Low-level Loop Analysis and Pipelining of Applications mapped to Xilinx FPGAs

Hossein Omidian
Xilinx Inc.
San Jose, CA, United States
hosseino@xilinx.com

Guy G.F. Lemieux
University of British Columbia
Vancouver, BC, Canada
lemieux@ece.ubc.ca

***Abstract*—In this paper we investigate using low-level loop analysis to identify common loop patterns in the netlist generated by the synthesis flow and use loop optimization techniques to increase Fmax of applications implemented on Xilinx FPGAs. Ordinarily, feed-forward paths in the netlist can be easily pipelined. The focus of this study is only sequential loops (with feedback cycles) that are more challenging to optimize. We show that, using low-level loop analysis, we can improve Fmax on average by 57% and achieve an average Fmax of 714MHz across seven industrial designs. Using aggressive loop combining, we also show that we can save 18% area on average while still improving the Fmax by 15% to 41% on four of the seven designs.**

***Keywords*-Low-level Loop Analysis, FPGA, Pipeline, Loop Transformation**

I. Introduction

The logic synthesis process sometimes results in slow logic structures at the netlist level that reduce Fmax. If these slow structures appear in feed-forward logic, pipelining is easy and one of the most effective ways to improve performance. Pipelining improves Fmax by introducing registers on the most critical paths and reducing the effective number of gates and wire distance that a signal has to traverse in one cycle. Figure 1 shows a simple example of how pipelining a feed-forward path works. Figure 1a shows the original design with 3 levels of LUTs between to two registers. The delay for the most critical path for this design is 6ns, which corresponds to 166MHz. As shown in Figure 1b, it's possible to improve the performance of this simple design by inserting two extra registers between the LUTs. This reduces the critical path to 2ns and increases Fmax to 500MHz. To preserve functional correctness with feed-forward logic, pipelining requires inserting an equal number of register stages on all of the parallel paths so they are balanced. Ganusov et al., proposed an automated approach to pipeline feed-forward paths using pipelining methodologies [1]. They also showed the bottleneck for some designs are sequential loops which cannot be easily pipelined.

In contrast to feed-forward paths, slow structures appearing inside a sequential loop cannot be easily pipelined. For example, Figure 2 shows how inserting a register in a loop path breaks the sequence and functionality of the loop. The original loop in Figure 2a shows a counter which is initialized to zero and increments the output value by 1 each clock cycle. Figure 2b illustrates how pipelining of the counter loop will

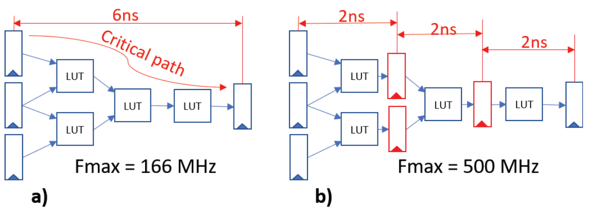


Fig. 1. Simple feed-forward pipelining.

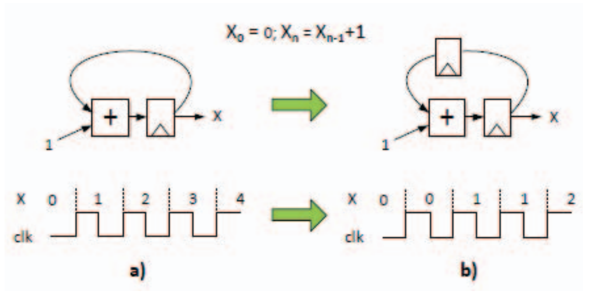


Fig. 2. Pipelining a sequential loop breaks functionality.

produce a different sequence of values from the original design – each count value is produced twice. Fixing this requires a more complex transformation than simply inserting flip-flops. In this case, it may be possible to split the logic into separate odd and even loops, where each loop increments the previous value by 2, and the results are merged on alternate cycles. This allows the merged output rate to be double each of the loop rates.

Previous studies such as Borch et al. show the impact of micro-architectural loop structures in system-on-chip designs [2]. Our own experience has shown that 80% of industrial designs on FPGAs run below device capability because of slow loops in the designs. These studies suggest that sequential loops are quite commonplace, so a strategy is needed to find them and transformations are needed to improve them.

