A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits

Carmine Rizzi*, Andrea Guerrieri†, Paolo Ienne†, 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

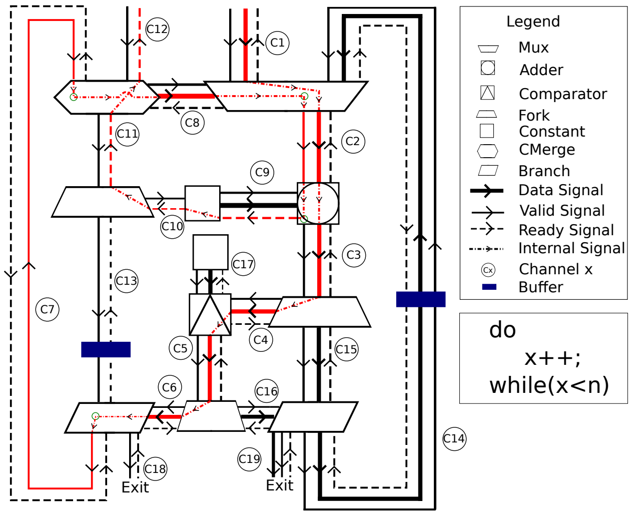
***Abstract*—The ability of dataflow circuits to implement dynamic scheduling promises to overcome the conservatism of static scheduling techniques that high-level synthesis tools typically rely on. Yet, the same distributed control mechanism that allows dataflow circuits to achieve high-throughput pipelines when static scheduling cannot also causes long critical paths and frequency degradation. This effect reduces the overall performance benefits of dataflow circuits and makes them an undesirable solution in broad classes of applications. In this work, we provide an in-depth study of the timing of dataflow circuits. We develop a mathematical model that accurately captures combinational delays among different dataflow constructs and appropriately places buffers to control the critical path. On a set of benchmarks obtained from C code, we show that the circuits optimized by our technique accurately meet the clock period target and result in a critical path reduction of up to 38% compared to prior solutions.**

I. Introduction

Dataflow circuits [1, 2] have recently been explored in the context of *high-level synthesis* (HLS), as they implement dynamically scheduled pipelines that achieve high throughput in irregular and control-dominated programs [3, 4]. Although their pipelining capabilities are often superior to static solutions, dataflow circuits typically suffer from long combinational paths that limit the achievable operating frequency; this effect largely nullifies the benefits of dynamic scheduling and prevents the widespread usage of dataflow circuits in HLS.

Dataflow circuits employ *buffers* to break combinational paths and ensure that all combinational delays meet the desired clock period target. However, in a distributed dataflow network, these combinational paths are difficult to identify: apart from the datapath operators (with easily determinable connections and combinational delays), dataflow circuits contain a two-way distributed control mechanism that the dataflow units use to exchange data with their predecessors and successors. These control signals travel in opposite directions, combine with the datapath in particular places, and interact with each other; the resulting combinational paths are not self-evident and their impact on the circuit delay is challenging to capture.

In this work, we present a comprehensive timing model for dataflow circuits that enables accurate frequency estimates and precise critical path regulation. Apart from the timing of the datapath, our model includes timing information about the dataflow control network and accounts for all interactions between the datapath and the control signals. We incorporate our timing model into a buffer placement strategy that maximizes circuit throughput under a clock period constraint.



```
do
    x++;
while(x<n)
```

Figure 1: Considering only the datapath of a dataflow circuit (thick lines) is not sufficient for accurate critical path regulation, as the critical path often spans through both the data and the bidirectional control network. One such path is highlighted in red: the data propagates through channels C1 to C6, combines with the forward-going control signal (thin line, channels C7, C8 and C2), and then interacts with the backward-going control signal (dashed line, channels C9 to C12), thus causing a long combinational delay. The purpose of this work is to systematically identify such long combinational paths and appropriately break them with buffers to honor a clock period target.

On a set of benchmarks obtained from C code, we show a frequency improvement of up to 38% and a total execution time reduction of up to 34% over prior solutions. Our circuits meet the same clock period target as their static counterparts, which increases their performance benefits in cases where dynamic scheduling is useful and makes them competitive with static circuits otherwise, thus extending the usefulness and applicability of dynamically scheduled HLS.

II. Where is the Critical Path in a Dataflow Circuit?

The dataflow circuit in Figure 1 implements the functionality of the code on the right and is constructed using a standard C-to-dataflow HLS strategy [3]. It is built out of dataflow *units*, connected by *channels*, that exchange data (thick lines) and communicate with a set of handshake control signals: a *valid* signal (thin lines) travels in the same direction as the data and indicates to each successor that a unit holds a *token*,

