An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits

Carmine Rizzi*, Andrea Guerrieri†, and Lana Josipović*
*ETH Zurich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland
†Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247686

***Abstract*—Dataflow circuits promise to overcome the scheduling limitations of standard HLS solutions. However, their performance suffers due to timing overheads caused by their handshake communication protocol. Current pipelining solutions fail to account for logic optimizations that occur during FPGA synthesis, thus producing over-conservative results. In this work, we develop an FPGA mapping-aware timing regulation technique for dataflow circuits; it relies on FPGA synthesis information to identify the circuit's critical path and optimize it through register placement. Our dataflow circuits Pareto-dominate state-of-the-art solutions, with up to 29% and 21% execution time and area reduction, respectively.**

I. Introduction

HLS-produced circuits typically rely on pre-characterized timing information to control the circuit's critical path [1]. So far, HLS tools that produce dynamically scheduled, dataflow circuits from C/C++ code [2, 3] relied on the same strategy: the delay of each unit is obtained by synthesising, placing, and routing it in isolation, and this information is then taken into account when pipelining the circuit with registers (i.e., buffers) [4, 5]. However, this approach fails to account for the cross-unit optimisations and simplifications that occur during FPGA synthesis, placement, and routing, thus causing several undesired effects: (1) The overestimated dataflow unit latencies may cause conservative pipelining and unnecessary resource overheads by placing redundant buffers and preventing logic optimizations across buffer-separated pipeline stages. (2) The same conservative pipelining may unnecessarily lower the throughput and, thus, performance, if a buffer is redundantly placed on a throughput-critical cycle. (3) Placement and routing may introduce long and unpredictable combinational paths that cause the frequency to deviate from the target [6].

In this work, we devise a buffer placement strategy for dataflow circuits that considers the effects of FPGA synthesis to produce timing-efficient dataflow designs. Our strategy iteratively explores how dataflow units synthesize into FPGA constructs (i.e., LUTs); it identifies circuit portions that form single structural units and guides the timing optimizer such that particular logic structures remain intact during buffer placement. On a set of benchmarks obtained from C code, we show that the resulting dataflow circuits systematically achieve Pareto-optimal solutions whose timing is significantly improved with respect to state-of-the-art HLS frequency regulation approaches.

II. The Need for Mapping-Aware Timing Regulation

Figure 1 shows a portion of a dataflow circuit, consisting of an interconnect of three dataflow units: the forks dispatch data to multiple successors and the join synchronizes its inputs before producing a *token* for the successor. The units communicate with fine-grain handshake signals to indicate their readiness to accept data from their predecessors as well as the validity of their data to the successors. The longest combinational paths through the shown units (containing logic to compute the *ready* and *valid* handshake values) are the coloured paths through all three units.

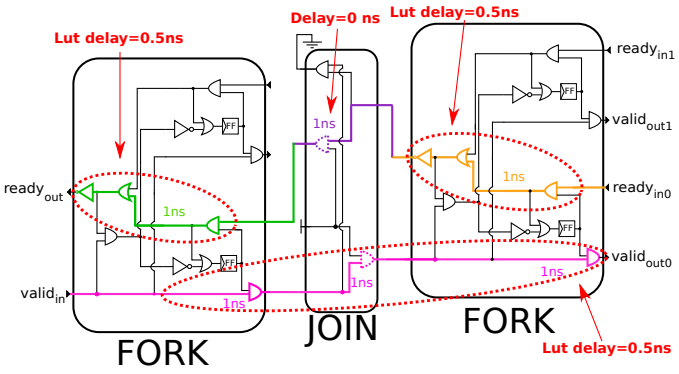

Figure 1: Precharacterized timing information in HLS may be inaccurate and conservative, thus leading to suboptimal area and performance. In this example, a timing model based on pre-characterized combinational delays would assume a delay of 3 ns through the coloured paths. The top path that belongs to the left fork, join, and right fork shown in green, purple, and orange is synthesized into only 2 LUTs (dashed red) with a total delay of 1 ns. Similarly, the bottom pink path is synthesized into only one LUT, with a delay of 0.5 ns.

When measured in isolation, the portions of these combinational paths within the forks and join have a delay of 1 ns each. However, when implemented on an FPGA, these paths are first merged and simplified using logic synthesis (e.g., the dashed AND gate in the join is removed), and then mapped onto FPGA Look-Up Tables (shown in red dashed); the delays of the resulting constructs are completely different than the ones specified at RTL level (in this example, we assume a delay of each LUT to be 0.5 ns, thus, the total delay of the shown paths after implementation is only 1 ns for the top path and 0.5 ns for the bottom one). Clearly, assuming the pre-characterized operator delays when optimizing the circuit's timing would result in conservative assumptions: it may require breaking these paths with buffers that are not actually needed. These buffers would not only be an expensive overhead but may also lower the circuit's throughput and completely prevent logic optimizations such as the ones in the figure, without any improvement of the circuit's frequency.

