied to FPGA designs, by adding new flip-flops to a design.

In Figure 3, a model can be seen that is the same as the example in Figure 2 but with an added register in between. The maximum path delay has been reduced by half

to $2ns$, this leads to a doubling of the frequency: $f_{Max} = 500MHz$. It is at the cost of extra latency. This means it will take an extra clock cycle until the output is valid. For streaming applications where input is constantly being streamed to the design this initial latency should not matter, but for applications where input is not constantly being fed into, this initial delay may cause performance issues. In those cases, adding pipeline stages will be a design consideration between latency and clock frequency.

Another issue presents itself when there are loops present in the design. Simply inserting pipeline stages into a sequential loop might introduce functionality errors. For instance it can be seen in Figure 4 that adding a flip-flop in the feedback loop, results in the functionality changing.

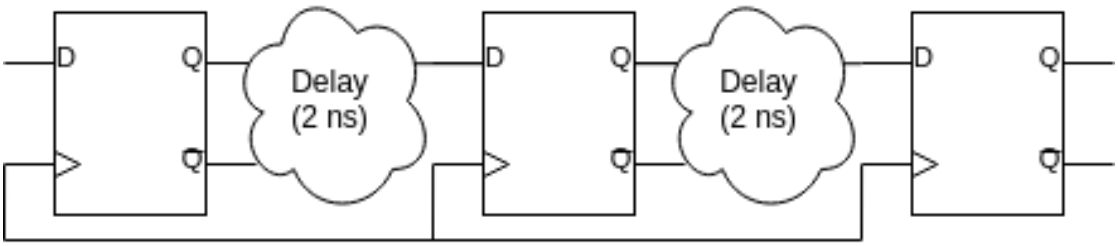


Figure 3: A pipelined model

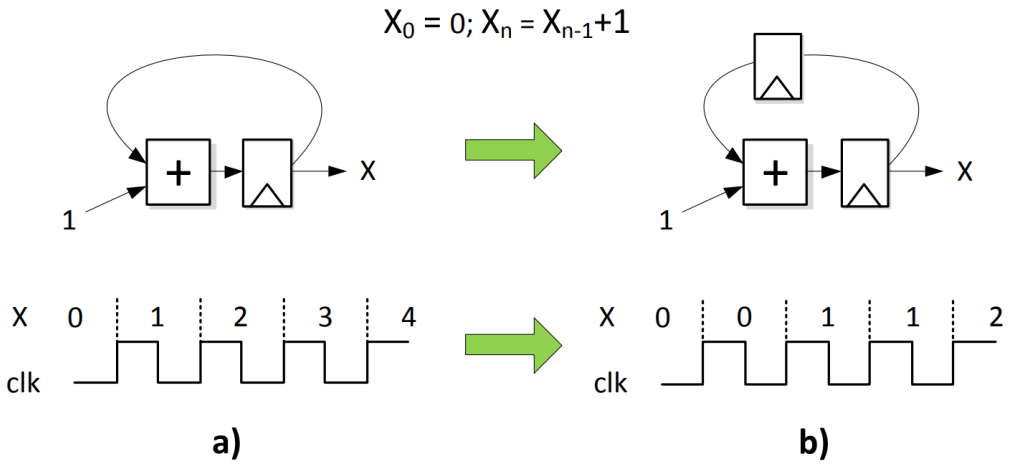


Figure 4: Example of pipelining sequential loops [4]

2.3 RapidSmith2

RapidSmith2 is 'a library for Low-level Manipulation of Vivado Designs at the Cell/BEL Level' [8] written in Java. It is a direct successor of the earlier BYU project that was aimed at the previous IDE of Xilinx: ISE. It has been developed to be able to manipulate a design at a very low level at multiple steps in the FPGA design flow, as can be seen in the Usage Model in Figure 5.

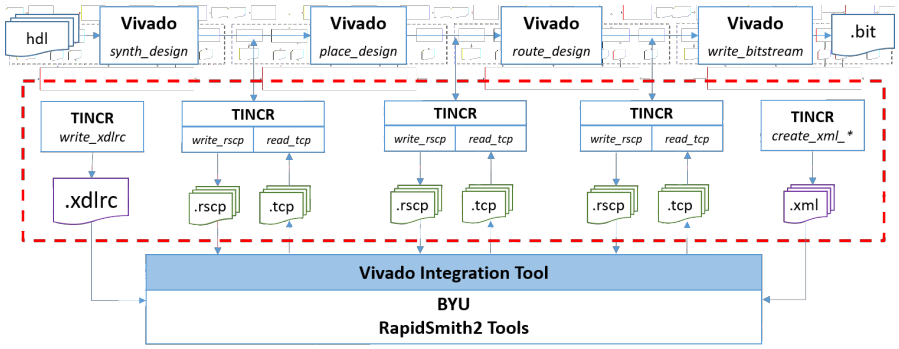


Figure 5: Vivado and RapidSmith2 Usage Model [8]