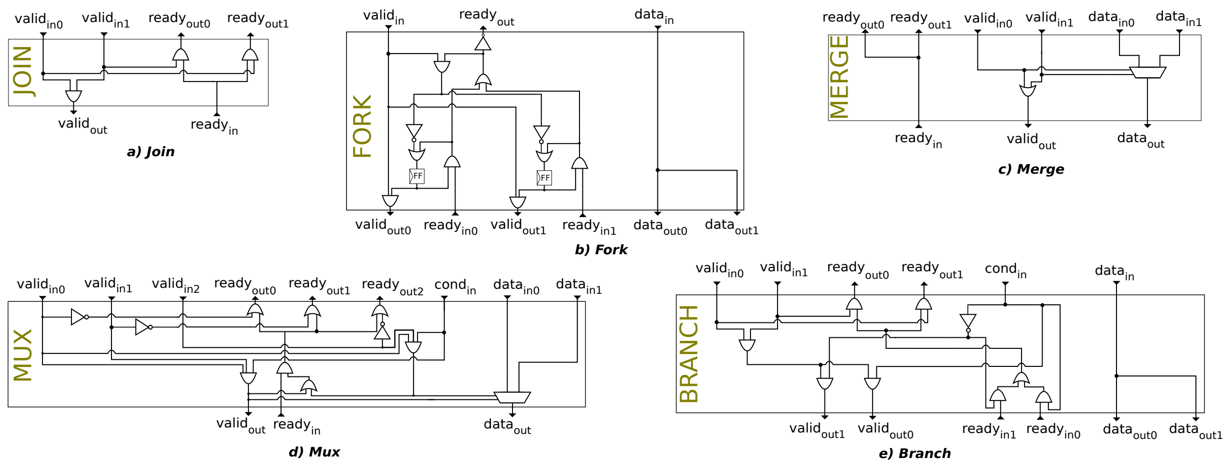

JOIN
valid_in0 valid_in1 ready_out0 ready_out1
valid_out ready_in
a) Join
FORK
valid_in ready_out data_in
FF
valid_out0 ready_in0 valid_out1 ready_in1 data_out0 data_out1
b) Fork
MERGE
ready_out0 ready_out1 valid_in0 valid_in1 data_in0 data_in1
ready_in valid_out data_out
c) Merge
MUX
valid_in0 valid_in1 valid_in2 ready_out0 ready_out1 ready_out2 cond_in data_in0 data_in1
valid_out ready_in data_out
d) Mux
BRANCH
valid_in0 valid_in1 ready_out0 ready_out1 cond_in data_in
valid_out1 valid_out0 ready_in1 ready_in0 data_out0 data_out1
e) Branch

Figure 2: Gate-level descriptions of dataflow units [1, 5]. The goal of this work is to develop a mathematical description of the timing characteristics of dataflow units—we use this information to optimize the critical path of dataflow circuits obtained from high-level code.

whereas a *ready* signal (dashed lines) propagates backward and indicates to each predecessor the readiness of a unit to accept a new token. The initial value of x enters the mux through channel C1. The token circulates through the loop (and x is incremented by 1) until the exit condition is met (as determined by the branch); then, the execution terminates and the final value of x exits the loop. The left loop is dataless and used to regulate the behavior of the datapath [6]: an initialization token enters from the starting point (C12) and circulates through the loop (dictated by the same exit condition as the token x); on each passage, it indicates to the mux on the right which of the two operands to select and triggers the constant for incrementing x.

The shown circuit is perfectly functional, as the buffers in channels C13 and C14 ensure the absence of combinational cycles. However, the critical path of the circuit is not regulated in any way—the circuit requires additional buffers that ensure that all combinational delays are smaller than the desired clock period target. An intuitive way to reason about this placement is to consider the combinational delays of the operators on the datapath (e.g., mux, adder, comparator): whenever the sum of the operator delays is larger than the target, the path must be broken with a buffer. This simplistic strategy would result in additional buffers in the channels carrying the data (e.g., a buffer in channel C3 would break the combinational path between the adder and the comparator), but it does not consider that the bidirectional control signals interact with each other and with the datapath, which may create long and complex combinational paths. An example of one such path (out of many) is highlighted in red in the figure. The datapath through channels C1 to C5 computes the loop exit condition which is forked to the branches. In the left branch, the condition determines the validity of the appropriate branch output; thus, the data of channel C6 combinationally produces the valid signal in channel C7. This signal propagates through the cmerge and mux, ultimately reaching the adder through channel C2. In the adder, the validity of one input is used to indicate the readiness of the adder to accept data from the other input—thus, the valid signal from channel C2 combinationally produces the ready signal in channel C9, which continues propagating backward, up to the input C12. The existence of such long combinational paths is not immediately clear, yet they may significantly impact the critical path of the circuit.

In the rest of this paper, we provide a systematic methodology to identify and regulate the critical path of complex dataflow networks. In Section III, we describe dataflow circuits and prior performance optimization approaches. Section IV summarizes the main idea of this paper. In Section V, we devise a timing model for dataflow units that captures various interactions and combinational delays between their data and control signals. In Section VI, we incorporate this model into a complete performance optimization strategy that exploits our timing information to appropriately place buffers into the dataflow circuit. In Section VII, we demonstrate that our approach accurately identifies and regulates the critical path, thus resulting in significant speedups compared to prior performance optimization strategies for dataflow circuits.