There have been several studies on different loop optimizations to achieve highest Fmax and throughput. Techniques such as loop splitting and similar transformations can be used to optimize some designs with loops [3], [4]. In these approaches, the user needs to spend considerable time analyzing the design and manually parallelizing and pipelining it as much as possible. In contrast, another approach is to use an automated design space exploration tool to investigate space/time tradeoffs. For example, the JANUS system [5]) explores many design parameter settings embedded into a C program compiled using Vivado HLS; it finds the best parameter settings to meet the user target (throughput or area) while optimizing for the other (area or throughput).

These aforementioned approaches are useful for optimizing loops in the designs. Unfortunately, they both require some degree of expertise to implement in the high-level source code or at the RTL level. Moreover, in our experience, even after using these mentioned techniques, a netlist produced from synthesis can still contain many timing-critical sequential loops.

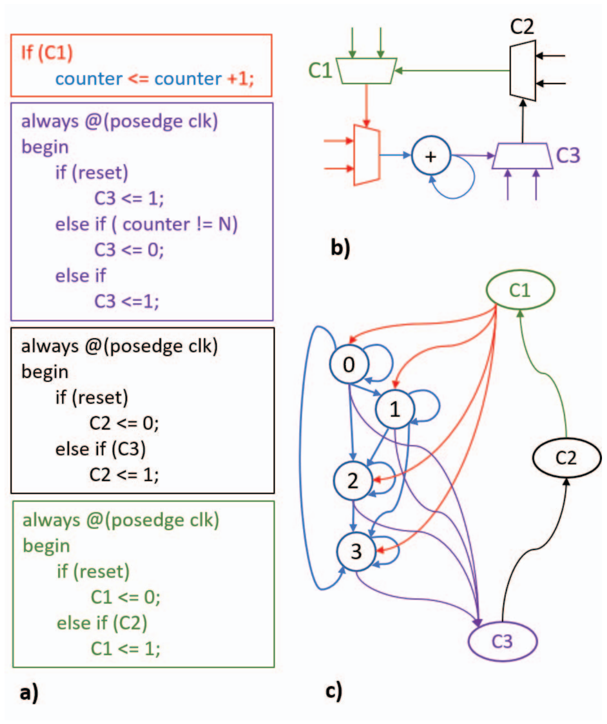
In this paper, we add loop analysis to the physical optimization stage in the synthesis process and automate low-level loop optimization for FPGAs. All of the designs we examine have already gone through an aggressive physical synthesis optimization provided by Xilinx Vivado, yet we are able to find significant additional improvement in delay and area. This assures us that we are working on loops that cannot be found by the state-of-the-art industrial tools.

More specifically, this paper focuses on sequential loops in industrial designs and applies or proposes different netlist-level transformations to increase overall Fmax or throughput. We analyze seven large industrial designs, ranging from 150,000 to 300,000 LUTs in size, and find specific loop patterns that benefit from additional transformation and optimization. We present four of these patterns found and show how to fix them in detail. Fixing may involve automated netlist modification, hints to the user for manual transformation, or automated insertion of compiler directives. Just as software compilers clean up lot of code to allow designers to write clear and readable code instead of optimized code, we believe synthesis tools need to clean up slow hardware structures.

II. Low-level Loop Optimization Methodology

Synthesis tools transform a design from its original source form into a logic circuit represented as a netlist. A netlist shows how different FPGA logic resources (e.g., LUT, FF, BRAM and DSP) are connected. Logic resources in the netlist can be part of feed-forward paths or sequential loops.

As a motivational example, Figure 3 shows a simple loop generated by a synthesis tool; this was extracted from one of the industrial designs we examined. Figure 3a shows a portion of the Verilog code representing the loop. Figure 3b shows an approximate logic schematic where the if-statements are represented by MUXs. Figure 3c shows a graph representation of the same loop in the netlist which was generated by the synthesis tool. In this case, a counter is shown in blue. Each node in the counter graph is representing a count value. Each if-statement was implemented as a LUT (nodes C1, C2 and C3 in the graph). This example shows that *four completely different parts of the source code can result in a low-level loop*. In the industrial circuits we examined, an average of



```
If (C1)
    counter <= counter +1;
```

```
always @(posedge clk)
begin
    if (reset)
        C3 <= 1;
    else if ( counter != N)
        C3 <= 0;
    else if
        C3 <=1;
```

```
always @(posedge clk)
begin
    if (reset)
        C2 <= 0;
    else if (C3)
        C2 <= 1;
```

```
always @(posedge clk)
begin
    if (reset)
        C1 <= 0;
    else if (C2)
        C1 <= 1;
```

Fig. 3. Counter with a loop controller

46% of the logic is inside sequential loops. Moreover, in most cases, we also observed that those loops are the main reason that prevents us from achieving higher Fmax.