In Xilinx latest IDE Vivado, a TCL interface was added, which makes it possible to write scripts and automate different parts of the design flow. While this works well for automation, it is not always an appropriate tool for writing bigger external tools that need to modify and analyze parts of the design. RapidSmith2 heavily leans on the TINCR project. TINCR is a TCL Framework for creating external tools for Vivado. It provides an API for querying and manipulating existing Vivado design and has a set of comments to export designs to open file formats. This last set of commands is used to generate the required files that RapidSmith2 can open and parse. At any point in the FPGA Design Flow, TINCR can create a so called 'RapidSmith2 Checkpoint'. This 'checkpoint' can be loaded by RapidSmith2 and the design will be represented by it's RapidSmiths2 Java objects and can eventually be exported again to a 'TINCR Checkpoint'. This checkpoint file can again be read by TINCR and imported into Vivado for further processing. [8]

After loading a checkpoint file, the imported checkpoint is defined into three top-level java classes: Device, CellLibrary and CellDesign. Device contains all the physical structures of the FPGA, and CellLibrary all the possible implementable cells on the physical FPGA. CellDesign is the structure that encapsulates the full design.

2.3.1 CellDesign

CellDesign is the top-level Java class which contains the entire imported design and provides several methods to manipulate the design. As described in Section 2.1, the synthesis of HDL code results in a low-level netlist containing implementable elements. The

CellDesign class encapsulates this netlist and provides physical references to the elements, provided that the design has been placed and/or routed. A detailed data structure of this class can be seen in Figure 6.

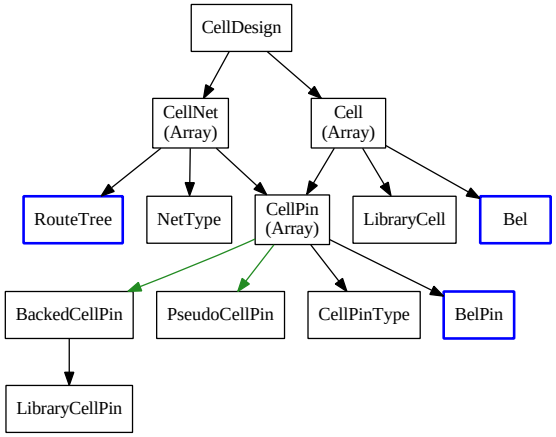


Figure 6: RapidSmith2 design data structure tree. [8]

A CellDesign or netlist is splitted up in two structures: Cells and CellNets. Cells are the individual elements in a netlist such as LUTs and flip-flops. These types of cells are all described in the CellLibrary, so that each individual Cell object contains a 'LibraryCell'. CellNets contain the connections between the cells. Both 'Cells' and 'CellNets' contain the individual CellPins and could be requested from either object. In the case of the Cell, it contains all the individual pins of the Cell, and in the case of the CellNet it contains all the pins that are connected with each other in a Net. RapidSmith2 provides all the methods to manipulate existing Cells and CellNets or create new ones and add them to the design.

3 Method

This chapter will show related work and look more in-depth at the possible methodologies to improve the throughput.

3.1 Automatic Pipelining Methodologies

There are multiple papers that have researched methods to optimize synchronous designs. First were Leiersons and Saxe in 1991 on their *Retiming* approach [7]. The Retiming method is a technique to balance all existing registers in a synchronous design to find the optimum location for them in the design to reduce the critical path. This optimization can thus improve the throughput of a system while keeping the initial latency constant. Two minor additions to this approach have been proposed by Weaver [9], *Repipelining* and *C-Slow Retiming*.

The technique Repipelining differs from Retiming in that it will introduce extra latency but further improve the throughput of the design. By adding an N amount of registers to the input signals of the design, retiming can be used to optimally balance these registers to introduce an N amount of pipeline stages to the design. One issue in this approach is that doesn't find an optimal amount of pipelines stages and can pass a point of no additional performance increase.

Another option is C-Slow Pipelining, further investigated by Weaver et al. in 2003 [10]. C-Slow pipelining is implemented by replacing registers in the design by a sequence of a separate amount C of registers. Again, retiming is used to balance the flip-flops through the design. The biggest difference between this method and retiming or repipelining is that the technique can improve design with feedback loops. This comes with a requirement; "Both repipelining and C-slow retiming can be applied only when there is sufficient task-level parallelism, in the form of either a feed- forward pipeline (repipelining) or independent tasks (C-slowing)" [9]. In C-Slow pipelining the design is separated in C tasks. Enough task parallelism should be present to improve the throughput of the design. The scope of this thesis limits itself to designs without feedback loops. This means C-Slow pipelining does not need to be further investigated, since streaming applications should show enough task-level parallelism in terms of a feed-forward pipeline.

When analyzing the performance of only the retiming algorithm in Table 1 of the C-Slow pipelining tool developed by Weaver et. al [10], some evident issues arise. While the Synthetic datapath and LEON Processor benchmark get marginally improved maximum frequencies, both the AES core and Smith/Waterman benchmark actually perform worse. Since these two benchmarks consist of single feedback loops, these programs do not benefit from the Retiming algorithm. Nevertheless, Weaver also notes that "This tool does not use a perfectly accurate delay model and has to place registers after retiming, so it sometimes creates slightly sub-optimal results." [9]. When looking in-depth into the

Retiming algorithm, it is evident that a proper estimation of the delays in the design is absolutely necessary for decent results.