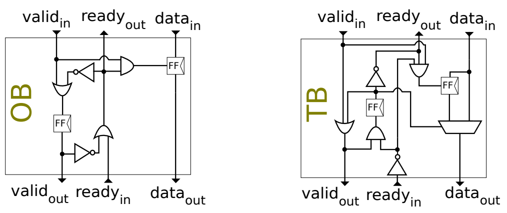
III. Background and Related Work

In this section, we describe the dataflow units and buffers that we use. We then outline what others have done before us to optimize dataflow circuit performance.

A. Dataflow Units

In this work, we consider synchronous dataflow units which have been extensively described in literature [1, 5] and used to construct dataflow circuits out of software programs [3, 7]. We provide the gate-level descriptions of the basic dataflow units in Figure 2; all other units we employ are a composition or trivial extension of those in the figure.

(1) A *join* (Figure 2a) synchronizes incoming tokens and is used inside operators that require multiple operands (e.g., the adder and comparator in Figure 1). Whenever both predecessors provide valid data (i.e., both $valid_{in}$ signals are set to 1), the join becomes valid. The join indicates its



valid$_{in}$ ready$_{out}$ data$_{in}$
OB
FF
FF
valid$_{out}$ ready$_{in}$ data$_{out}$
valid$_{in}$ ready$_{out}$ data$_{in}$
TB
FF
FF
valid$_{out}$ ready$_{in}$ data$_{out}$

Figure 3: Opaque (OB) and transparent (TB) buffer [1]. These buffers can be inserted on any dataflow channel to control the critical path and regulate throughput. Using buffers to optimize throughput has been extensively explored, yet their role in controlling the critical path has not been fully investigated; we tackle this aspect in this work.

readiness ($ready_{out}$) to the predecessors when their data are both valid and the successor is ready to accept a token. As the $ready_{out}$ signal requires the information of the opposite operand validity, there is a combinational path from the $valid_{in}$ signal of the other operand (e.g., from $valid_{in_0}$ to $ready_{out_1}$).

(2) A *fork* replicates a single token at its input and sends it to its multiple outputs. Figure 2b shows a 2-output eager fork: it can dispatch tokens to the successors at different times and has internal registers to keep track of where the token was sent; the fork indicates readiness to accept new data only after the previous token has been issued to all successors.

(3) A *merge* (Figure 2c) is employed in places where program control flow meets (e.g., at an input of a compiler basic block [8] with multiple predecessors). The internal data multiplexer selects the input coming from the active predecessor, as determined based on the validity of the inputs.

(4) A *mux* is a deterministic version of the merge: in addition to the data, it receives a condition that determines which input to select. As indicated in Figure 2d, the condition $cond_{in}$ regulates the validity and readiness of the mux, as well as the select signal of the internal data multiplexer.

(5) A *branch* is used to implement control flow; it sends a token to one of its multiple successors based on the condition value. As shown in Figure 2e, the output validity is computed based on the input condition. The readiness of each input is determined by the validity of the other, as both the data and the condition must be available for the branch to issue a token.

As illustrated in this section, dataflow units have a variety of combinational paths that connect their datapath to the control network, as well as placed where the bidirectional control signals interact. These features may impact the circuit's critical path and must be accounted for when optimizing performance.

B. Buffers in Dataflow Circuits

Thanks to the latency-insensitivity of dataflow circuits, buffers can be placed on any dataflow channel without compromising the circuit's correctness [9]. Yet, they have a key impact on performance: they can increase the circuit's frequency by breaking combinational paths and regulate its throughput by storing incoming tokens to avoid backpressure [10].

We differentiate two types of buffers [1], whose gate-level representations are shown in Figure 3: (1) *Opaque buffer* (OB) has a register on the data and the valid path, thus delaying the incoming token for one clock cycle; on the other hand, its readiness is determined combinationally based on the readiness of the successor. (2) *Transparent buffer* (TB) contains a multiplexer that can propagate the input data token combinationally to the output in case the successor is available; otherwise, the token is stored in an internal register. In contrast to OB, the TB breaks the ready path with a register, whereas its valid signal propagates through the unit combinationally. Both buffers can be adjusted to contain multiple buffer slots by increasing the number of internal registers (thus, essentially, transforming the buffer into a FIFO that can store multiple tokens) [10].

As the figures suggest, the insertion of each of these buffers into a dataflow channel will have a dual effect on timing: some combinational paths will be broken, yet the delay of others will increase due to the buffer's combinational logic. This effect may also contribute to the circuit's critical path.

C. Performance Analysis of Dataflow Circuits

Standard HLS tools use modulo scheduling [11, 12] to perform pipelining and retiming; yet, this approach is not applicable in the dataflow context where a static schedule is absent. Instead, authors have explored mathematical models based on Petri net theory to analyze dataflow circuit performance [13]–[15]. Several techniques for asynchronous dataflow circuits optimize throughput via buffer sizing [16, 17]; those targeting synchronous dataflow circuits extend this problem to simultaneously optimize throughput and clock period [10, 18].