Although intuitively clear, understanding the delay effects of logic synthesis in dataflow networks is not straightforward due to the complex interactions between the datapath operators and bidirectional handshake signals [5]. What is more, systematic buffer insertion that considers this information is challenging, as the placement of a buffer may affect possible synthesis optimizations and invalidate the prior topological assumptions. We tackle this problem in this paper: we propose an iterative buffer placement that considers the mapping of a dataflow circuit to FPGA constructs and gradually adds buffers to strategic positions in the circuit, such that their effect on logic optimizations is accounted for and minimized. In Section III, we outline what others have done before us to regulate the timing of HLS-produced circuits. In Section IV, we discuss our methodology

to understand the synthesis and mapping of the dataflow circuit to the underlying FPGA constructs. In Section V, we present our algorithm for step-wise buffer insertion that accounts for the circuit's mapping. We show the superiority of the circuits optimized with our strategy over state-of-the-art solutions in Section VI.

III. Background and Related Work

Classic HLS scheduling assigns operations to particular time steps; it uses techniques such as modulo scheduling to pipeline and retime the circuit [7, 8]. Many scheduling approaches can be solved iteratively by rescheduling operations to improve the target cost function (e.g., throughput, latency) under particular constraints [9]. More recent HLS works include synthesis information into the scheduling problem to improve timing estimation and accuracy [1, 10]. None of these algorithms is applicable to dataflow circuits, where no static schedule is devised and the exact time in which each operation executes is determined at runtime.

The works closest to ours are those of Tan et al. [1] and Zheng et al. [11]. Tan et al. propose a scheduling approach that accounts for the mapping of original operations onto LUTs; in contrast, we consider the operation mapping after logic synthesis and optimization for more accurate frequency regulation. Zheng et al. account for placement and routing; their strategy is iterative, just like ours, but its starting point is a conservative register placement based on pre-characterized delays which we entirely omit. The same work discusses mapping ambiguities in the presence of shared registers; we tackle this problem in a more general way in Section IV-A. Neither of these approaches directly applies to dynamic scheduling, which is our primary target.

Complementary works focus on the problem of estimating routing congestion and interconnect delays [12, 13]; although this aspect is orthogonal to our contribution, our approach could be enhanced by these techniques to reduce the effect of routing on the achieved critical path, as we will discuss in Section VI.

Several works explore mathematical models based on Petri nets to optimize the performance of synchronous [4, 5, 14] and asynchronous [15, 16] dataflow circuits. The main idea is to strategically place and size buffers to achieve the desired area/performance target. Buffers can be placed on any dataflow *channel* between the predefined dataflow units without compromising correctness [4]; internally, they must remain intact to honor the latency-insensitive handshake protocol. The most recent of these works [5] represents the dataflow circuit with a fine-grain timing model that captures the combinational delays on different *timing domains* (i.e., the datapath, the bidirectional handshake signal delays, and their occasional interactions). The idea is to employ mixed-integer linear programming to maximize the throughput of the system, ϕ, while minimizing the total number of inserted buffers, $\sum_i^N R_i$:

$$max\ (\alpha \cdot \phi - \beta \cdot \sum_i^N R_i), \tag{1}$$

under a specified clock period (CP) constraint and tuning constants α and β. However, none of these strategies accounts for synthesis information when regulating the circuit's critical path; thus, the resulting buffering is often over-conservative or imprecise. In this work, we include synthesis information into our timing optimization and iteratively refine the buffering to precisely regulate the circuit's frequency. To fairly compare our mapping-aware timing model with state-of-the-art work [5], our evaluation in Section VI employs the same MILP formulation as described above. Yet, our iterative refinement strategy is perfectly general—it could be employed to improve the accuracy of any dataflow-oriented buffer insertion strategy and adapted to any optimization objective.

IV. Mapping-Aware Algorithm

Section II illustrated the need to consider synthesis information when estimating the critical path of a dataflow circuit. To this end, we propose a mapping-aware algorithm that extracts this information from a standard technology mapper and devises an accurate timing model accordingly. The model represents each dataflow unit with delay nodes and explicitly specifies dataflow channels that connect them. In this way, it enables buffer placement between units such that the functionality of the communication protocol is always respected (see Section III).