Benchmark	**Unretimed**	**Automatically Retimed**
AES core	48 MHz	47 MHz
Smith/Waterman	43 MHz	40 MHz
Synthetic datapath	51 MHz	54 MHz
LEON Processor	23 MHz	25 MHz

Table 1: Results of retiming four benchmarks [9]

3.2 Xilinx Vivado Pipeline Analysis

In a Xilinx paper by Ganusos et. al. [4], an algorithm for improving performance is looked into. They introduce an automated extra pipeline analysis that has been implemented from Vivado 2015.3 on as '*report_pipeline_analysis*'. By using the timing models implemented in Vivado, an existing design is analyzed for critical paths, extra pipeline stages are added and a report is generated. Since the implementation uses the outcome of the report of the pipeline analysis, the algorithm will be described here further in detail.

Similar to the Retiming algorithm, the design is being modeled by a graph. In the retiming algorithm , the circuit is defined as a graph with each node being a propagation delay and each edge being a path that can either contain a register or not. In the Xilinx algorithm, each node represents a 'pin'; both IO-pins and Cell pins. Each edge represents the path between the pins, either internally in the Cell or externally. An example can be seen in Figure 7.

A graph is an excellent model to computationally describe a design. It describes the flow of the data in the design and timing information can also easily be added to each node. An algorithm can simply traverse throughout the circuit. After having defined how a design should look, a valid pipeline stage can be described.

Earlier has been established that a pipeline stage is an added register to an existing design. A very simple single input to single output example of this was shown in Figure 3. However, it becomes more complicated when dealing with multiple inputs, outputs and paths in a design. A valid pipeline stage in this case are multiple added registers in the design, where every single path between an input- and output pin only contains a single added register. This ensures that every forward path contains a single added latency, and thus the functionality of the design does not change. Naturally, the added register can only be inserted before or after a Cell, not between the pins. Due to hardware limitations some pins can not have a direct register in front or after either. Gusanov et. al. uses the following definition:

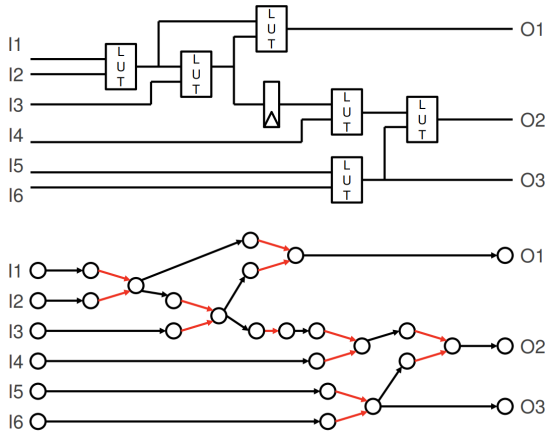


Figure 7: A digital circuit and its graph representation. Red colored edges are internal paths [4]

Definition 3. *A pin is qualified as* ***legal*** *if a pipeline register can be inserted on its connected net [4]*

They propose the following algorithm on how to implement a pipeline stage, their consideration for designs with loops have been omitted.

1. Append all the legal pins p into list L.
2. Sort L on basis of Slack
   (a) First priority on increasing Slack
   (b) Second priority on decreasing possible slack improvement by virtually inserting a pipeline stage
3. $stagePins = \emptyset, TFIPins = \emptyset, TFOPins = \emptyset$
4. For each p of L:
   (a) If $p \in TFIPins \cup TFOPins$; Continue.
   (b) Append p to $stagePins$
   (c) Append all pins of L that are the Transitive Fan-In to $TFIPins$
   (d) Append all pins of L that are the Transitive Fan-out to $TFOPins$
5. Return $stagePins$

Listing 1: Pipelining algorithm [4]

In Step 2 of the algorithm in Listing 1, the list of legal pins are sorted first increasingly on slack. As paths cross multiple pins with the same slack, the second sort priority is necessary to find out which pins would introduce the best slack improvement. In Step 4,

the fan-in and fan-out is added to two extra lists. The 'Fan-In' are all pins of the possible paths that can lead to the point where the register gets added, the 'Fan-Out' are the pins of all further possible paths. This ensures that every forward path only contains a single latency as $TFIPins \cup TFOPins \notin stagePins$. Further mathematical proof can be read in the paper [4].

Ganusov et. al. proposes two methods of implementation:

1. Using the generated report to directly implement all calculated pipeline stages, then place and route. They note that this gives a reasonable estimation, but without placement and updated timings it is not perfect.
2. An iterative approach:
   (a) First Synthesis, Optimization and Placement
   (b) Till no improvement is found:
      i. Recalculate pipeline report and insert Pipeline.
      ii. Update Placement and Timings.
   (c) Lastly Routing

   This approach can improve frequency improvements since actual placement and timings get updated during the insertion of pipeline stages.

While the paper describes the results of the iterative approach, none of the approaches have actually been made available to the Xilinx Vivado suite at the time of writing. The user of Vivado currently only has the tool that will generate the report available. Inserting the pipeline stages is up to the designer.

4 Implementation

In Section 3, various pipelining methodologies have been investigated in-depth. The approaches by Weaver [9] have been looked at and only Repipelining is a viable implementation that would increase throughput of the design. One downside of this approach is that it needs a proper implementation of the retiming algorithm, which requires a well designed time model for the specific FPGA. The Xilinx Automated Pipeline Analysis [4] evades this problem due to the fact that it can make direct use of the timing models in Vivado. This also makes it an implementation that is unspecific to a certain FPGA family, as long as it has Vivado support.