In this section, we focus on the work of Josipović at al. [10] that targets synchronous circuits obtained from C code and is thus the most relevant for our work. This strategy aims to optimize circuit timing as follows: (1) *Throughput* (ϕ). The throughput of each program loop is maximized by inserting TB buffers into dataflow channels and sizing them appropriately, i.e., by modifying the number of buffer slots, $\mathit{BuffSlots}_{C_i}$, in dataflow channel C_i. (2) *Clock period* (CP). The combinational delays of the datapath are accumulated across dataflow units and channels. Whenever the delay is larger than the target CP, an OB is placed in the channel to break the combinational path (i.e., at least one buffer slot is inserted into the channel, $\mathit{BuffSlots}_{C_i} \geq 1$). These two aspects are combined in a single *mixed-integer linear programming* (MILP) problem that maximizes circuit throughput under a clock period constraint, while minimizing the total number of inserted buffers: $\max(\phi - \lambda \sum \mathit{BuffSlots}_{C_i})$, where λ is a tuning constant.

Although this approach has been demonstrated successful in achieving high-throughput pipelines, it fails in capturing the intricate details of dataflow networks described in the previous sections, thus resulting in imprecise critical path analysis and significant deviations between the achieved and targeted clock period (the authors measured a CP that is up to 2.4× of their target). In our work, we also aim to optimize throughput and clock period simultaneously; we thus follow the same throughput optimization strategy outlined in point (1) and employ the same cost function as indicated above. Our contribution is a new methodology for critical path regulation

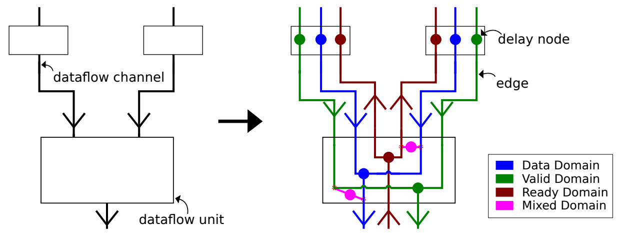


Figure 4: The main idea of our optimization strategy is to represent a network of dataflow units and channels as a fine-grain graph, where nodes and edges represent combinational delays and interconnects between the data and control signals. This approach allows us to optimize circuit timing with greater accuracy.

that overcomes the limitations of point (2) and ensures that dataflow circuits successfully meet the clock period target.

IV. Key Insight

It is evident from Figure 2 that, aside from the datapath, dataflow units contain (in some cases, significant) combinational logic that computes their control signals; additionally, there are many situations where the control signals interact with each other or combine with the datapath. What is more, although buffers from Figure 3 break some combinational paths (and thus help to regulate the critical path), they simultaneously introduce a delay increase on other paths. Clearly, a composition of such units into a complete dataflow system may cause complex signal interactions and, consequently, long combinational paths (such as the one illustrated in Figure 1). Thus, to control the critical path of the circuit, we need a systematic way to reason about these effects.

The main contribution of this work is a comprehensive mathematical model that accounts for all combinational paths and delays in a dataflow circuit and precisely regulates its critical path. The basic idea of our timing optimization approach is illustrated in Figure 4: instead of considering the dataflow network as an interconnect of dataflow units and channels (and suffering from the limitations discussed in Section II), we break down these constructs into nodes and edges representing each individual combinational path and delay constituent. Instead of a single dataflow channel, our model describes individual edges carrying the data, valid, and ready signals and explicitly indicates points where they interconnect; instead of complete dataflow units, the nodes of our representation indicate distinct combinational delays of every input-output combinational path through each unit. This fine-grain representation allows us to precisely compute all combinational delays in the dataflow circuit and to accurately place buffers to break long combinational paths. In the following sections, we formalize our timing representation and present our mathematical formulation for performance optimization.

V. Looking Inside the Dataflow Units

Figure 5 details the timing representations of the dataflow units illustrated in Figures 2 and 3. Every dataflow channel is broken into edges representing the data (if present), valid, and ready signal. Every combinational path through a unit

