Automated Extra Pipeline Analysis of Applications mapped to Xilinx UltraScale+ FPGAs

Ilya Ganusov\textsuperscript{1}, Henri Fraissee\textsuperscript{1}, Aaron Ng\textsuperscript{1}, Rafael Trapani Possignolo\textsuperscript{2}, Sabya Das\textsuperscript{1}

\textsuperscript{1}Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124-3400. \{ilya.ganusov\}@xilinx.com
\textsuperscript{2}Dept. of Computer Engineering, University of California, Santa Cruz, CA 95064. \{rpossign\}@ucsc.edu

Abstract—This paper describes the methodology and algorithms behind extra pipeline analysis tools released in the Xilinx Vivado Design Suite version 2015.3. Extra pipelining is one of the most effective ways to improve performance of FPGA applications. Manual pipelining, however, often requires significant efforts from FPGA designers who need to explore various changes in the RTL and re-run the flow iteratively. The automatic pipelining approach described in this paper, in contrast, allows FPGA users to explore latency vs. performance trade-offs of their designs before investing time and effort into modifying RTL. We describe algorithms behind these tools which use simple cut heuristics to maximize performance improvement while minimizing additional latency and register overhead. To demonstrate the effectiveness of the proposed approach, we analyse a set of 93 commercial FPGA applications and IP blocks mapped to Xilinx UltraScale+ and UltraScale generations of FPGAs. The results show that extra pipelining can provide from 18\% to 29\% potential Fmax improvement on average. It also shows that the distribution of improvements is bimodal, with almost half of benchmark suite designs showing no improvement due to the presence of large loops. Finally, we demonstrate that highly-pipelined designs map well to UltraScale+ and UltraScale FPGA architectures. Our approach demonstrates 19\% and 20\% Fmax improvement potential for the UltraScale+ and UltraScale architectures respectively, with the majority of applications reaching their loop limit through pipelining.

I. INTRODUCTION

Extra pipelining is one of the most effective ways to improve performance of FPGA applications. It improves Fmax by introducing registers on the most critical paths and reducing effective number of gates and wire distance that a signal has to traverse in one cycle. Figure 1 shows a simple example datapath which can be sped up by 2x by adding an extra pipeline stage. FPGA designers can often narrow the performance gap compared to ASICs by inserting deeper levels of pipelining into their designs.

Typically, synchronous designs have natural levels of pipelining. For example, all bits representing a number need to be aligned on a clock cycle before being written to memory. Another example of natural pipelining are Finite State Machines (FSMs), which change their state on a clock-by-clock basis. In this paper, we refer to extra pipelining as going beyond the natural pipelining required for correct functional operation of a design.

In practice, extra pipelining is a manual and laborious process. Designers have to identify a collection of the most critical paths, manually modify RTL to insert extra registers on those paths, add extra registers to balance the flow of data, verify that introduced changes preserved functionality, and then re-run implementation flow. After sign-off timing, the designer may discover new paths that need to be pipelined - and the process repeats. This process is both time-consuming and error-prone. In addition, it is not clear ahead of time how many extra stages of latency will have to be added to the design to reach the desired Fmax.

To alleviate this problem, the Xilinx Vivado Design Suite [17] release 2015.3 introduced a new report_pipeline_analysis flow for automated pipeline analysis. This paper describes the methodology and algorithms behind this analysis tool. This methodology allows designers to explore pipelining opportunities without changing the RTL and see the full set of design trade-offs before making any design changes. Using this flow, FPGA users can analyze arbitrarily large designs at any stage of the implementation flow - post-synthesis, post-place, or post-route. It automatically classifies all design paths into feedforward (pipelinable) and loop (non-pipelinable) paths, and performs consecutive cuts of the entire design to find out the best improvement in Fmax per extra stage of pipelining.

In addition to pipelining critical paths, it makes sure that all parallel non-critical paths are also pipelined to preserve correctness of the dataflow. Our pipelining algorithm uses simple cut algorithms to maximize performance improvement while minimizing additional latency and register overhead. After analyzing the design in this manner, it presents detailed information on trade-offs between extra levels of pipelining, achievable frequencies, and number of registers required to implement those changes.