For various reasons, the earlier algorithm described in Section 3.2 has not been implemented externally and the generated Vivado report is used. First, implementing a timing model would possibly introduce less optimal results due to a sub-optimal model. Secondly, it would make the implementation specific to a certain FPGA family. Thirdly, at this time of writing there is no proper software or library to combine the timing information that Vivado holds with the low-level modifications RapidSmith2 offers. While the 'Automatic Repipeliner' was written with the ZedBoard containing the Zynq-7000 SoC in mind, it is not only written specifically for this FPGA. With small changes, the implementation could be very well used on any Vivado supported FPGA Family.

The program was been written in Java 1.8. Vivado version 2017.2 is used. It heavily relies on the functionality provided by RapidSmith2, which again leans on the Vivado methods that TINCR provides. One requirement when using RapidSmith2 is to first generate the needed device files. The RapidSmith2 Tech Report [8] describes in full detail in how to create and install new design files. Note that every FPGA with a different part number requires new device files.

```
1  if { $argc != 2 } {
2          puts "The rscp.tcl script requires a top-level name and extension name"
3          puts "vivado -mode batch -source rscp.tcl -tclargs aes_top v"
4          exit 2
5  } else {
6          link_design -part xc7z020clg484
7          read_verilog [glob *.[lindex $argv 1]]
8          synth_design -top [lindex $argv 0] -flatten_hierarchy full
9          #100MHZ Clock
10         create_clock -period 10.000 -name CLK -waveform {0.000 5.000} [get_ports {clk}]
11         write_checkpoint -force [lindex $argv 0].dcp
12         package require tincr
13         tincr::write_rscp [lindex $argv 0]
14         close_project
15
16         file delete {*}[glob *.log]
17         file delete {*}[glob *.jou]
18         file delete {*}[glob *.dmp]
19 }
```

Listing 2: Synthesis 'rscp.tcl' TCL Script

The TCL script in Listing 2 is used to automatically synthesize a HDL design and outputs the needed 'RapidSmith2 CheckPoint' for the implementation. Vivado can run

in 'batch-mode' and sources TCL script to automatically execute, even arguments can be provided. 'rscp.tcl' is aimed at Verilogs designs for the ZedBoard specifically, but can easily be altered for VHDL and different part numbers. The script also includes an arbitrary 100 MHz clock to the design for timing purposes, writes a Vivado Checkpoint for further possible evaluation and simulation of the original design and deletes all the log- and dump files Vivado creates. Note that RapidSmith2 can only work with designs that have a fully flattened hierarchy.

```
1  if { $argc != 2 } {
2          puts "The sim.tcl requires a TCP file and testbench file"
3          puts "vivado -mode batch -source sim.tcl -tclargs aes_128_After test_aes_128.v"
4          exit 2
5  } else {
6          package require tincr
7          tincr::read_tcp [lindex $argv 0].tcp
8          save_project_as -force Proj/[lindex $argv 0]
9          write_vhdl -force -mode funcsim [lindex $argv 0]_sim.vhd
10         set_property SOURCE_SET sources_1 [get_filesets sim_1]
11         import_files -fileset sim_1 -norecurse Test/[lindex $argv 1]
12         start_gui
13         file delete {*}[glob *.log]
14         file delete {*}[glob *.jou]
15         file delete {*}[glob *.dmp]
16 }
```

Listing 3: Simulation 'sim.tcl' TCL Script

The second script in Listing 3 is for evaluation and simulation of the generated design. As described earlier in Section 2.3, RapidSmith2 applications exports the design as a TINCR CheckPoint. The script opens this checkpoint, adds a bench test and turns it into a Vivado Project that can be used to simulate the design.

4.1 Pipeline Report Interpreter

First it is needed to look at how the pipeline analysis tool of Vivado works. 'report_pipeline_analysis' has to be used in combination with the '-include_paths_to_pipeline' argument and exported to a file for a complete report. The generated document has three main parts: 'Report Description and Glossary', the 'Intra-Clock Summary', and the 'Paths to pipeline'.

The first part is only a description of the loop and a small glossary of the tool. The 'Intra-Clock Summary' is a report to indicate the current f_{Max} and the possible f_{Max} after every pipeline insertion. For the implementation of the pipeline tool, only the first and last row are of interest. An example of this is shown in Table 2. The first row being the current design which shows us the actual f_{Max}. The last row being the theoretical finalized design with the improved f_{Max} and the newly added delay in the feed-forward path.

The 'Paths to pipeline' as shown in Table 3 contain all the individual 'instructions' to pipeline the design. All instruction are written down as path cuts, with the start- and endpoint listed with the necessary added registers in that path.

Clock	Added Latency	Ideal FMax	Ideal Delay	Requirement	WNS	Added Pipe Reg	Total Pipe 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 2: Example first and last line of the Intra-Clock Summary

Clock/PathGroup	Path Cut	Added Pipe Reg	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 3: Example of two 'instructions' of 'Paths to pipeline'

An interpreter is written in Java to import the information of the pipeline report to the RapidSmith2 application. A detailed UML diagram of the classes involved can be seen in Figure 8. All functionality has been en-capsuled by the 'ReportPipeLineInterpreter' that handles the IO with the file and splits the report in two. By using the 'parse' method it generates the 'IntraClockSummary' and 'PipelineSummary' class. Both these classes individually parse their part of the report and have methods to return the parsed information. Through the main class all the get-functions of these classes can be accessed.