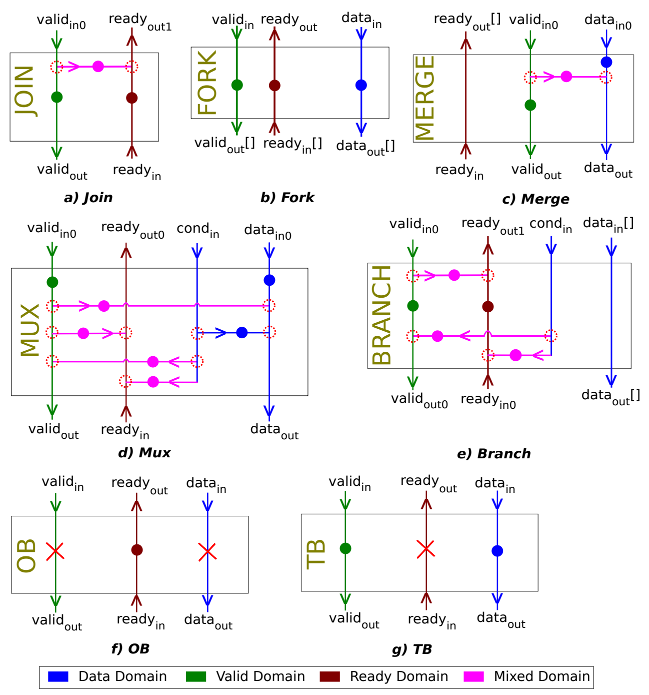


Figure 5: Timing representation of dataflow units from Figures 2 and 3. It captures all combinational delays and interconnects between the data, valid, and ready signals within each unit. The OB and TB have backward- and forward-going combinational delays, respectively, which our performance optimization strategy also considers.

is represented as an edge from input to output, where a node corresponds to a combinational delay (the empty circles indicate a delay equal to zero) and the red cross is a register that breaks the combinational path. The schematics show only a single distinct input-output pair of each dataflow unit—for simplicity, we omit the symmetrical input-output connections.

Based on their properties and the type of information that they carry, we classify the edges and nodes of our representations into four *timing domains*: (1) *Data domain*, $Dom(D)$, is a set of all nodes and edges that carry exclusively data information. The nodes of this domain are characterized with the datapath (i.e., computational unit) delays. (2) *Valid domain*, $Dom(V)$, is the set of edges and nodes that represent the valid control signal propagation. The nodes of this domain are characterized with the combinational delays between the input ($valid_{in}$) and output ($valid_{out}$) signals of each dataflow unit. (3) *Ready domain*, $Dom(R)$, is the set of edges and nodes that represent the ready control signal propagation. The nodes of this domain are characterized with the combinational delays between the input ($ready_{in}$) and output ($ready_{out}$) signals of each dataflow unit. (4) *Mixed domain*, $Dom(M)$, is a set of edges and nodes that represent the interconnects between the data, valid, and ready domains. The nodes of this domain are characterized with the combinational delays between the corresponding inputs and outputs of each dataflow unit.

The edges and nodes in the mixed domain enable us to reason about complex paths in which the domains interconnect,

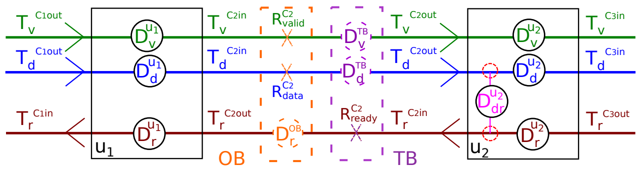


Figure 6: Mathematical model for timing optimization. The figure indicates the variables that we employ to regulate the critical paths of our circuits; essentially, they sum up combinational delays across particular nodes and channels and ensure that every delay larger than the target is broken with a buffer.

such as the one shown in Figure 1: the transition of the data into the valid network in the leftmost branch is caused by the mixed node between the branch condition and valid signal shown in Figure 5; similarly, our mux description captures the transition of the data (i.e., condition $cond_{in}$) network into the valid network, thus explaining the delay propagation from C8 to C2 in Figure 1. The direction change of the path in the adder (i.e., from C2 to C9) is exactly what our join representation shows as a mixed path between the valid and the ready signals. Thus, our unit timing representation is key to analyzing the critical path of the circuit; in the following section, we incorporate this notion into a complete mathematical formulation for timing optimization.

VI. A Comprehensive Mathematical Model for Dataflow Timing Optimization

As the performance optimization model outlined in Section III-C [10], we aim to maximize circuit throughput under a clock period constraint while minimizing the number of inserted buffers. Thus, we adopt the MILP-based cost function and throughput constraints directly from that work. In this section, we describe the novelty of our optimization approach: the MILP constraints that regulate the clock period, that we derive from the unit timing models presented in Section V. Our constraints provide the following optimization capabilities that prior approaches omit: (1) we optimize combinational delays of the datapath as well as of the control network, (2) we account for interactions between the datapath and the bidirectional control signals, and (3) we consider the combinational delays of the buffers in the data and control paths.

Variables and constants. Our constraints use the following variables and constants, illustrated in Figure 6:

- $R_{timeDom}^{C_i}$ is a binary variable indicating if a register is placed on an edge of channel C_i, in time domain $timeDom$ (e.g., $R_{valid}^{C_i} = 1$ indicates that the valid edge in dataflow channel C_i requires a register).
- $BuffSlots_{C_i}$ is an integer variable indicating the number of buffer slots in channel C_i. This value is unique for each channel and its minimum value is bound by the presence of registers on individual edges (e.g., $R_{valid}^{C_i} = 1$ implies that an OB is present in the entire channel, thus $BuffSlots_{C_i} \geq 1$).
- $T_{timeDom}^{C_i,Port}$ is a real variable describing the time (i.e., combinational delay) at the input/output port of an edge of channel C_i, in time domain $timeDom$.
- $D_{timeDom}^{u}$ is a real timing constant which represents a delay node in $timeDom$, within a unit u.
- CP is a constant representing the target clock period.
- CP_{max} is a constant and upper bound for any time value; its value must be larger than CP.