We evaluate the viability and effectiveness of the implemented framework on a suite of 90+ industrial applications mapped to production Xilinx UltraScale+ FPGAs. We run our pipelining exploration tool on a set of representative designs and demonstrate that extra pipelining can potentially improve performance by 18\%-29\% on average. We also show that the distribution of performance improvements is bimodal, with roughly half of the benchmark suite showing performance potential of more than 5\%, while the other half has most critical critical paths in loops and cannot be improved with pipelining alone.
We further evaluate how well highly-pipelined designs map to the Xilinx UltraScale+ FPGA architecture [1]. We insert extra registers into the designs with pipeline potential based on the cut schedule suggested by the `report_pipeline_analysis` tool and measure Fmax improvement after re-placing the design. Results show that extra pipelining improves Fmax by 19% on average and final performance in most designs meets or exceeds the initial loop limit. The few designs with significant gap between loop limit and achieved Fmax are explained mostly by the limitations of our insertion flow which cannot modify combinational DSP cascades, which we plan to address in the future. In addition, we find that extra pipelining increases register count by about 60% on average and in most designs the area does not grow with extra pipelining since the flip-flop to LUT ratio after pipelining is significantly less than the 2:1 ratio provided in the UltraScale+ architecture.

The rest of the paper is organized as follows. Section II presents pipelining methodology used in this paper. Section III describes algorithms and implementation details of the proposed framework. Section IV discusses experimental setup and results. Section V talks about related work and Section VI concludes the paper.

II. PIPELINING METHODOLOGY

This section provides a brief perspective on pipelining opportunities and limitations. First, we discuss how pipelining improves performance and how functional correctness can be guaranteed by balancing parallel paths. Second, we show how pipelining loops can break functionality and therefore limit extra pipelining performance potential. Finally, we briefly discuss how extra pipelining affects the initial state of the design and why it is important to consider this when pipelining feedforward paths followed by loops.

A. Pipeline Stages

Inserting extra pipelining stages improves the performance by reducing the length of the critical paths. Figure 1 provides a simple example of how pipelining works. In the original design in Figure 1a, the most critical path between two registers passes through 3 nets and 2 LUTs and incurs a total delay of 4ns, which corresponds to 250MHz Fmax. We can improve the performance of this simple circuit by inserting an extra register in the middle of that critical path, thus reducing the path to 2ns and increasing Fmax to 500MHz, as shown in Figure 1b.

To preserve functional correctness, inserting a register on a critical path also requires pipelining of all parallel paths. The register between LUTs B and C serves that purpose. Alternatively, we could balance the latency of parallel paths by putting extra registers on the inputs of LUT B, which would require two registers instead of one. When pipelining designs in RTL, designers have to identify all parallel paths and correctly balance the latency on all of them while minimizing the total number of added registers. One of the goals of this paper is to demonstrate how this process can be automated when analyzing designs at the netlist level.

B. Sequential Loops

Pipelining sequential loops may lead to functional errors and presents challenges to automatic pipelining. Figure 2 provides a simple example of how inserting registers in a loop breaks functionality. The original loop in Figure 2a represents a counter which is initialized to 0 and increments the output value by 1 on every cycle. Figure 2b shows how pipelining of this counter loop will produce a different sequence of values from the original.

Since extra pipelining of sequential loops cannot guarantee functional correctness, we exclude all paths which are part of loops from the list of pipelinable paths. In other words, we classify all paths in the netlist as feed-forward (pipelinable) nets and loop (non-pipelinable) nets, and analyze performance improvement potential on feedforward path until we hit the loop performance limit.

Though pipelining alone cannot improve loop frequency, other loop transformations can. For example, the loop in Figure 2 can be split into two separate loops, each of which increments previous value by 2. The frequency of each loop will remain the same, but the output can now be sampled at twice the speed since two values will be produced per cycle. This kind of transformations is outside the scope of this paper.
C. Initial State

As described earlier, in this paper we take a view that feedforward paths can be pipelined while maintaining functional correctness, whereas loop paths are not pipelenable. However, if a design contains a mix of feedforward paths and loops (which is a typical case), then pipelining and balancing feedforward paths are not enough to maintain functional correctness. We also need to consider the initial state of the loops.

For example, let us change a loop in Figure 2a from a counter to an accumulator by replacing constant $l$ with a variable $Y$. Let us also assume that logic paths generating $Y$ are feedforward and the value of $Y$ in initial state is zero. If we insert one extra pipeline stage along those paths, we need to guarantee that $Y$ is zero in the first two cycles. If we insert two extra stages, we need to guarantee that $Y$ is zero in the first three cycles, and so on.