Below, we discuss fixing four types of loop patterns.

A. Overview

First, we classify all paths as either feed-forward (which can be easily pipelined) or loop (which are not easily pipelined). Then, we analyze the loops and focus on the critical ones that are amenable to optimization techniques.

B. Finding Loop Patterns

We created a tool to find loops in the netlist and simplify them by finding primitive loops inside big independent loops. This make the loop pattern simple and understandable to developers. Simplified loops can be used by application developers (both RTL level and HLS level) to see the low-level effect of their implementations and gives them hints to avoid complex loop patterns. Hence, identifying common loop patterns is the primary objective of this tool. The second objective is to automatically modify the netlist to optimize simple loop patterns. If that is too difficult, then a message can be printed instructing the user where to insert directives to HLS implementations, or how to modify RTL, to facilitate the desired pipelining speedup.

As mentioned before, it's not possible to simply add pipeline registers to a sequential loop; more sophisticated transforms need to be used to improve the Fmax. First we synthesize the design and tag the well-known structures such as counters.

This can be done using compiler information from the high-level code. Then we run logic optimization to make sure the design is optimized in order to get the highest Fmax. After that we place and route the design on an FPGA in order to get the accurate timing and Fmax results for the design. Using a loop classifier, we classify the feed-forward logic and sequential loop logic and do the loop analysis. For feed-forward logic, we use automated pipeline analysis to get the independent Fmax results for feed-forward logic and make sure feed-forward logic is not the bottleneck. Then we run timing analysis for the design to find a set of critical paths. If the sequential loop logic is the bottleneck of the design, we start the loop analysis process.

The first step of loop analysis is finding loop patterns in a netlist, tagging them and simplifying big loops by finding primitive loop patterns inside them. To do this, first we classify the tagged components in the loop and run exhaustive sub-graph isomorphism algorithm to find primitive patterns in the loop and simplify the big complex loop. Finally we add the simplified loop patterns to our loop pattern library. Algorithm 1 shows the finding loop patterns process.

```
Input:  A netlist
Output:  A set of simplified loop patterns
 1: do_Synthesis(netlist)
 2: tag_Components(netlist)
 3: do_Logic_Optimization(netlist)
 4: do_Place_and_Route(netlist)
 5: (Feedforward,Loops) ← loopClassifier(netlist)
 6: FeedForward ← PipelineAnalysis(FeedForward)
 7: Critical_Paths = Timing_Analysis(FeedForward, Loops)
 8: if Critical_Paths are on Loops then
 9:     for all loop in Loops do
10:         //Finding primitive patterns for each loop
            Classify_Tagged_Components(loop.netlist)
            Subgraph_Isomorphism(loop.netlist,Loop_Library)
11:         Simplified_pattern ← Collapse_Small_Patterns
12:         Loop_Library ← Simplified_pattern
13:     end for
14: end if
```

Algorithm 1: Finding Loop Patterns

Moreover, finding loop patterns and simplifying loop patterns makes loop visualization easier for the user and eventually gives users hints on how their implementations generate different loops and which part they need to optimize to avoid complex loop patterns. Figure 4 shows a loop pattern before finding a primitive loop within it. Figure 5 shows the same loop after simplifying. Figure 5 can be easily read and used by application developers.

C. Loop Analysis

As discussed before, we need to do more than simple pipelining for sequential loops. After finding different loop patterns, we need to use various approaches (e.g., loop

Fig. 4. Simple loop expanded

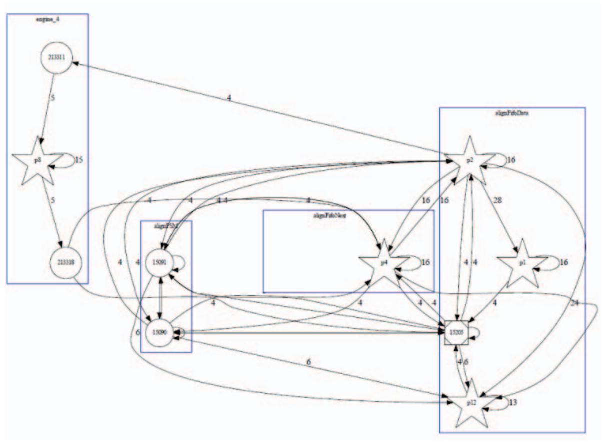


Fig. 5. Simplifying loop by collapsing inside loop patterns

cutting, loop transformation techniques and loop pipelining-combining). In all those cases, we need to make sure these approaches don't change the functionality of the application. Below, we show three simple common loop patterns and discuss loop optimization approaches used to improve the loops in the design.