Figure 2 illustrates the stages of this algorithm applied to a dataflow graph (DFG), composed of a fork (F), shifter (<<), adder (+) and branch (B). Although, for simplicity, we here consider an acyclic DFG, our methodology is general and supports cyclic graphs as well. We describe the main steps of our mapping-aware timing model creation in the remainder of this section.

A. LUT to DFG Mapping

Any technology mapper targetting FPGAs inputs a circuit (in our case, in the form of a DFG) and generates the corresponding graph of LUTs. Our timing model needs to understand how this mapping occurs and identify which physical constructs are used to implement each DFG path—this information can then be used to accurately model the delay of each path and to appropriately break it with buffers. The intermediate representation of the mapper labels each LUT output with the information on which operation of the DFG it originates from (in case of logic restructuring, it labels it with the operation that contributes most to computing the LUT output value); we can therefore trivially map each LUT to the corresponding DFG operator, as shown in Figure 2. However, it is not always straightforward to determine which DFG path their connection belongs to, as we will see next.

In the rest of this section, we refer to a *LUT edge* as an edge connecting two LUTs, $\texttt{L}_{\texttt{src}}$ and $\texttt{L}_{\texttt{dst}}$, in the LUT graph. We refer to dataflow operators as DFG *units*, connected by DFG *channels*; a sequence of units and channels forms a DFG *path*. Our task is to identify a unique DFG path for each LUT edge of the LUT graph, so that we can appropriately include it into our timing model, which we will discuss in Section IV-B.

One LUT edge to one DFG path. When $\texttt{L}_{\texttt{src}}$ and $\texttt{L}_{\texttt{dst}}$ of a LUT edge belong to DFG nodes connected through a single DFG path, the LUT edge trivially maps to this path. This is the case for the LUT edge connecting $\texttt{L}_6$ and $\texttt{L}_7$: $\texttt{L}_6$ belongs to the adder, $\texttt{L}_7$ to the branch, and they are connected through a single path (i.e., path adder-branch)—thus, LUT edge $\texttt{L}_6$-$\texttt{L}_7$ maps to this path. The same holds for LUT edge $\texttt{L}_2$-$\texttt{L}_6$, which maps to path fork-shift-add (i.e., red and blue edges of the DFG in Figure 2). Similarly, $\texttt{L}_1$-$\texttt{L}_3$ maps to path fork-fork, thus implying that this LUT edge is internal to the fork node. The fact that there is no LUT computing the shifter output implies that the shifter logic is included in the logic of another LUT on the path—in this case, $\texttt{L}_6$ shares the logic to compute the shift and the addition. We will include this phenomenon into our timing model in Section IV-B.

One LUT edge to many DFG paths. In this case, a single LUT edge could correspond to multiple candidate paths. For instance, in Figure 2, $\texttt{L}_5$ computes the output of the fork and connects to $\texttt{L}_7$ that calculates the branch output—however, there are two DFG paths connecting the fork and the branch (i.e., the left path through shift

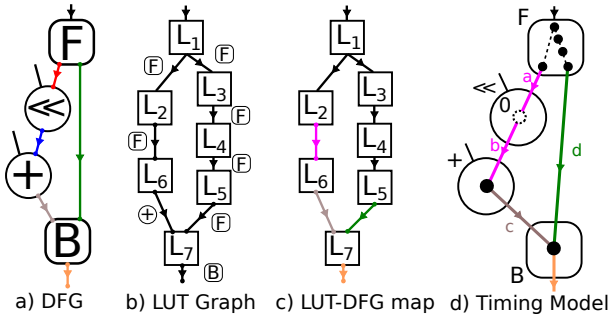

Figure 2: Our main steps to create a mapping-aware timing model from a DFG of a dataflow circuit. The resulting timing model accounts for the LUT mapping of DFG constructs and it is compatible with any performance optimization strategy for dataflow circuits.

and add, and the right path that connects them directly). Since the technology mapper does not provide us with a unique mapping of this edge, our algorithm simply chooses the DFG path with fewer dataflow units (in the example of Figure 2.a, this corresponds to the right path, with only two units). Naturally, there is no guarantee that this choice is correct—as we will show in Section V, our strategy can correct an optimistically chosen mapping by iteratively refining the synthesis information and improving the mapping accordingly.