The approach described in this paper generates cut schedule for pipelining feedforward paths, but it does not evaluate the impact of the extra logic required to maintain the initial state. There is some work on automated algorithms to adjust initial state, but it cannot be guaranteed in general (e.g., due to unregistered inputs in feedback loops). These kinds of algorithms are outside the scope of this paper.

III. PIPELINING ALGORITHMS

The following sections describe the algorithms used for automatic pipeline analysis and pipeline insertion. In practice, the design is first partitioned into its different clock domains and then each clock domain is analyzed and pipelined independently of others. In order to simplify the notations, we assume in the remainder of this section that the current design has only one clock domain.

The first step of pipeline analysis and pipeline insertion consists of estimating the maximal achievable frequency. The second step consists of inserting virtual or real pipeline stages until this frequency is achieved.

A. Estimation of the maximal achievable frequency

This is the maximal frequency of the design that can be obtained with an optimal pipeline strategy. It is defined by the smallest of the two following frequencies:

1) The maximal device frequency. This limit is given by the maximal frequency of the components used by the designs such as DSP or BRAM
2) The maximal frequency within the loops of the design (the loops cannot be pipelined, as explained in the previous section).

There are essentially two ways to measure the maximal frequency within the loops of the designs. The first way is to run the CAD tool on the design without any modification of its constraints and collect the slacks within the loops after place and route. However, this might be too pessimistic since the tool does not typically try to optimize the loops if they do not contain critical paths. The second way consists of disabling all the constraints of the feed-forward paths before running the CAD tool to make sure that it optimizes the frequency of the loops as much as possible. This guarantees that the measuring frequency is an upper-limit of the achievable one.

B. Pipeline stage

Both pipeline analysis and insertion rely heavily on building iteratively stages of pipelined registers. This section provides some background and proposes an efficient method to build pipeline stages.

We model the design by a graph $G = (V, E)$ where the nodes represent the pins of the instances, and the edges represent the connections between pins (either internal arcs between two pins of the same instance or net connections between two pins of different instances). We note $I \subseteq E$ and $O \subseteq E$ the primary inputs and outputs of the design. Figure 3 shows a simple circuit and its associated graph where the internal edges (between two pins of the same instance) are colored in red.

Fig. 3. A small circuit and its abstraction into a directed graph. The edges between pins of the same instances are colored in red

A valid pipeline stage is a set of nodes $S$ such that every path from $I$ to $O$ contains exactly one node of $S$. This property ensures that the added latency is affecting all paths in the same way and does not change the functionality of the design. In practice, $S$ represents the locations where pipeline registers should be inserted, at the input of nodes representing input pins and at the output of nodes representing output pins. A direct consequence is that there cannot be any path between two nodes of $S$ and in particular, no node of $S$ can be inside a loop. Figure 4 shows an example of an invalid and a valid pipeline stage for the preceding circuit.

Definition 1. A pin is qualified as legal if a pipeline register can be inserted on its connected net.

Typically, not all the pins of a design are legal. For example, the pins inside loops (more precisely inside the Strongly Connected Component (SCC) of $G$) are not legal. Also, the...
internal pins such that the inputs of a MUXF7 or MUXF8 are not legal either.

We present below an algorithm to compute a valid pipeline stage. During this process, the pins of the designs are partitioned into 3 sets: stagePins, TFIPins, and TFOPins.

**Algorithm 1:** Computing a pipeline stage

1. Collect all the legal critical pins of the design into legalPinList. A pin is defined as critical if its slack is below the minimal slack within loops.
2. Sort the pins in legalPinList by increasing slack as the primary goal and decreasing slack improvement as secondary goal.
3. Append the remaining legal (non-critical) pins of the designs to the end of legalPinList.
4. stagePins = ∅, TFIPins = ∅, TFOPins = ∅.
5. For each pin p of legalPinList do:
   - If p is already in TFIPins or TFOPins, continue.
   - Add p to stagePins.
   - Add the pins in the Transitive Fan-In (TFI) of p into TFIPins.
   - Add the pins in the Transitive Fan-Out (TFO) of p into TFOPins.
6. return stagePins

In practice, if multiple pins have the same slacks, we need to give a higher priority to the ones that allow the best slack improvement after pipeline insertion. This is achieved by evaluating each critical pin the slack improvement resulting from inserting a pipeline register at its location and using it as a secondary goal for the sort of the step 2.