All individual instructions get, after being parsed in the 'PipelineSummary' class, stored in a very basic 'PipelineInstruction' class. This instruction contains all necessary information for inserting a pipeline cut at the right location. An extra addition are the set- and get-functions for the AddedDelays, this provides the instructions a current state of the already added registers. This will be explained in the next section.

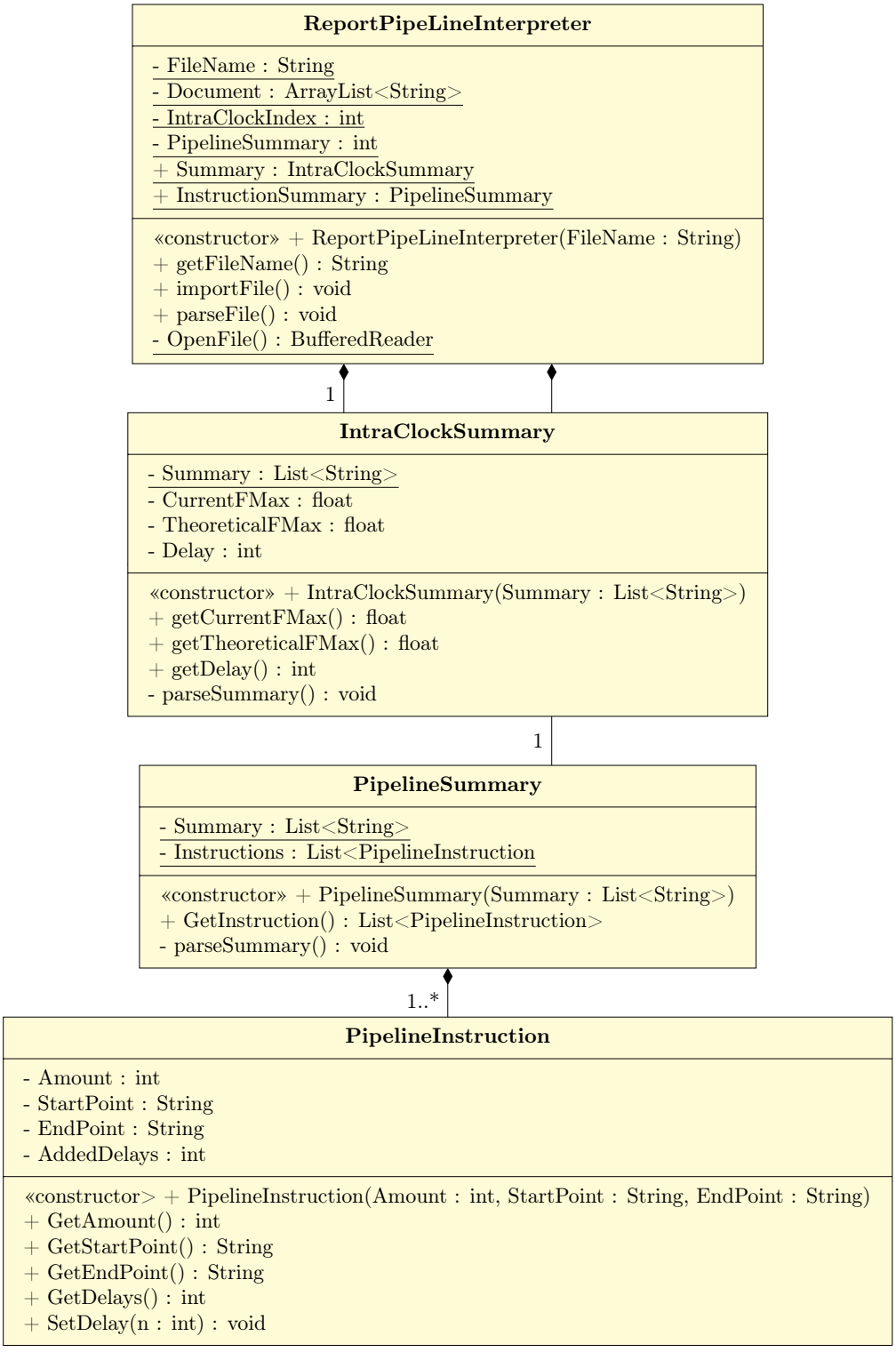


Figure 8: UML diagram of ReportPipeLineInterpreter and composition classes

4.2 Pipeline Insertion and Recycling

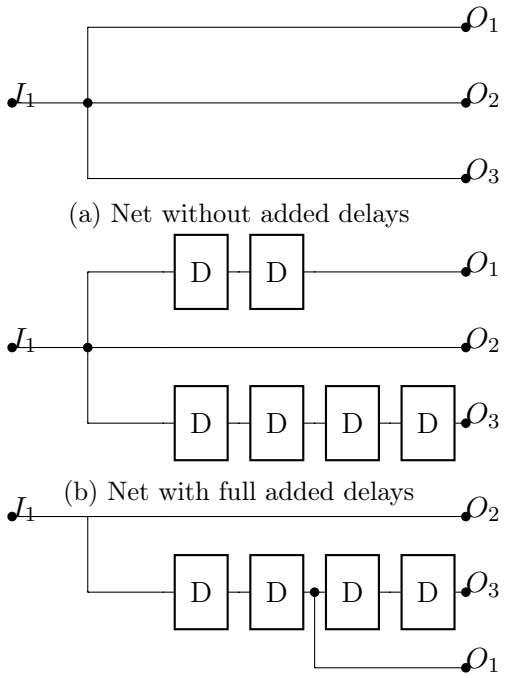
After all the instructions from the pipeline analysis report are parsed, the application will move onto executing these instructions. The RapidSmith2 library provides us with the tools to add these registers. The structure in how RapidSmith2 operates is described in more detail in Section 2.1. Due to the fact that no automatic interface exists yet that can run a design multiple times through Vivado and RapidSmith2 only the first direct approach as listed in Section 3.2 is investigated.

The instructions as seen in Table 3 are supposed to be read that an amount of registers should be added on the path between the Startpoint and the Endpoint. This is the 'path', but this does not directly mean that the CellNet looks exactly the same. For instance, the CellNet shown in Figure 9a shows a net with a single input and multiple outputs. When executing the instruction, it should not affect the other 2 paths on the CellNet. That is why the inserted registers should always be added just before the Endpoint. Multi-driven nets, nets which have multiple input signals, are not supported.

Fortunately, the naming convention the pipeline report uses is quite simple. The first part of the Start- or Endpoint is the Cell name, the part after the last slash '/' is the CellPin name. With the use of regex the String can be split and both the Cells and CellPins can be retrieved from the CellDesign. The CellNet connecting the StartPoint to the EndPoint is being disconnected from the EndPoint CellPin, a new array of N registers with new CellNets inbetween is set in place and is ultimately connected again to the EndPoint.

When empirically looking at the Pipeline report, it was often seen that there could be multiple instructions with the same StartPoint and different EndPoints, so although it were two other paths, the CellNet would be exactly the same. This could result in situations were unnecessary extra registers would be added. An example can be seen in Figure 9b. Two delays would be added in front of EndPoint O_1 and four delays in front of O_3, which ultimately results in six added flip-flops. Another option would be to recycle existing registers like in Figure 9c. The resulting design is functionally and timing-wise the same as Figure 9b but with a save of two added registers.