One LUT edge to no DFG path. Due to logic synthesis restructuring, it might occur that a LUT edge cannot be associated with any DFG path (i.e., $\mathtt{L_{src}}$ and $\mathtt{L_{dst}}$ belong to two DFG nodes that are not at all connected in the DFG). In this case, we create an artificial edge that connects the corresponding DFG nodes in our timing model and introduce a combinational delay between them. Although this edge cannot be physically broken with a buffer, it contributes to the delay calculation and enables accurate frequency regulation (e.g., if the artificial edge is on the critical path, the path will be broken with a buffer either before or after it).

To create a complete LUT to DFG mapping, we iterate through all LUT edges and assign them to DFG paths according to the rules above. A presence of one or multiple LUT edges on a DFG path implies the presence of one or multiple combinational paths (with, possibly, different delays) along it; if there is no LUT edge on the DFG path, this implies that it has been optimized out by logic synthesis and can thus be ignored in the timing model.

B. Timing Model Generation

The result of the previous step is the list of all edges of the LUT graph mapped to DFG paths. In this step, we employ this mapping to generate the timing model of the circuit.

For each LUT in the LUT graph, we place a *delay node* in the dataflow unit that the LUT maps to; this node will be characterized with a predefined LUT delay. For each LUT edge in the LUT graph, we traverse the corresponding DFG path (as determined in the step above); in every dataflow node on the path from $\mathtt{L_{src}}$ to $\mathtt{L_{dst}}$, we place a *"fake"* delay node with combinational delay of 0.

An example of a complete timing model is depicted in Figure 2.d. The full black circles represent the combinational delay nodes, obtained from the LUTs in the left figure. Each node has a unitary delay equivalent to 1 LUT. For example, the node inside the adder corresponds to $\mathtt{L_6}$ in the LUT graph. The fork has multiple delay nodes, corresponding to LUTs $\mathtt{L_1}$ to $\mathtt{L_5}$. The empty (dashed) node in the shifter corresponds to a *fake* delay node of delay zero, inserted into this unit as it has no internal LUT.

The resulting timing graph of Figure 2.d is compatible with buffer placement algorithms for dataflow circuits: edges connecting delay nodes among different units strictly follow the dataflow edges and can thus be broken with a buffer, whereas the edges internal to the units remain intact [5]. We can therefore provide this timing model to any dataflow performance optimizer. The key difference from prior solutions is that the delays of our timing representation accurately reflect the circuit's post-synthesis LUT implementation; thus, the buffering can be performed with greater accuracy, as we will demonstrate in Section VI.

C. Penalty Computation

The timing model of the previous section can be directly plugged into any existing dataflow circuit pipelining strategy [4, 5] to place buffers on channels between dataflow units and control the critical path. However, placing buffers between two units might prevent logic optimizations between them and increase area (e.g., the shifter and adder of Figure 2 can share a LUT only if there is no buffer between them). We thus aim to avoid the placement of buffers between units that share a significant portion of their logic.

To this end, we introduce the notion of *penalty*, which serves as an indicator of the amount of logic shared among neighboring units:

$$\mathit{Penalty}(c_i) = \frac{|X_{fake}(c_i)|}{|X(c_i)|}, \tag{2}$$

where c_i is the channel for which we compute the penalty, $X(c_i)$ represents the sets of all delay nodes in the source unit of channel c_i, and $X_{fake}(c_i)$ is the set of *"fake"* delay nodes of the same unit connected to channel c_i. The penalty represents the fraction of nodes of the source unit that are shared with its successor over its total number of LUTs; its highest value is 1, indicating that the unit shares all of its logic with its neighboring unit.

In our performance optimizer, we associate each channel with a penalty computed as above and attribute it as a weight to any buffer placed on this channel. Our objective function aims to minimize the penalty-weighted sum of inserted buffers and will therefore break channels with lower penalty whenever possible:

$$\mathit{max}\left(\alpha \cdot \phi - \beta \cdot \sum_{i}^{N} R_i \cdot \Big(1 + \mathit{Penalty}(c_i)\Big)\right). \tag{3}$$

The only difference with respect to Equation 1 is the term $\mathit{Penalty}(c_i)$, representing the penalty of placing a buffer R_i on channel i.

Consider the example in Figure 2.d. If the path fork-shift-add-branch must be broken to honor a CP constraint, there would be three candidate channels, *a*, *b*, and *c*, for placing a buffer. For the computation of the penalty of *a*, the source node is the fork ($|X| = 5$ and $|X_{fake}(a)| = 0$). The penalty of this channel is 0. This means that no logic of the fork is shared with the shifter. For channel *b*, $|X| = 1$ and $|X_{fake}(b)| = 1$, the penalty is 1, which means that all the logic of the shifter is shared with the adder. For channel *c*, $|X| = 1$ and $|X_{fake}(c)| = 0$, the penalty is 0, which means that there is no logic shared is between adder and branch. Hence, channels a and c are preferred over channel b when placing a buffer.