**Lemma 1.** At the end of the algorithm 1, we have:

1. \( I \cap TFOPins = \emptyset \) and \( O \cap TFIPins = \emptyset \)

2) There are no pin to pin connections between the following pin sets:
   a) TFOPins \( \not\rightarrow \) TFIPins
   b) TFOPins \( \not\rightarrow \) stagePins
   c) TFIPins \( \not\rightarrow \) TFOPins
   d) stagePins \( \not\rightarrow \) TFIPins
   e) stagePins \( \not\rightarrow \) stagePins

**Proof.** The cases 1, 2.a, 2.b, 2.d follow from the TFI, TFO definitions. The case 2.e follows from the fact that as soon as a pin is selected to be in stagePins, its TFI and TFO are excluded from the candidates for stagePins.

The case 2.c is more technical and requires us to make two assumptions. First, we assume that there are no multi-driver nets. Second, we assume a full connectivity between the input and output pins of instances which means that there is always an edge between an input pin and an output pin of the same instance. Assuming full connectivity is not restrictive but limits marginally the possibilities for pipeline.

Let’s assume an edge \( e \) from \( p_1 \in TFIPins \) to \( p_2 \in TFOPins \). Since \( p_2 \in TFOPins \), there is a pin \( p_o \in TFOPins \cup stagePins \) driving \( p_2 \). Similarly, since \( p_1 \in TFIPins \), there is a pin \( p_i \in TFIPins \cup stagePins \) driven by \( p_1 \). Note that we can have \( p_i = p_o \in stagePins \).

There are two possible cases. Either \( e \) is a net connection between a driver pin and a load pin or \( e \) is a connection between an input pin and an output pin of the same instance.

Since \( p_2 \) has at least two input pins (\( p_1 \) and \( p_o \)), \( e \) cannot be a net connection given that we assume no multi-driver nets.

Thus, there is an instance \( I \) with input pins \( p_i, p_o \) and output pins \( p_2, p_i \). Since \( p_o \in TFIPins \), all the input pins of \( I \) must also be in TFIPins because of the full connectivity assumption. This contradicts the fact that \( p_o \not\in TFIPins \).

**Lemma 2.** If all the pins are either in stagePins, TFIPins, or TFOPins at the end of the algorithm 1, then stagePins is a valid pipeline stage.

**Proof.** According to Lemma 1, the only possible connections between pins are:

- TFIPins \( \rightarrow \) TFIPins
- TFIPins \( \rightarrow \) stagePins
- stagePins \( \rightarrow \) TFOPins
- TFOPins \( \rightarrow \) TFOPins

Let’s consider a path \( P = \{p_1, p_2, ..., p_n\} \) between \( I \) and \( O \).

There are three cases to consider, according to Lemma 1:

1) \( p_1 \in stagePins \). According to the possible connections listed above, \( p_2, p_3, ..., p_n \in TFOPins \) so there is exactly one pin of \( P \in stagePins \).

2) \( p_n \in stagePins \). According to the possible connections listed above, \( p_1, p_2, ..., p_{n-1} \in TFIPins \) so there is exactly one pin of \( P \in stagePins \).

3) \( p_1 \in TFIPins \) and \( p_n \in TFOPins \). According to the possible connections listed above, there exist \( k \in \{2, ..., n-1\} \) such that \( p_k \in stagePins \). All the pins before \( p_k \) must be in TFIPins and all the pins after
Algorithm 2: Iterative pipeline flow

input : A netlist in RTL format
output: A routed netlist with extra stages of pipeline registers

1  doSynthesis(netlist);
2  doLogicOpt(netlist);
3  doPlacement(netlist);
4  designSlack ← getSlack(netlist);
5  loopSlack ← getLoopSlack(netlist);
6  while loopSlack > designSlack do
7      insertPipelineStage(netlist);
8      updatePlacement(netlist);
9      updateTiming(netlist);
10     designSlack ← getSlack(netlist);
11     loopSlack ← getLoopSlack(netlist);
12 end
13 doRouting(netlist);

Alternatively, the pipeline stages can be inserted after routing, which requires, in addition, updating the routing after each stage. In our experiments, this does not generally bring a lot of frequency improvement and can significantly increase the runtime. However, this strategy might work well for some particular designs.

In practice, there are several additional cases that need to be handled. For example, if the maximal achievable frequency (as defined in the first section) is lower than the frequency of the loops, we need to use it as our target frequency in Algorithm 2, since it will not be possible to go above. We also need to control the number of additional registers used for pipeline to make sure it does not exceed the number of available registers of the current device. This latter situation is rarely seen in practical designs.