Loop splitting. Figure 6a shows the simplified graph representation of the example mentioned in Figure 3. Assuming

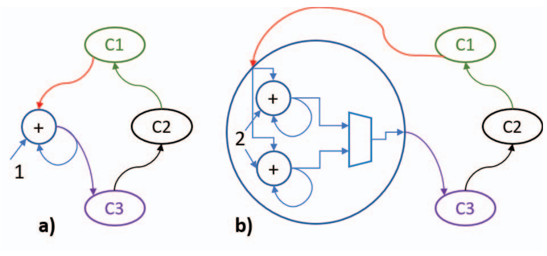


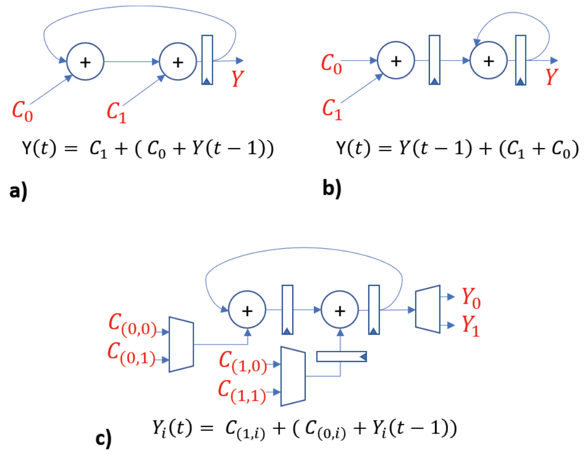
Fig. 6. Loop splitting example

the counter is on the critical path, we need to improve it in order to eliminate it from the critical path. As shown in the example in Figure 2, pipelining the counter changes the sequence of the counter as well as its functionality. Instead, loop transformation techniques such as loop splitting/unrolling can be used. Figure 6 shows the counter split into two separate loops (odd and even), each of which increments by 2. In this case, the Fmax of each counter remains the same, though the output of combined node (big node containing two counters) can be sampled at twice the speed since two values are generated per clock cycle. This eliminates a loop as a critical path and potentially allows the Fmax for C1, C2 and C3 to increase by a factor of two. This loop optimization technique may increase area, and it can be easily automated.

Some might argue that splitting the counter into two counters can be done at the implementation stage by the developers. There are three main reasons that handling this in a tool can be beneficial. First, in all the design we explored, we observed that loops can be generated from different parts in the whole RTL project. Developers might not be aware of how those loops are generated. Moreover, traditional logic optimization techniques add another layer of obfuscation on how loops are generated in the netlist. Second, each part of the RTL project can be owned by different developers, so doing this manually requires more analysis and communication during development. Third, to split a counter manually, developers need to add one more clock to the design manually. This adds one more constraint to the design which restricts the synthesis, placement and routing processes. On the other hand, the tool is aware of available clocks and where the design is placed, so handling counter splitting can be done easily by the tool.

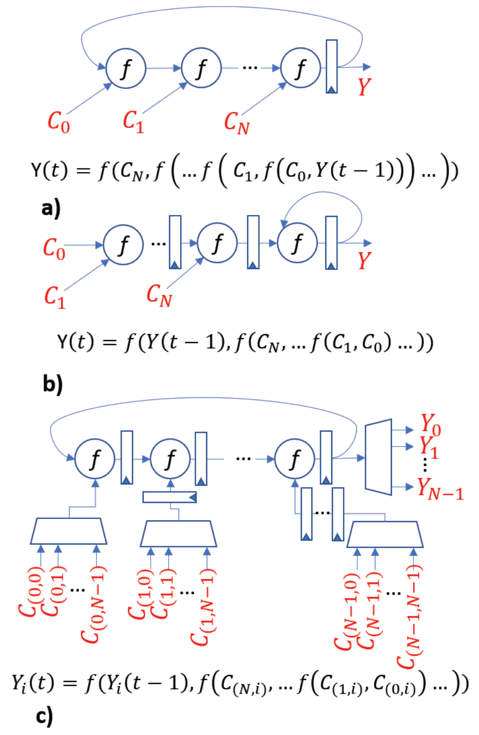
Associative Refactoring. Figure 7a shows another simplified common loop we observe in a different industrial design. The pattern is similar to an adder chain with a feedback loop. Since the add operation is associative, we can simply change the order and transform the loose loop in Figure 7a to an adder chain with an accumulator at the end (a tight loop) shown in Figure 7b. This transformation can improve the Fmax by a factor of two. Moreover, this transformation can be done for any associative function. This loop transformation approach can be easily automated.