Because of the recycling of the newly added registers, a state (or AddedDelays) had to be added to the 'PipelineInstruction' class, and uses the algorithm shown in Listing 4. Sorting the list of pipeline instructions ensures that the least necessary amount of registers will be added. The newly added flip-flops and CellNets inbetween will always be conveniently named after the StartingPoint of the instruction, so that new instructions containing a state can easily connect to the right register.



(a) Net without added delays

(b) Net with full added delays

(c) Net with recycled delays

Figure 9: An example of delay insertion

1. Get list L containing all pipeline instructions p.
2. Sort list decreasingly on the amount N of needed registers of p.
3. For each $p \in L$:
   (a) If state of p is zero (no earlier added delays):
      i. Add N registers before p EndPoint.
      ii. Set State to N.
      iii. For all $p \in L$ with the same StartPoint, set state to N.
   (b) Else:
      i. Connect EndPoint of p with earlier added register CellNet.
4. Export design to TINCR checkpoint.

Listing 4: The Pipeline insertion algorithm including recycling

5 Evaluation

This chapter will go through the results and evaluation of the automatic pipeline implementation. Several existing designs will be tested on performance increase and resource usage. A short introduction to the FPGA architecture is added to be able to evaluate the resource usage of the applications.

5.1 ZedBoard and the Zynq-7000

Evaluation of the Automatic Pipeliner is aimed at the Avnet ZedBoard [2]. The FPGA SoC on the chip is the 'Zynq-7000 All Programmable SoC XC7Z020-CLG484-1'. This Zynq-7000 SoC combines a Dual-core ARM Cortex-A9 with a Artix-7 FPGA [13]. The processing power of the ARM core will not be used, since only the FPGA architecture is of interest. The ZedBoard has a FPGA that holds 53,200 LUTs and 106,400 registers, which means that there abundance of Flip-flops with a 2:1 ratio towards LUTs.

All logic elements are embedded in so called 'Configurable Logic Blocks'(CLB). One CLB contains two slices. One slice again has four LUTs, eight Registers, Carry logic and multiplexers on the same area [12]. Not all elements on a slice have to be used and signals can be fairly freely routed through the area. This means that for designs that do not make optimally usage of this abundance of registers might have enough unused flip-flops available that the used FPGA area will not dramatically increase by inserting pipeline stages. Adding pipeline stages will thus not immediately increase the used area as much as the added registers.

5.2 Advanced Encryption Standard (AES) Core

The first application looked at for repipelining is the 'Fully Pipelined AES Core' [3] provided by Freecores. AES is an encryption technique that can encrypt data blocks of 128, 192, and 256 bits. AES can be implemented with loops, as multiple stages have to be executed multiple times. Fortunately, based on the implementation it is known how often these steps have to be repeated so the application can be unrolled in multiple identical stages. This implementation is such an unrolled 128-bit AES core.