IV. RESULTS

A. Experimental Setup

We evaluate our pipelining approach using an extended version of the Vivado Design Suite [17] version 2016.1. We modified Vivado algorithms to implement pipeline analysis based on the algorithms described above to report the potential impact of extra pipelining. We also implemented a prototype flow to automatically insert pipeline stages into the netlist for subsequent placement and routing. Some of the features described in this paper are not currently available in Vivado, but we are considering their productization in future releases.

Our design suite consists of 93 representative customer designs and Xilinx IP blocks. These designs represent kernels from a wide variety of FPGA applications – from wired and wireless communications to test and measurement to emulation. Their identifying characteristics have been removed from our results. Designs vary in size from 8k to 464k LUTs and the average design contains approximately 130k LUTs. Other characteristics of the benchmark suite are captured in Table I.
All the benchmark designs are implemented (synthesized, optimized, placed, and routed) on the Xilinx UltraScale+ series of devices [15]. We used both Kintex and Virtex devices depending on the application requirements. We chose the minimum-sized device from UltraScale+ device family to accommodate each design. In terms of utilization, we allow up to 90% utilization of LUTs, and up to 95% utilization of FFs and other hard blocks (DSP, BRAM, etc.).

For the designs with multiple clock domains, we calculate FMax improvement using the geometric mean (gmean) of improvement ratios for clock domains with negative setup slack. All reported averages over the benchmark suite are also based on the geometric mean of improvement ratios over all 93 designs in the suite.

### B. Exploring Pipelining Potential

To explore pipelining potential, our methodology is as follows. We first over-constrain each design’s timing requirements to its maximal achievable frequency as described in Section III-A. We do this for two reasons. One, pipelining FMax is limited by the slowest loop’s FMax, so we want to take a design to the loop limit. Two, a design often quickly meets timing after pipelining with the original timing constraints. This is problematic because paths that meet timing are not optimized during Placement and Routing. Because of that, we will never reach the upper-bound of pipelining performance. So, to force the flow for maximum pipelining performance, we tighten the timing constraints to the maximal achievable frequency, which corresponds either to the slowest loop limit or to the maximum device frequency.

As explained in Section III-A, there are two ways to measure the maximal frequency within the loops of the designs. The first way is to measure the timing within the loops after Placement and Routing, which we call the initial loop limit. The second way is to first disable all paths except the ones inside a loop, tighten the remaining constraints as much as possible, and measure the max frequency obtained after Placement and Routing. This measure provides an upper bound of the possible FMax and we call it the tight loop limit.

Figure 6 shows performance potential of extra pipelining. In our suite of 90+ commercial designs, the initial loop limit...
and the tight loop limit demonstrate 18% and 29% FMax improvement on average, respectively. While this performance potential is quite significant, the distribution of FMax gains is bimodal, with roughly half of all designs showing less than 5% potential with initial loop-limit, while the other half demonstrates dramatic performance potential, up to 600% on one of the designs.

To understand why many designs have limited performance potential with extra pipelining, Figure 7 shows the loop-limit of each design on the X axis and the corresponding fraction of sequential elements in loops for the same design on the Y axis. For better visibility, we limited X axis at 250% and excluded design #93 with the highest loop limit of 601%. It is easy to see that the designs with the most potential tend to have less than 25% of their sequential elements in loops. Meanwhile, the designs with greater than 25% of design in loops show much smaller performance potential. This is because the more loops there are in the design, the higher the chance that the critical path is in a loop. If the critical path of the design is in a loop, pipelining cannot improve the FMax of the design, without optimizing the loops.

These results also show that in many cases, loops represent a significant portion of the designs and may be difficult to improve. This highlights the need for a deeper understanding of loop structures and their fundamental properties.

C. Performance Results after inserting extra pipeline stages

Using the timing constraints from above, we run the Pipeline Optimization flow described in Algorithm 2, which iteratively builds pipeline stages until the target frequency is achieved or no more frequency improvement can be obtained. To produce data in time for this submission, we stopped after 16 iterations of pipelining+placement or after the initial loop limit was reached, whichever condition was met first.

Figure 8 shows performance improvement from extra pipelining compared to initial loop limit. The designs are sorted by their initial loop limit potential, but only 44 designs that have at least 5% FMax improvement potential are shown for clarity. Over the entire suite of 93 designs, the gmean FMax improvement is about 9% lower than the theoretical value of the designs.