Combining. Another approach can be adding pipeline stages inside the loop to increase the frequency, similar to C-slow retiming [6]. As mentioned before, simply adding pipeline stages inside the loop can change the sequence of the loop while improving the Fmax. For example, if we add a register between two adders in figure 7a, we increase the Fmax by factor of two but the new circuit only generates one valid output every two clock cycles. This means the throughput has not improved. Although it might look we haven't improved the loop by adding a pipeline stage, we have opened an opportunity to *combine* two isomorphic loop patterns and generate a valid output for each of them on alternate clock cycles. Figure 7c shows a *combined-pipelined* loop. Every clock cycle, it generates one output for each loop in a round-robin manner. This can save area by implementing



C_0
C_1
Y
$Y(t) = C_1 + (C_0 + Y(t-1))$
a)
$Y(t) = Y(t-1) + (C_1 + C_0)$
b)
$C_{(0,0)}$
$C_{(0,1)}$
$C_{(1,0)}$
$C_{(1,1)}$
Y_0
Y_1
$Y_i(t) = C_{(1,i)} + (C_{(0,i)} + Y_i(t-1))$
c)

Fig. 7. Transforming or pipelining an adder chain



C_0
C_1
C_N
Y
$Y(t) = f(C_N, f(\ldots f(C_1, f(C_0, Y(t-1)))\ldots))$
a)
$Y(t) = f(Y(t-1), f(C_N, \ldots f(C_1, C_0)\ldots))$
b)
Y_0
Y_1
Y_{N-1}
$C_{(0,0)}$
$C_{(0,1)}$
$C_{(0,N-1)}$
$C_{(1,0)}$
$C_{(1,1)}$
$C_{(1,N-1)}$
$C_{(N-1,0)}$
$C_{(N-1,1)}$
$C_{(N-1,N-1)}$
$Y_i(t) = f(Y_i(t-1), f(C_{(N,i)}, \ldots f(C_{(1,i)}, C_{(0,i)})\ldots))$
c)

Fig. 8. Transforming and pipelining a function chain

one pipelined loop N times faster instead of N slow, separate loops. This is the only technique we discovered that may save area. It can be automated by pattern matching.

The combined-pipelined adder loop just described can be applied in more general situations. Figure 8a shows a big function chain with feedback. If the function f is associative, it's possible to transform it into a function chain with a tight loop feedback at the end as shown in Figure 8b. Regardless of whether the function f is associative or not, we can use pattern matching to find N similar patterns and use combining

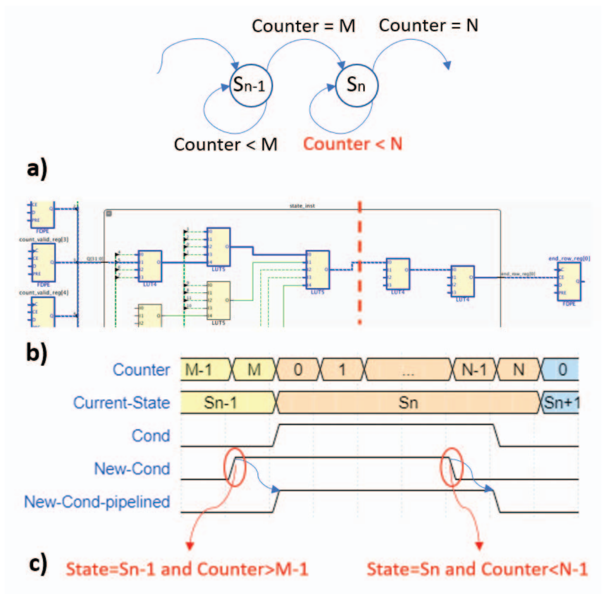


Fig. 9. Pipelining a loop inside an FSM by changing a condition

to improve the throughput and save area.