D. Timing Domains

State-of-the-art performance optimization strategies for dataflow circuits diversify dataflow circuit signals into *timing domains* [5] which model delays of the datapath, handshake signals, and their interactions explicitly, as mentioned in Section III. Our mapping-aware timing modeling so far considered only a single domain—we here generalize our approach.

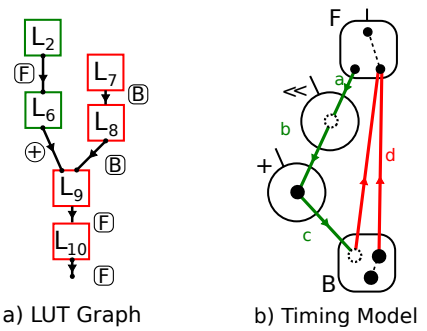

Figure 3: The mapping algorithm applied to a LUT edge connecting two different timing domains (shown in red and green). We rely on an existing dataflow modeling strategy [5] to identify places where timing domains meet and construct our timing model accordingly.

For each individual domain, we can apply the strategy of this section in isolation: the data, valid, and ready domain signals strictly follow the directed paths of the DFG; as they are immediately visible in the DFG structure, we can easily match them with LUT edges, as described in Section IV-A. However, this is not the case when domains interact, as the interacting paths might not correspond to directed DFG paths and, thus, might not be visible in the DFG. In other words, the DFG does not contain the information on domain interaction and the DFG path corresponding to a LUT edge capturing this interaction might not be immediately determinable. To this end, we rely on the model of Rizzi et al. [5] to obtain a list of all DFG nodes where domains interact—we can then reconstruct all paths between LUTs of different domains and map the LUT edges to these paths following the same strategy as in Section IV-A.

Consider the example in Figure 3. The LUTs highlighted in green and red correspond to two distinct timing domains; they interact in L_9. The timing model of each domain can be constructed as discussed above; however, the LUT edge from L_6 to L_9 does not correspond to any DFG path (i.e., there is no path from the adder to the fork). The modeling strategy of Rizzi et al. [5] identifies the branch as the connecting point of the green and the red domain; this allows us to identify the path add-branch-fork, with the delay of L_6 in the adder, L_9 in the fork, and an artificial node that connects them in the branch. This delay path can now be treated as any other timing model path.

V. An Iterative Approach

The approach from the previous section provides us with an accurate post-synthesis circuit description; we can use the timing model and penalties to place buffers following any dataflow pipelining approach. Despite the delay accuracy of this methodology, placing a buffer into the circuit impacts its synthesis (e.g., logic optimizations) and changes its LUT mapping; the more buffers we place, the more distorted the circuit mapping becomes from what we originally assume. To maintain high accuracy of our delay estimation and, thus, frequency control, it is imperative that we account for the synthesis and mapping changes due to buffering.

To this end, we develop an iterative pipelining approach: the idea is to gradually insert buffers into the circuit, while keeping track of the mapping changes that occur in the process. We can then account for these changes in the following iterations, while adding more buffers in a step-wise manner. In addition, the assumptions made during the mapping stage in Section IV-A (due to the ambiguity in mapping LUT edges to DFG paths) can be corrected in subsequent iterations. For instance, in the example of Figure 2, if the output edge of $\texttt{LUT}_5$ has been mapped to the wrong path in the DFG and a buffer incorrectly placed, we will identify it in the following iteration and remap the edge to the correct path.

Our iterative approach is illustrated in Figure 4. Initially, we provide the synthesizer with the dataflow graph of the circuit with buffers placed on the loop back edges of all combinational cycles. The output of the synthesizer is analysed by our LUT-to-DFG mapper, as explained in the previous section. We use its outputs to generate the timing model of our dataflow circuit and employ it to compute the penalties of each channel. We integrate the penalties into the optimization function of our timing model and provide it to a solver, which generates the optimal buffer placement for the given optimization function, thus completing an iteration.