Only 8 out of 44 designs show a significant gap between initial loop limit and achieved FMax after pipelining. Three of those designs (#27, #35, #36) have their most critical path in an unpipelined DSP cascade. Though UltraScale architecture provides optional pipeline registers in such cascades, our pipeline insertion flow lacked the capability to modify DSP or BRAM cascades and could not improve timing of such paths. Design #44 stopped pipeline cuts prematurely since it exceeded number of available registers in the device. However, we found that after pipelining it contained a lot of back-to-back registers which could be more efficiently mapped to SRL back registers which could be more efficiently mapped to SRL. Design #44 stopped pipeline cuts prematurely since it exceeded number of available registers in the device. However, we found that after pipelining it contained a lot of back-to-back registers which could be more efficiently mapped to SRL back registers which could be more efficiently mapped to SRL. To prevent register overflow, we expect to implement these features in the future which should help to close the remaining gaps as well.

Figure 9 shows performance improvement from extra pipelining compared to tight loop limit. The designs are sorted in the same order as in Figure 8. As expected, none of the designs exceed their tight loop limit. These results also show that after extra pipelining most of the designs either reach their tight loop limit or come very close to it. The designs with the biggest gaps are mostly the same designs that showed gaps with initial loop limit. The average achieved FMax improvement is 19%, which is slightly above the initial loop limit of 18%.

Figure 7. Correlation between ratio of sequential elements in loops and potential FMax improvement with extra pipelining

Fig. 7. Correlation between ratio of sequential elements in loops and potential FMax improvement with extra pipelining

Fig. 8. Achieved FMax after extra pipelining compared to initial loop limit

Fig. 9. Achieved FMax after extra pipelining compared to tight loop limit
upper-bound given by the tight loop limit, and is mostly explained by the limitations of our netlist modification flow, which is quite encouraging.

D. Effect of extra pipelining on latency

Figure 10 shows the impact of extra pipelining on the cycle latency of each design. The designs are sorted in the same order as in Figure 8. Each bar shows how many additional stages of latency (cuts) were inserted into the designs to achieve FMax improvement shown in Figures 8 and 9.

Results show that the amount of added cycle latency is moderate and varies from one to 16 stages. On average, designs required 6 pipelining cuts to achieve the best FMax. There is no strong correlation between achieved FMax and added latency. For example, design #33 achieves 70% speedup with only one cut, while design #6 took 6 cuts to achieve 8% FMax speedup. In general, we find that the amount of extra latency required to reach loop limit is largely a function of design structure rather than design size or performance potential.

E. Effect of extra pipelining on register count

Figure 11 shows the impact of pipelining on register count. As in Figure 8, we sort the designs by Fmax improvement from pipelining and show only the designs that exhibit at least 5% potential improvement. Since the Xilinx UltraScale architectures provide two FFs per LUT, an ideally balanced design would have a 2:1 ratio between FFs and LUTs.

These results show that out of 44 designs which can potentially benefit from pipelining, only 5 exceed 2:1 ratio between FFs and LUTs after extra pipelining are inserted. The ratio after pipelining is around 1.1 on average and it varies from 0.25 up to 9.6 for individual designs. The individual designs with high ratio are DSP based designs with very few LUTs, so even for these designs the number of register after pipelining is not very high.

Note that current implementation of pipelining algorithm does not take advantage of the SRL capability to represent sequentially-connected FFs, and this could significantly reduce the FF count for outlier designs. Based on this, we believe that Xilinx UltraScale devices provide sufficient registers for the majority of commercial applications, while avoiding the costs associated with over-provisioning for too many FFs.

F. Relative performance compared to prior generations of Xilinx FPGAs

The report_pipeline_analysis tool described in this paper can be used on all Xilinx FPGA families supported by Vivado Design Suite to date: 7-series implemented in TSMC’s 28nm process technology, UltraScale devices implemented in TSMC’s 20nm technology, and UltraScale+ generation implemented FinFET+ 16nm process node, also manufactured by TSMC. Figure 12 shows relative performance improvements of baseline and pipelined designs mapped to UltraScale and UltraScale architectures normalized to the FMax of the same applications mapped to Xilinx 7-series product family.