Early Conditions. A fourth common pattern we will discuss, called early conditions, is shown in Figure 9a. It's a part of simple FSM where each state waits for fixed amount of time using a counter. For example, state S_n initializes its counter to zero and waits in the same state while counter is less than N. With big counters, conditions such as $Counter < N$ following by initialization of the current state and next state often generates several levels of logic which slows down the circuit. Figure 9b shows the netlist generated by the synthesis tool. There are two registers at both ends and there are five LUTs in between. Three LUTs at the left side of the dashed line are generated to implement the condition logic and two LUTs at the right side of the dashed line are generated to implement the initialization logic. It's possible to increase the Fmax by adding a pipeline stage between condition logic and initialization logic, but this would generate the final condition one clock cycle late.

Instead, we can generate the condition one cycle early. Figure 9c shows the waveform of original condition, current state and counter. To correct the circuit, we add a new condition, computed one cycle early. After a pipeline delay, this can be combined with the initialization logic. Figure 9c shows "New-Cond", which is $((State = S_{n-1}$ AND $Counter > M-1)$ OR $(State = S_n$ AND $Counter < N-1))$; after a one cycle pipeline delay, this is equivalent to the original condition. In our experience, these types of patterns can be automated for simple FSMs, but more complex cases likely require manual modifications by experts. However, even in the complex cases, an analysis tool can give developers hints on how to modify FSM timing.

These four sequential loop patterns can be detected and pipelined to improve clock speed. In the next section, we describe the results of optimizing 11 sequential loop patterns in seven large industrial designs.

III. Experimental Results

Our experiments are carried using Xilinx Vivado tools and UltraScale+ devices. We have developed a tool that analyzes a post-synthesis netlist and its timing analysis to find critical sequential loop patterns. Depending upon the loop pattern, the tool either automatically modifies the netlist or suggests modifications to be made to the source. When modifications are suggested, we make the changes by hand. After all changes are made, we produce an optimized netlist and use timing analysis to determine the final Fmax.

TABLE I
Types of Loop Patterns Detected

Pattern	Method	Automated Mod.
1. Counter	Splitting	Yes
2. Combinational chain	Refactoring	Yes
3. Combinational chain	Combining	Yes
4. FSM	Early condition	Some
5. Triangle Accumulator	Cutting	Yes
6. Combined counters w/ controller	Cutting	Yes
7. Counter with a controller loop	Cutting	Yes
8. CRC lookalike	Combining	Yes
9. FIFO chain	Cutting	Yes
10. FIFO with read/write pointer	Cutting	No
11. FIFOs with a controller	Cutting	No

We examined seven industrial designs ranging from approximately 150,000 LUTs to 300,000 LUTs. In those seven designs, we identified more than 50 loop pattern instances. Among those instances, we found the 11 distinct pattern types listed in Table I. The first four pattern types are described in the previous section; page limits prevent us from discussing the remaining seven patterns, but most of them are resolved by cutting the loop with a register and transforming the logic to restore correctness in ways similar to those already described. All 11 of these patterns can be detected by our analysis tool. For many of these patterns, the netlist can be automatically modified and output in optimized form. In some cases, the tool only gives a hint of the changes needed, and relies upon the user to modify and recompile the netlist source code.

After running the tool on each of the industrial designs, we listed the optimization types that were applied to each

TABLE II
Loop optimization methods used for each design

Design	Loop optimization methods
design_1	Cutting, Splitting, Refactoring
design_2	Cutting, Splitting, Refactoring
design_3	Refactoring, Early condition
design_4	Cutting, Splitting
design_5	Cutting
design_6	Cutting, Splitting
design_7	Cutting, Splitting, Refactoring

TABLE III
LOGIC LEVELS IMPROVEMENT AFTER USING LOW-LEVEL LOOP ANALYSIS

Design	Logic levels before	Logic levels after
design_1	4	3
design_2	4	3
design_3	5	4
design_4	13	3
design_5	6	3
design_6	5	3
design_7	7	3