Data domain constraints. The datapath constraints ensure that all datapath delays meet the clock period target. We apply them to all edges and nodes in the data domain, $Dom(D)$.

Equation 1 computes the combinational delay propagation through each node in $Dom(D)$ and is formulated for each input-output pair of edges of the node. Equation 2 ensures that all combinational delays at the edge input and output must be smaller than the target clock period; it is formulated for each data domain edge of channel C_i. Equation 3 determines whether the data domain edge must be cut with an OB.

$$T_{data}^{C_i,in} \geq T_{data}^{C_j,out} + D_{data}^{u} \tag{1}$$

$$T_{data}^{C_i,in} \leq CP, \; T_{data}^{C_i,out} \leq CP \tag{2}$$

$$T_{data}^{C_i,out} \geq T_{data}^{C_i,in} - CP_{max} \cdot R_{data}^{C_i} + D_{data}^{TB} \cdot R_{ready}^{C_i} \tag{3}$$

If the delay at the edge output is smaller than the CP target, Equation 3 will set $R_{data}^{C_i}$ to 0; this will propagate the delay from the input to the output of the edge and increase it by the delay of the TB, in case it is present in the channel (i.e. if $R_{ready}^{C_i} = 1$). If the output delay is larger than the target, the edge is cut by an OB by setting $R_{data}^{C_i}$ to 1. As the OB might be followed by a TB on the same channel, the output time must be adjusted to account for its presence and corresponding delay:

$$T_{data}^{C_i,out} \geq D_{data}^{TB} \cdot R_{ready}^{C_i} \tag{4}$$

This equation assumes that, if a channel contains both an OB and a TB, the OB is placed before the TB, as depicted in Figure 6. The opposite order is possible as well; the same delay value would then be added to the channel input time. Without loss of generality, we follow the order described above and place buffers in our circuits accordingly.

Valid domain constraints. The constraints for the valid domain are analogous to those of the data domain; they apply to each edge of channel C_i and node that are in $Dom(V)$:

$$T_{valid}^{C_i,in} \geq T_{valid}^{C_j,out} + D_{valid}^{u} \tag{5}$$

$$T_{valid}^{C_i,in} \leq CP, \; T_{valid}^{C_i,out} \leq CP \tag{6}$$

$$T_{valid}^{C_i,out} \geq T_{valid}^{C_i,in} - CP_{max} \cdot R_{valid}^{C_i} + D_{valid}^{TB} \cdot R_{ready}^{C_i} \tag{7}$$

$$T_{valid}^{C_i,out} \geq D_{valid}^{TB} \cdot R_{ready}^{C_i} \tag{8}$$

Ready domain constraints. Consistently with Figure 5g, the combinational delay in the ready domain is cut by a TB (indicated by $R_{ready}^{C_i} = 1$), whereas the presence of the OB ($R_{data}^{C_i} = 1$) increases the combinational delay on the ready edge. For each edge of channel C_i and node that are in $Dom(R)$, the following constraints apply:

$$T_{ready}^{C_i,in} \geq T_{ready}^{C_j,out} + D_{ready}^{u} \tag{9}$$

$$T_{ready}^{C_i,in} \leq CP, \; T_{ready}^{C_i,out} \leq CP \tag{10}$$

$$T^{C_i,out}_{ready} \geq T^{C_i,in}_{ready} - CP_{max} \cdot R^{C_i}_{ready} + D^{OB}_{ready} \cdot R^{C_i}_{data} \quad (11)$$

$$T^{C_i,out}_{ready} \geq D^{OB}_{ready} \cdot R^{C_i}_{data} \quad (12)$$

Mixed domain constraints. The presence of delays in the mixed domain is dictated by the dataflow unit topology and differs for each unit (see Figure 5). The edges internal to the dataflow units cannot be broken with a register (as it would require changing the unit structure), so the paths of the mixed domain do not require buffer placement constraints (such as Equations 3, 7 and 11); they simply propagate a delay from one timing domain to another. Thus, for every input-output edge pair of each node in the domain $Dom(M)$:

$$T^{C_i,in}_{timeDom1} \geq T^{C_j,out}_{timeDom2} + D^{u}_{timeDom2\text{-}timeDom1} \quad (13)$$

We use this equation to describe all mixed domain connections from Figure 5 (shown in pink). For instance, the equation $T^{C_7,in}_{valid} \geq T^{C_6,out}_{data} + D^{branch}_{data\text{-}valid}$ expresses the connection between the condition entering the left branch of Figure 1 from the datapath (channel C6) and its output valid signal (C7).