When comparing unpipelined versions of the designs in our suite, UltraScale devices provide about 29% FMax improvement in FMax on average, while UltraScale+ family delivers 78% speedup compared to 7-series generation of FPGAs. Extra pipelining pushes performance benefits to 53% for UltraScale and 112% for UltraScale+. The natural question is how much performance could be extracted with extra pipelining from 7-series generation of devices. Though we did not have an opportunity to run extra pipeline analysis on 7-series in time for this submission, we would expect it to be slightly
less friendly to highly pipelined designs than UltraScale or UltraScale+ and deliver average Fmax increase between 14% to 16%.

The relationship between achieved FMax after pipelining and loop limits is very similar between UltraScale and UltraScale+ architectures. In both cases, achieved FMax is slightly above initial loop limit and below tight loop limit. This indicates that both architectures respond well to highly-pipelined designs, though we expect that UltraScale+ architecture to be superior for high-speed designs due to the introduction of the time-borrowing technology [6].

V. RELATED WORK

Evaluating and optimizing pipeline depth (i.e. number of pipeline stages) and balance (i.e. register position) has been the subject of multiple papers. Retiming, an optimization technique that changes the position of registers, was first proposed by Leiserson and Saxe [9], as a technique to change the placement of registers, depending on different optimization objectives (e.g. minimize cycle time, reduce area, so forth). Leiserson and Saxe model synchronous circuits as directed adjacency graphs (DAGs), where the weights in the edges represent the latency (i.e. number of pipeline stages) in the wires represented by that edge. Nodes are assigned the combinational delay of the logic they represent. The resulting cycle time of the circuit is calculated as the maximum of the sum of delays on any path where sum of the edge weight is zero. Singh and Brown [12] further explore post-placement retiming in the context of FPGAs.

Although changing the position of registers in sequential circuitry is quite trivial, the reduction of cycle time possible is limited by the number of stages already present in the circuit. To further reduce cycle time, the addition of new pipeline stages is required. In conventional synchronous circuits, adding new pipeline stages changes the latency of design, and thus impacts their correctness.

Synopsys® Design Compiler provides some pipeline analysis and insertion capabilities [13]. The pipeline design directive in Synopsys® tools is able to add balanced pipeline stages (number defined by the user) in simple (no path forks or joins) combinational designs. Timing and area is further optimized by performing retiming [13]

Altera’s Stratix 10 FPGA is planning to extensively rely on retiming and pipelining to improve application FMax, as described by Lewis et al. [10]. Stratix 10 routing architecture has been modified to provide optional register elements at every interconnect hop, as well as at the inputs of ALM, DSP, and memory blocks. Altera’s Quartus II design suite has been extended to take advantage of the ubiquitous register elements in the routing, mainly to perform optimal retiming as well as maintain more accurate timing slack aware of subsequent retiming opportunities. Authors seem to indicate that most of the designs reach their loop limit after pipelining, which is similar to our results on Xilinx UltraScale+ architecture.

Elastic circuitry [5] improves over latency-insensitive designs by allowing fine-grained recycling. In this approach, registers are replaced by special units called elastic buffers. Elastic buffers are registers that include handshake logic, a forward valid signal that indicates whether data is valid (as opposed to pipeline bubbles), and a backward stop signal that indicates that a stage is busy and cannot take new data. Special fork and join operators are used when required in the data path. The pipelining algorithm explores the design space by adding additional stages after memory components in the circuit. This recycling approach has the advantage of being fast, since insertion points are clear and well known. A final retiming step is used to improve the balance across stages, and thus the circuit frequency. In this scenario it is possible to pipeline sequential loops, but with throughput degradation or or requiring RTL changes [14].

In High-Level Synthesis (HLS), circuits are described in a High-Level programming language, such as C, where timing information is not available. Then, scheduling algorithms decide how many pipeline stages will be placed in the final circuit and their positions. Scheduling in HLS has the advantage of being able to pipeline sequential loops. For instance, Chao et al. [4] propose a heuristic-based approach that incrementally improves the scheduling. The algorithm is capable of retiming and reducing the pipeline depth in loops, but not increase them. Thus it partially performs recycling, but only to reduce the latency.