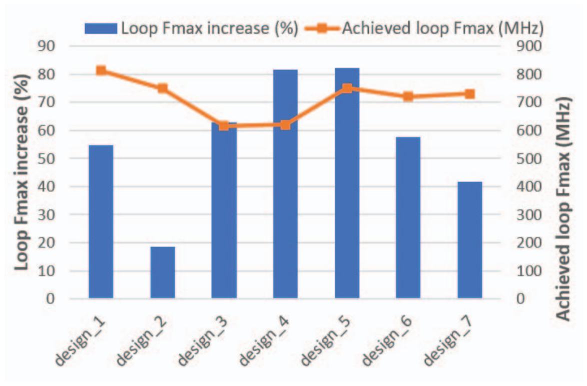


Fig. 10. Fmax results

circuit in Table II. In addition, Table III shows the (worst-case) number of logic levels before and after using the tool. As we can see, the tool is able to reduce logic depth in all designs and results in the higher Fmax shown in Figure 10. Here, the percentage increase in Fmax is shown on the left y-axis (bar graph), whereas the final optimized Fmax is shown on the right y-axis (line graph). In these results, the tool is focusing on Fmax improvement only, so it does not apply pipelining and combining in cases where it doesn't improve timing. The tool improves Fmax between 19% and 82%, or 57% on average. In absolute terms, it achieves an Fmax of 714MHz on average.

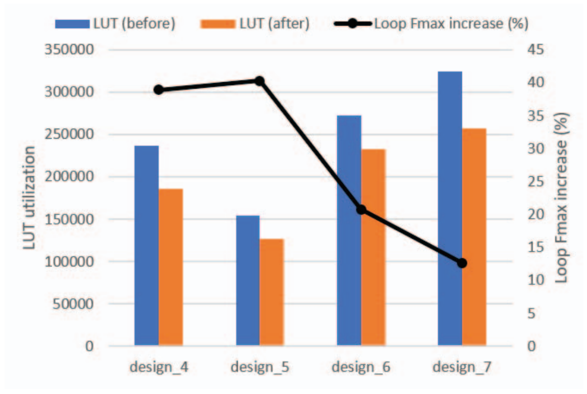


Fig. 11. Saved area and achieved Fmax using aggressive combining

In four of the seven designs, the tool found multiple instances of a loop pattern similar to Figure 8. (In the other three designs, this loop pattern was not found, so the results are unchanged from the previous figure.) Using pattern matching, the tool can automatically identify these instances and use the pipelining and combining technique to save area. This time, we run the tool with the goal of saving as much area as possible, possibly at the expense of improving Fmax. Figure 11 shows the amount of area saved in these four designs; on average, 18% area is saved. In addition, Fmax has also improved between 15% and 41%. While the Fmax improvement is not as high as before, it is sometimes necessary to save area as well.

IV. Conclusion

In this paper we investigate using low-level loop analysis on a netlist to identify common loop patterns generated by synthesis. While feed-forward paths are easily pipelined, sequential loops require greater care. We have identified 11 types of patterns that can be accelerated among seven industrial designs. The paper describes four of these patterns and the required optimizing transformations: splitting, refactoring, combining, and early conditions. These methods are implemented in a netlist analysis tool that either patches the netlist, or tells the user how to modify the source. When applying this tool and optimizing for Fmax, we were able to improve Fmax by an average of 57% and achieve an overall Fmax of 714MHz. When applying this tool and optimizing for area, it saves an average of 18% of LUTs and still get 15% to 41% Fmax improvement. All of these design modifications can be made at the source level, but in many cases it would result in unreadable and hard-to-maintain source code. It is also worthwhile to note that these improvements were made after the Xilinx physical synthesis process; Vivado was unable to make the same types of transformations.

References

[1] I. Ganusov, H. Fraisse, A. Ng, R. T. Possignolo, and S. Das, "Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–10.

[2] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *International Symposium on High Performance Computer Architecture*. IEEE, 2002, pp. 299–310.

[3] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop splitting for efficient pipelining in high-level synthesis," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 72–79.

[4] S. Dai, G. Liu, R. Zhao, and Z. Zhang, "Enabling adaptive loop pipelining in high-level synthesis," in *Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2017, pp. 131–135.

[5] H. Omidian and G. G. Lemieux, "Janus: A compilation system for balancing parallelism and performance in OpenVX," in *Journal of Physics: Conference Series*, vol. 1004, no. 1. IOP Publishing, 2018, p. 012011.

[6] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement c-slow retiming for the Xilinx Virtex FPGA," in *International Symposium on Field-programmable Gate Arrays*. ACM/SIGDA, 2003, pp. 185–194.