The documentation of the Core lists their own results using Xilinx ISE WebPack and synthesizing for the Xilinx 5VLX50T. This is listed as 'Reported' in Table 4 and 5. As can be seen the original results perform $\approx 15\%$ worse. This can be due to multiple factors: It has been synthesized in a different IDE and for another device. Less registers are being used which could mean that the synthesizer optimized some flip-flops away. The theoretical version is what the tool predicts and shows a massive improvement. For the Total Registers at Repipelining, the total amount of flip-flops without recycling is shown.

Unfortunately, this prediction of the tool seems overly optimistic and the actual repipelining application shows an $\approx$ 20% throughput improvement with 19 added delay. After gathering these results, both the original and repipelined designs are being placed 'OUT-OF-CONTEXT' (without placing IO) and the Slice usage is measured. As seen, the Register Utilization increases with 152.70%, this is more than the increased Slice usage as predicted earlier. This does mean that a 20% increase of performance means that the application is more then twice as big. Fortunately by recycling and not directly following the instructions of the tool, the repipeliner is able to use 43535 less registers then predicted by the tool.

The recycle addition to the initial algorithm shows a lot of improvement when comparing the resource usage. While the Throughput stays the same, the recycled version uses less registers and slices. An interesting point to note is that f_{Max} after Placing and Routing is considerably lower for the implementation without recycling, this is most likely due to the way higher Slice Utilization introducing more routing delay.

Unfortunately, the performance does not increase linearly to the size, meaning that a doubling of Slice usage does not convert into the doubling of the throughput improvement. This means that parallelization of this AES core would perform better at a lower area cost.

	Frequency (MHz)	**Throughput**	**Improvement**	**Extra Delay (cycles)**
Reported	~330	~42 Gbps	-	-
Original	278.96	35.71 Gbps	-14.98%	-
Theoretical	557.92	71.41 Gbps	70.02%	19
Repipelining	395.14	50.58 Gbps	20.43%	19
Repipelining + Recycling	395.14	50.58 Gbps	20.43%	19

Table 4: Throughput Improvement AES

	Total Registers	**Register Utilization**	**Slices**	**Post P+R (MHz)**
Reported	7873	-	-	-
Original	6482	6.09%	1699	190.94
Repipelining	59912	56.31%(+824.63%)	10490 (+517.42%)	202.35
Repipelining + Recycling	16377	15.39% (+152.70%)	3860 (+127.19%)	302.48

Table 5: Resource Usage AES

5.3 FIR Filter

Another design that has been evaluated is a FIR Filter provided by Digi-Key [6]. The output of a standard FIR filter is defined by the following equation:

$$y[n] = \sum_{i=0}^{N} b_i \cdot x(n-i) \tag{3}$$

Where N is the order of the filter and b_i are the coefficients. Although this filter could be implemented serially, with a known and constant maximum filter order, it is possible to implement this with parallel execution. A filter with order $N = 4$ with 4-bits of data has been implemented.

The FIR-Filter has been evaluated in the same way as the AES core in Section 5.2. The results are shown in Table 6 and 7. In this result again a big difference is seen between the theoretical throughput and the implementations. The recycling implementation uses 240 less registers while keeping the throughput roughly the same. After Placing and Routing the maximum frequency is again higher for the recycled implementation. An interesting number is the amount of slices in both implementations: Even with the extra pipeline stages added, the Slice Utilization increased way less then the Register Utilization.

In this example it is seen that the performance increase after recycling actually is higher than the area increase. This means that the recycling optimization performs better than parallelization would.

	Frequency (MHz)	**Throughput**	**Improvement**	**Extra Delay (cycles)**
Original	167.15	668.6 Mbps	-	-
Theoretical	353.63	1.41 Gbps	110.89%	2
Repipelining	220.19	880.76 Mbps	31.73%	2
Repipelining + Recycling	241.12	964.48 Mbps	44.25%	2

Table 6: Throughput Improvement FIR, N=4

	Total Registers	**Register Utilization**	**Slices**	**Post P+R (MHz)**
Original	42	0.04%	40	162.31
Repipelining	340	0.32% (+700%)	101 (+152.5%)	198.02
Repipelining + Recycling	100	0.09% (+125%)	45 (+12.5%)	234.25

Table 7: Resource Usage FIR, N=4

What can be seen from both results is that the Vivado Pipeline Analysis tool is highly optimistic in their theoretical assessment of how much improvement the Repipelining algorithm would bring. The paper [4] shortly describes how their maximum frequency is obtained; The maximum device frequency and the max loop frequency is obtained and the smallest of these two is the maximum frequency possible. This frequency is unfortunately not reached. Both the original pipelining implementation and the one that recycles added registers show that the extra flip-flops are unnecessary and after placing and routing even decrease performance.

6 Discussion

In this thesis various new and different topics have been looked into that are not part of the standard Electrical Engineering curriculum. Research has been done in various techniques to improve throughput on logic level and a good perspective has been brought out on the current options in the field related to repipelining. Unfortunately working with the tooling brought some problems.