We then provide the optimized circuit as input to the logic synthesizer and check if the desired number of logic levels (i.e., the CP target) has been achieved. If this is not the case (e.g., the buffer placement has negatively impacted the logic optimizations), we perform another buffering iteration. We select a subset of the previously found buffers, such that they are evenly distributed across the circuit's basic blocks (i.e., they are sparsely distributed and thus affect independent logic portions) and their penalties are minimal (i.e., they disrupt the minimal amount of logic optimizations in the circuit); we remove all other buffers. We provide this partially buffered circuit as input to the next iteration, which repeats the procedure above; the predefined buffers are fixed, but new buffers can be freely added by the solver based on the updated circuit mapping. This procedure repeats until all paths meet the desired number of logic levels.

Although there is no theoretical guarantee that the iterative approach converges (i.e., the continuous circuit updates may repeatedly cause significant synthesis disruptions and prevent the circuit from meeting the target), in practice, the post-synthesis circuit implementation quickly converges to the desired point in which all path lengths are accurately regulated by the buffers. We will confirm this in our evaluation in Section VI-B.

VI. Evaluation

In this section, we evaluate the effectiveness of our mapping-aware timing regulation approach.

A. Methodology and Benchmarks

We implement our strategy in Dynamatic [17], an open-source HLS compiler that produces synchronous dataflow circuits from C code. Our code is open-source as an add-on to Dynamatic [18].

Dynamatic contains a backend-agnostic timing optimizer [5]. Its timing model is obtained directly from the RTL descriptions of the dataflow units, whose delay values are determined by characterizing each unit in isolation. The optimizer uses a mixed-integer linear programming (MILP) model to place buffers; the objective is to maximize circuit throughput while honoring a particular clock period (CP) constraint. We reproduce this flow for our evaluation baseline: we measure the number of logic levels individually in each synthesized unit and calculate its combinational delay accordingly, by assuming the same predefined logic level delay value of 0.7 ns as we assume in our circuits.

We easily incorporate our strategy into the existing Dynamatic flow: we replace its circuit model, as described above, with the mapping-aware model, obtained from the circuit synthesized with ABC (using command '*if -K 6*") and as described in Section IV; we extend the MILP formulation to account for mapping penalties, as described in the same section. All other optimization aspects are

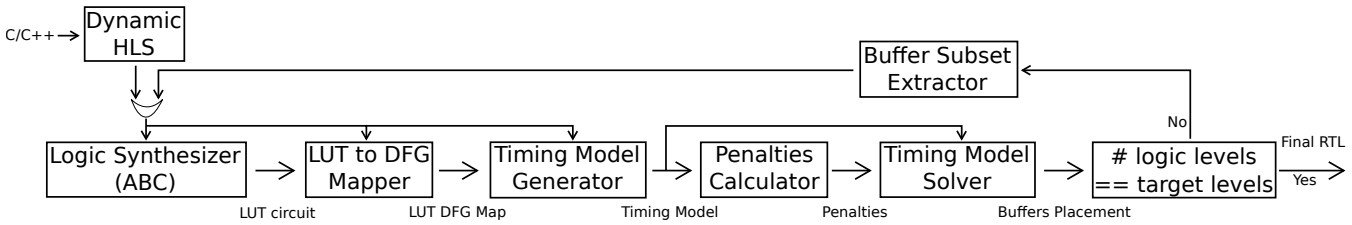

Figure 4: Starting from the C/C++ code, *Dynamic HLS* generates the DFG description of the circuit. The *Logic Synthesizer* (in our case, ABC) creates the LUT graph from the DFG. The *LUT-to-DFG Mapper* maps the LUT graph to the DFG. The *Timing Model Generator* receives the map and DFG as inputs to produce the timing model which is used by the *Penalty Calculator* to retrieve the penalties for each dataflow channel. The *Timing Model Solver* determines the optimal buffer placement for a pre-defined objective function. If the circuit with the computed buffers positions does not respect the target logic level count, the procedure is repeated with a fixed subset of buffers from the previous iteration. Once the target number of logic levels is achieved, the algorithm terminates.

identical as in the original model and we can, thus, fairly compare the two strategies. Note that our approach is in no way limited to this particular optimizer; our model and preferential strategy could be used with any buffering approach for dataflow circuits.

We aim to constrain all combinational delays to at most 6 logic levels of an estimated delay of 0.7 ns; we thus expect our CP to be around 4.2 ns. Dynamatic uses one MILP run to meet this target, whereas we iteratively run our algorithm. We use ODIN-II 8.1.0 [19] with Yosis to extract a blif description of the circuit and provide it as input to ABC 1.01 [20]. We use ModelSim 2021.2 [21] to verify our circuits and obtain the clock cycle count; we calculate the total execution time as a product of the CP and clock cycle count. We present all area and timing results after placing and routing the circuits with VPR 8.1.0 targetting a Stratix-IV device (using its modified architecture from the VTR benchmarks [22]). We use the Gurobi solver [23] to retrieve the MILP solutions.