Buffer consistency constraints. The decision to place an OB cuts simultaneously the data and the valid path (see Figure 5f); thus, the $R^{C_i}_{data}$ and $R^{C_i}_{valid}$ variables must always have the same value, as specified by Equation 14 (i.e., either both paths or neither are cut by an OB). According to Equation 15, the placement of OB and TB dictates the minimal number of buffer slots in the channel.

$$R^{C_i}_{valid} = R^{C_i}_{data} \quad (14)$$

$$BuffSlots_{C_i} \geq R^{C_i}_{data} + R^{C_i}_{ready} \quad (15)$$

Equation 15 connects the constraints presented in this section to the cost function and throughput constraints that we employ, as discussed in Section III-C; the number of slots bound by this equation may be further increased for throughput regulation. The output of the MILP (i.e., values of $R^{C_i}_{timeDom}$ and $BuffSlots_{C_i}$ for each channel C_i) indicate the types and sizes of buffers that should be placed in each dataflow channel to achieve maximal throughput under a given clock period constraint.

VII. Evaluation

In this section, we evaluate the effectiveness of our timing optimization strategy.

A. Methodology and Benchmarks

We incorporate our performance optimization strategy into Dynamatic [7], an open-source HLS compiler that synthesizes C code into synchronous dataflow circuits. Our optimizer is open-sourced as a plugin to Dynamatic at dynamatic.epfl.ch. Dynamatic contains the performance analysis described in Section III-C that considers exclusively the datapath delays (i.e., it employs a simpler version of our Equations 1 to 3 that excludes buffer delays; all other timing aspects discussed in Section VI are left out entirely); we use this approach as a baseline for our evaluation. Instead, we model the dataflow units as described in Section V and employ the MILP constraints from Section VI. Our other circuit generation aspects are identical to Dynamatic, so we can fairly compare the two performance optimization approaches; the only difference in the generated circuits will be the buffering and, consequently, the critical path, directly determining the achieved clock period. Although we demonstrate our approach on Dynamatic, our technique is in no way limited to this HLS strategy and can be applied to any dataflow-oriented HLS approach [3, 4, 20].

To obtain the delay values required by our timing model from Section V, we characterize all dataflow units post-place-and-route, targetting a Kintex-7 Xilinx FPGA. We measure delays between every unit input and output port and include them in an extensible unit timing library so that our approach could be easily adapted to other targets. We use the *Gurobi Optimizer v. 9.5.1* [21] to solve the mixed-integer linear programming problem from Section VI; we set the clock period constraint to 4 ns and the timeout of the solver to 3 minutes. We verify all designs in ModelSim [22] and use it to obtain the clock cycle count; we present our area and timing results post-place-and-route with Vivado [23].

We evaluate a selection of integer and floating-point kernels from typical HLS suites [24, 25] and recent works on dynamic scheduling [3, 4]; the source codes of our benchmarks are publicly available [26]: (1) *gsum* and *gsumif* are typical cases where dynamic scheduling is superior to static; they contain loops with irregular control flow and conditional loop-carried dependencies that prevent static HLS from pipelining the designs. (2) *getTanh* and *histogram* have irregular memory accesses that cause static HLS to make conservative scheduling assumptions; instead, dataflow circuits use a *load-store queue* (LSQ) [19] to dynamically resolve them. (3) All other benchmarks are standard HLS kernels where static and dynamic HLS are expected to achieve qualitatively similar pipelines; however, as discussed before, the performance of dataflow circuits typically suffers from frequency degradation, thus making them an undesirable solution [4].

B. Results: Comparison with Standard Datapath Optimization

Table I details the comparison of our circuits (DVRM) with the approach of Dynamatic (Naive); the CP column indicates the achieved clock period. The Naive method does not capture any of the combinational delays of the control network nor its interactions with the datapath. Thus, it fails in meeting the CP target (4 ns) in the majority of the benchmarks. In contrast, DVRM models delays in the data, valid, ready, and mixed domain, and finds a superior buffer placement that meets the clock period target in all our benchmarks. The exceptions are *getTanh* and *histogram*, where the CP target is violated due the LSQ at the memory interface, whose critical path is larger than our CP target and not optimizable by our strategy. To confirm this intuition, we synthesized the LSQ of the *getTanh* benchmark in isolation and measured an internal delay of 4.56 ns. We note that the problem of effectively pipelining the internal structure of the LSQ is orthogonal to our contribution: accurate delay regulation within a dataflow network.

Our strategy typically places more buffers into the circuit than the Naive approach, thus resulting in higher FF and LUT