One fairly big issue[1] that arose was a problem with the RapidSmith2 checkpoint importer. Lots of time was spend here to try to evade the issue. For some ports in the AES design in Section 5.2 for example, there was a problem that the netlist interface could not read them and further changes to the design would be impossible. RapidSmith2 is a library in active development by the BYU Configurable Computing Lab so bugs and errors are to be expected. Fortunately, after filling in an issue report the error was quickly resolved by their team.

An issue with the Vivado Pipeline Analysis tool is that it is not that greatly documented and adds unnecessary registers as shown in Section 4.2. While the paper describes their algorithm in detail, it is difficult to set over this knowledge to their actual product which uses different technical terms and 'speaks' in paths instead of pins. The theoretical frequency listed is way too optimistic and it seems that the timing data used by the tool, in terms of amount of paths, is incomplete since the theoretical slack improvement is never reached. But this is only guessing work since the complete workings of the tool are not publicized. Finally, the Vivado tool gives instructions to unnecessary registers that do not improve the performance of the design, increase overhead tremendously and even perform worse after placing and routing.

[1] Github Issue: `https://github.com/byuccl/RapidSmith2/issues/358`

7 Conclusion

This research shows that automatic algorithms can be used and are a viable approach to improve throughput of an existing streaming application. External tooling has been used and made to provide the designer a program that can automatically improve their design post-synthesis. Even further improvement in terms of overhead reduction has been made in the form of 'Recycling'.

Results show that a decent improvement can be made in respect to extra latency and more overhead. While latency does not matter to a typical streaming application, the resource overhead should be noted. This design choice between extra register overhead, that shows itself in more area usage, and extra performance is in the end up to the designer. Parallelization should also be considered, especially when the throughput improvement reached is lower than the increase of area. In the case of a parallel solution, the outcome is always static: the doubling of the area results in the doubling of throughput. This does mean that it could perform worse if repipelining actually improves throughput beyond area increase.

The repipelining algorithm used shows improvement, and exported designs show correct functionality. Additionally it shows that RapidSmith2 is mature enough to be used for low-level changes to a Vivado design at any point in the FPGA design flow. The fact that the actual performance does not come close to the theoretical approximation proves to be hopeful for further Repipelining analysis in combination with RapidSmith2.

8 Recommendations

The first steps towards automatic throughput improvement of FPGA designs have been made. But more areas which could provide more improvement should be investigated.

At first the iterative approach described in Section 3.2 has not been investigated. The paper [4] describes this approach as the superior one since it can make usage of the improved timing knowledge after every inserted pipeline stage. Unfortunately, this is not a desirable approach, while in the first place the Vivado tool can't export single pipeline stages, it would also be a very tedious approach. In the time of writing an interface that could automate this process does not exist. A recommendation would be to write a Java interface that could automate the process of importing and exporting Vivado and RapidSmith2 designs. The TCL commands used by Vivado are fairly simple so it would be possible to write a TCL script generator and let Vivado execute these scripts by command of the Java RapidSmith2 application. With this approach, most of the steps such as synthesis could be done in the Java application directly.

Additionally the algorithm proposed by Ganusov et. al. [4] has to be implemented directly into Java instead of using the tool. At this point the reason why the tool does not reach the theoretical estimation cannot be further investigated. An option would be to design a timing model, unfortunately this would be very specific to an FPGA family. Another approach would be to investigate the timing information that Vivado provides. This report can provide the N worst paths the design currently has. RapidSmith2 presently does not have a complete interface for the Timing Report available. Parsing this information could provide us with enough timing data to iteratively repipeline the design.

At last more evaluation should be done on the hardware implementation of the repipelined designs. The research done has only evaluated throughput improvement based on the Vivado timing models. Actual implementation on the ZedBoard should be investigated and measurements should be done to get more accurate results.

References

[1] Aldec. Xilinx design flow. https://www.aldec.com/en/solutions/fpga_design/fpga_vendors_support/xilinx--xilinx-fpga-design-flow.

[2] Avnet. *ZedBoard Product Briefs*, 2018.

[3] Subhasis Das. Fully pipelined aes core. https://github.com/freecores/aes_pipe, 2009.

[4] Ilya Ganusov, Henri Fraisse, Aaron Ng, Rafael Trapani Possignolo, and Sabya Das. Automated extra pipeline analysis of applications mapped to xilinx ultrascale+ fpgas. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–10. IEEE, 2016.

[5] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*, chapter C. Elsevier, 2011.

[6] Scott Larson. Digi-key: Fir filter (vhdl). https://www.digikey.com/eewiki/pages/viewpage.action?pageId=78086825, 2018.

[7] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, 1991.

[8] Brent Nelson, Thomas Townsend, and Travis Haroldsen. Rapidsmith2 - a library for low-level manipulation of vivado designs at the cell/bel level. Technical report, Department of Electrical and Computer Engineering, Brigham Young University, feb 2018.

[9] Nicholas Weaver. Retiming, repipelining and c-slow retiming. In *Reconfigurable Computing*, pages 383–399. Elsevier, 2008.

[10] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement c-slow retiming for the xilinx virtex fpga. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 185–194. ACM, 2003.

[11] Xilinx. *Vivado Design Suite User Guide, Design Analysis and Closure Techniques*, 2012.

[12] Xilinx. *7 Series FPGAs Configurable Logic Block*, 2016.

[13] Xilinx. *Zynq-7000 SoC Data Sheet: Overview*, 2018.