We evaluate a collection of recent HLS kernels that were published as part of a recent work on dataflow circuits [5] and part of standard HLS benchmarks suites [24, 25]. In all our benchmarks, the MILP solver finds the optimal solution in under 3 minutes and our iterative method finds a solution in less than 3 iterations. In the rest of this section, we are interested in determining whether, for a given target CP, our solutions achieve superior area and performance with respect to the state-of-the-art strategy of Dynamatic [5].

B. Results

Table I summarizes our results; we also visualize them in Figure 5. Our circuits are superior to previous solutions in all performance and area aspects, as discussed in the remainder of this section.

Throughput and latency improvements. Prior work makes conservative delay assumptions and places redundant buffers on throughput- and latency-critical paths. Our strategy places buffers only when actually needed, thus systematically achieving higher throughputs and lower latencies, as reflected in the total number of clock cycles (i.e., *Clock Cycles* column in the table) and overall execution time.

LUT and FF savings. The redundant buffers of prior work typically also cause an unnecessary area overhead; as discussed above, our work omits these buffers and achieves simpler and cheaper circuits (see *LUTs* and *FFs* columns in the table).

Reliable critical path regulation. In all benchmarks, our approach achieves the targeted number of logic levels (see *Logic Levels* column), which indicates that our mapping algorithm successfully reasons about the circuit's mapping to FPGA logic, identifies actual post-synthesis critical paths, and accurately breaks them with buffers. This is not the case in prior work, which sometimes accidentally and nondeterministically achieves a good CP (e.g., *mvt* in the table

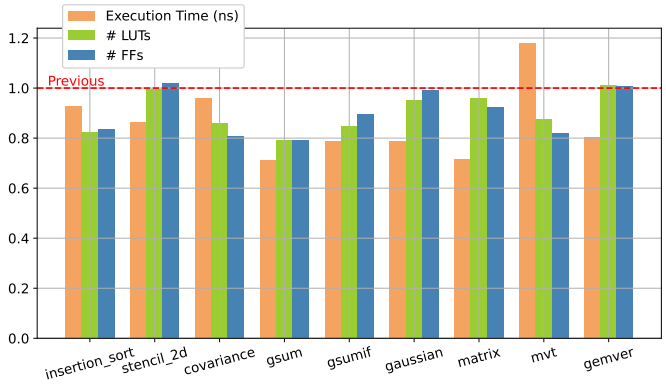

Figure 5: Execution time (i.e., product of CP and clock cycles) and resources (i.e., LUT and FF count) of the dataflow circuits optimized with our iterative, mapping-aware strategy, normalized to the corresponding circuits optimized with a state-of-the-art, mapping-agnostic approach [5]. Our designs are depicted as columns and the baseline values are all in dashed line equal to 1. Most of our circuits Pareto-dominate prior work by simultaneously reducing both execution time and resource requirements.

meets the CP target; although not reported in the table, we note that the achieved CP unpredictably diverges for slight target changes). The minor discrepancies from the target of 4.2 ns are due to the fact that we currently do not account for routing delays, which can significantly contribute to the critical path, as others have noted before us [12, 13]. It is important to note that accounting for this effect could only further improve our results and increase the benefits of our approach over the mapping-agnostic baseline that cannot account for neither mapping nor routing.

In summary, the fact that our approach always achieves Pareto-optimal solutions (in most cases, our circuits are Pareto-dominant as well, as they improve both area and performance) points to the effectiveness of our approach and demonstrates the need to include mapping awareness in dataflow circuit performance optimization.

VII. Conclusions

Dataflow circuits have recently gained notable interest in the context of HLS. However, current techniques to generate and optimize such circuits from high-level code are agnostic of the underlying hardware architecture, which makes their frequency difficult to estimate and regulate. In this work, we propose a timing optimization approach that iteratively refines the circuit's buffering while accounting for its LUT mapping on an FPGA. We demonstrate that our approach can