Recycling in regular synchronous circuits in the RTL level has also been explored [8]. In such cases, feedback signals are a concern, since changing the latency in such signals may lead to functional errors. The approach proposed by Kim et al [8] requires the designer to indicate which feedback paths are eligible to pipeline, since it is not possible for the automated flow to figure this out. This approach is based on cuts, i.e. splitting the DAG that represents the connections between register into two. At each iteration in the algorithm, the current critical path is selected to cut (roughly at its mid-point delay). All paths that are parallel to the critical path also need to be inserted in the cut to maintain correctness. Since the cut in the parallel paths can be done at multiple points, three possibilities are considered: an exhaustive approach, and two heuristic-based approaches. The heuristic-based approaches use a levelized graph, the mid-cut heuristic picks the level that splits the critical path delay in roughly half, and the multi-cut chooses between all the possible levels. In the simple circuits analyzed, the heuristic-based approaches yielded good results (close to optimal) with great speed-up [8]. The scaling to larger designs was not well explored, but leveling the gate graph has linear cost. It is unclear, however, how good the cuts would be in real-life designs.

Ideal cuts can be found through a min-cut algorithm. A fast algorithm for min-cut based on binary maximum-flow [7] has been explored. The binary approach has been shown to be 5 times faster than non-binary max-flow approaches [7]. Regular max-flow approaches are based on weighted DAGs, and the algorithm time is heavily dependent on the weights in the edges. The simplification to the binary case (either the edge has a weight 1 or 0, i.e. there is no edge) simplifies the run-
time to $O(RE)$, where $R$ is the number of registers initially in the graph and $E$ is the number of edges. Although the authors propose the use of binary max-flow in the case of retiming [7], the approach can be easily adapted to the insertion of extra pipeline stages.

In the context of FPGAs, latency-insensitive protocols have been evaluated to investigate recycling benefits [11]. It has been shown that Fmax can be considerably improved by the addition of extra pipeline stages. But even though the latency-insensitive approach had a small overhead, authors argue that the addition of pipeline stages in synchronous designs is error-prone [11]. Although the authors quantify area overhead in FPGAs, adding registers is relatively cheap [2], since they seem to be underused, if compared to logic [3].

Even though pipelining is usually considered to be a method to improve performance, the reduction of power consumption by pipelining in FPGAs has been explored [2], [3], [7]. This is a somehow surprising result, since one could think that the increase in the number of registers would increase the power. Nevertheless, adding extra pipeline stages can reduce logic depth, and thus reduce glitches. Glitches add to switching power, since they cause unnecessary switching. It has been shown that glitches have a domino effect [2], [3], i.e. they propagate until a flop is reached. Breaking the combinational paths by adding flops will also reduce the magnitude of such a domino effect [3]. This issue is more significant in FPGAs, since FPGAs tend to use gates with more inputs (4-6 input LUTs), while ASICs use mostly 1-3 input gates. Even though adding pipeline stages can reduce the power by as much as 66% [3], over-pipelining can end up increasing overall power consumption [3]. Thus, in terms of power, it seems to be reasonable to add extra stages as long as extra FPGA slices are not needed.

VI. CONCLUSIONS

This paper describes the methodology and algorithms behind extra pipeline analysis tools released in Xilinx Vivado design suite version 2015.3. These tools allow FPGA users to explore latency vs. performance trade-offs of their designs before investing time and effort into modifying RTL. We achieve this by automatically classifying all design paths into feedforward (pipelinnable) and loop (non-pipelinnable) paths, and performing consecutive cuts of the entire design to find out the best improvement in Fmax per extra stage of pipelining. In addition to pipelining critical paths, it also makes sure that all parallel non-critical paths are pipelined to preserve correctness of the dataflow. After analysing the design in this manner, our pipeline analysis tool presents detailed information on trade-offs between extra levels of pipelining, achievable frequencies, and number of registers required to implement those changes.

Our performance analysis of loop limits reveals that extra pipelining can potentially improve performance by 18%-29% on average on a set of 90 representative commercial designs and Xilinx IP blocks. We also show that the distribution of performance improvements is bimodal, with roughly half of the benchmark suite showing performance potential of more than 5%, while the other half have most critical critical paths in loops and cannot be improved with pipelining alone.

We further evaluate how well highly-pipelined designs map to Xilinx UltraScale+ and UltraScale FPGA architectures by inserting extra registers into the designs with pipeline potential, based on the cut schedule suggested by report_pipeline_analysis tool, and measure Fmax improvement after re-placing the design. Results show that extra pipelining improves Fmax by 19% on average and final performance meets or exceeds initial loop limit on the majority of designs. The few designs with significant gap between loop limit and achieved Fmax are explained mostly by the limitations of our insertion flow which cannot modify combinational DSP cascades, which we plan to address in the future. In addition, we find that extra pipelining increases register count by about 60% on average and in most designs the area does not grow due to sufficient supply of available registers in UltraScale+ FPGAs.

REFERENCES