Benchmark	CP (ns)		Clock Cycles		Execution Time (ns)		ET Ratio	# LUTs		LUT Ratio	# FFs		FF Ratio	Logic Levels	
	Prev.	Iter.	Prev.	Iter.	Prev.	Iter.		Prev.	Iter.		Prev.	Iter.		Prev.	Iter.
insertion_sort	5.11	5.02	232	219	1186	1100	-8%	3528	2903	-18%	2230	1867	-17%	6	6
stencil_2d	4.97	4.65	30674	28300	152450	131511	-14%	2396	2386	-1%	1567	1599	+3%	5	5
covariance	4.68	4.49	179494	179465	840032	805798	-5%	2653	2285	-14%	1616	1306	-20%	5	4
gsum	4.69	4.02	5368	4450	25176	17889	-29%	1084	858	-21%	649	514	-21%	6	4
gsumif	5.02	4.79	5271	4342	26461	20799	-22%	1513	1284	-16%	917	820	-11%	5	5
gaussian	5.35	4.75	5050	4481	27018	21285	-22%	1302	1241	-5%	808	801	-1%	4	5
matrix	4.64	4.77	101515	70828	471030	337850	-29%	1455	1397	-4%	966	891	-8%	5	6
mvt	3.83	4.53	20115	20072	77041	90927	+19%	2420	2117	-13%	1607	1317	-19%	4	5
gemver	5.92	5.31	9622	8632	56963	45836	-20%	7207	7281	+2%	5129	5177	+1%	5	4

TABLE I: Comparison of our iterative method (*Iter.*) with a state-of-the-art pipelining approach (*Prev.*) [5]. The solutions obtained by our iterative approach always meet the target of 6 logic levels, typically achieve a better clock period (CP), and consistently improve throughput (i.e., lower clock cycle count) over prior solutions, thus causing a persistent improvement in total execution time. In most cases, the resources (i.e., LUTs and FFs) improve as well, as our approach places fewer buffers at strategic positions in the circuit.

precisely control the number of logic levels (and, consequently, the combinational delay) of each dataflow pipeline stage. Our accurate buffering strategy reduces the execution time by up to 29% and the resource requirements by up to 21% in LUTs and FFs compared to prior dataflow circuit buffering strategies. Our work is the first step in understanding of how HLS-produced dataflow circuits map to FPGA constructs and the foundation to make them amenable to a variety of FPGA architectures.

REFERENCES

[1] M. Tan, S. Dai, U. Gupta, and Z. Zhang, "Mapping-aware constrained scheduling for LUT-based FPGAs," in *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 2015, pp. 190–9.

[2] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 2018, pp. 127–36.

[3] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, TX, Mar. 2005, pp. 177–86.

[4] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, CA, Feb. 2020, pp. 186–96.

[5] C. Rizzi, A. Guerrieri, P. Ienne, and L. Josipović, "A comprehensive timing model for accurate frequency tuning in dataflow circuits," in *32nd International Conference on Field-Programmable Logic and Applications*, 2022, pp. 375–83.

[6] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 1–42, Aug. 2022.

[7] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *Proceedings of the 32nd International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2013, pp. 211–18.

[8] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the 43rd Design Automation Conference*, San Francisco, CA, Jul. 2006, pp. 433–38.

[9] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[10] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for FPGA HLS using graph neural networks," in *Proceedings of the 39th International Conference on Computer-Aided Design*, Virtual, Nov. 2020, pp. 1–9.

[11] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen, "Fast and effective placement and routing directed high-level synthesis for FPGAs," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, Feb. 2014, p. 1–10.

[12] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs," in *Proceedings of the 29th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Virtual, Feb. 2021, pp. 81–92.

[13] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in FPGA high-level synthesis," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Mar. 2019, pp. 1130–5.

[14] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar, "A general model for performance optimization of sequential systems," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2007, pp. 362–69.

[15] M. Najibi and P. A. Beerel, "Slack matching mode-based asynchronous circuits for average-case performance," in *Proceedings of the 32nd International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2013, pp. 219–25.

[16] G. Venkataramani and S. C. Goldstein, "Leveraging protocol knowledge in slack matching," in *Proceedings of the 25th International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2006, pp. 724–29.

[17] L. Josipović, A. Guerrieri, and P. Ienne, "Dynamatic: From C/C++ to dynamically scheduled circuits," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, CA, Feb. 2020, pp. 1–10.

[18] "Mapping-aware buffer placement plug-in," Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7821272

[19] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II–an open-source Verilog HDL synthesis tool for CAD research," in *Proceedings of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines*, Charlotte, NC, May 2010, pp. 149–56.

[20] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[21] Intel, "ModelSim," 2021. [Online]. Available: https://www.intel.com/

[22] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. ElDafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "VTR 8: High performance CAD and customizable FPGA architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 2, pp. 1–55, Jun. 2020.

[23] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[24] L.-N. Pouchet, *Polybench: The polyhedral benchmark suite*, 2012. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench

[25] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, NC, October 2014.