Benchmark	Method	CP (ns)	Cycles	Exec. Time (μs)	Exec. Time Ratio	DSPs	DSP Ratio	LUTs	LUTs Ratio	FFs	FFs Ratio
gsum	Naive	5.22	5264	27.50	-	22	-	2053	-	2765	-
	DVRM (Our Work)	**3.78**	5371	20.30	D/N=0.74	22	D/N=1.00	2177	D/N=1.06	3067	D/N=1.11
	Static	3.36	10067	33.80	D/S=0.60	5	D/S=4.40	600	D/S=3.63	1023	D/S=3.00
gsumif	Naive	5.58	5192	29.00	-	26	-	2594	-	3478	-
	DVRM (Our Work)	**3.66**	5263	19.30	D/N=0.66	26	D/N=1.00	2758	D/N=1.06	4175	D/N=1.20
	Static	3.18	10047	32.00	D/S=0.60	4	D/S=6.50	579	D/S=4.76	963	D/S=4.34
getTanh	Naive	6.17	8174	50.40	-	16	-	2619	-	2363	-
	DVRM (Our Work)	**5.29**	8185	43.30	D/N=0.86	16	D/N=1.00	2644	D/N=1.01	2679	D/N=1.13
	Static	3.17	47003	149.00	D/S=0.29	5	D/S=3.20	425	D/S=6.22	576	D/S=4.65
histogram	Naive	4.56	4028	18.40	-	2	-	2001	-	1611	-
	DVRM (Our Work)	**4.94**	4029	19.90	D/N=1.08	2	D/N=1.00	2037	D/N=1.02	1709	D/N=1.06
	Static	3.51	13003	45.70	D/S=0.44	2	D/S=1.00	286	D/S=7.12	510	D/S=3.35
2mm	Naive	4.71	9287	43.70	-	16	-	2948	-	3281	-
	DVRM (Our Work)	**3.52**	9293	32.80	D/N=0.75	16	D/N=1.00	3146	D/N=1.07	4085	D/N=1.25
	Static	3.27	12930	42.30	D/S=0.77	5	D/S=3.20	612	D/S=5.14	939	D/S=4.35
3mm	Naive	4.44	14021	62.20	-	15	-	2922	-	3244	-
	DVRM (Our Work)	**3.73**	14216	53.10	D/N=0.85	15	D/N=1.00	3327	D/N=1.14	4018	D/N=1.24
	Static	3.39	18243	61.80	D/S=0.86	5	D/S=3.00	667	D/S=4.99	1068	D/S=3.76
mvt	Naive	4.12	20041	82.50	-	10	-	1673	-	2039	-
	DVRM (Our Work)	**3.47**	20106	69.80	D/N=0.85	10	D/N=1.00	1780	D/N=1.06	2384	D/N=1.17
	Static	3.28	21378	70.20	D/S=0.99	5	D/S=2.00	516	D/S=3.45	873	D/S=2.73
covariance	Naive	4.44	178463	792.90	-	12	-	2313	-	2499	-
	DVRM (Our Work)	**3.50**	178457	624.80	D/N=0.79	12	D/N=1.00	2447	D/N=1.06	2905	D/N=1.16
	Static	3.39	187178	635.10	D/S=0.98	5	D/S=2.40	697	D/S=3.51	994	D/S=2.92
gaussian	Naive	3.58	3738	13.40	-	3	-	677	-	511	-
	DVRM (Our Work)	**3.42**	3334	11.40	D/N=0.85	3	D/N=1.00	705	D/N=1.04	691	D/N=1.35
	Static	3.59	3294	11.80	D/S=0.97	3	D/S=1.00	180	D/S=3.92	341	D/S=2.03
matrix	Naive	3.91	68708	268.40	-	3	-	635	-	522	-
	DVRM (Our Work)	**3.39**	68710	233.10	D/N=0.87	3	D/N=1.00	715	D/N=1.13	737	D/N=1.41
	Static	3.28	65545	215.20	D/S=1.08	3	D/S=1.00	155	D/S=4.61	300	D/S=2.46
gemver	Naive	4.44	6508	28.90	-	18	-	2619	-	2363	-
	DVRM (Our Work)	**3.55**	6511	23.10	D/N=0.80	18	D/N=1.00	2644	D/N=1.01	2679	D/N=1.13
	Static	3.10	4301	13.40	D/S=1.73	18	D/S=3.20	425	D/S=6.22	576	D/S=4.65
kmp	Naive	6.38	8621	55.00	-	0	-	2601	-	2631	-
	DVRM (Our Work)	**3.95**	9625	38.00	D/N=0.69	0	D/N=0	2763	D/N=1.06	3232	D/N=1.23
	Static	3.23	8358	27.00	D/S=1.41	0	D/S=0	225	D/S=12.28	435	D/S=7.43

TABLE I: Timing and area comparison of dataflow circuits optimized using the strategy implemented in Dynamatic (Naive), our optimization (DVRM), and circuits by Vivado HLS (Static). All results are obtained post-place-and-route, for a target CP of 4 ns. Our circuits always succeed in meeting the clock period target; the exceptions are *getTanh* and *histogram*, where the CP increase is due to the load-store queue whose internal critical path is larger than our target; this design aspect is well-known [19] and orthogonal to our work.

requirements. This effect is completely expected, as the Naive strategy fails to recognize places where buffers are needed and thus omits them. Instead, we accurately determine the buffer positions and insert them to precisely control the critical path, thus mitigating the unpredictable CP variability that the Naive solutions suffer from. It is important to note that, despite the increased buffering, our strategy does not compromise design throughput and the clock cycle count remains essentially unchanged (apart from minor alterations due to the changes in datapath latency). Consequently, our solutions achieve significant improvements in overall execution time.

C. Results: Comparison with Static Scheduling

Dynamically scheduled circuits typically suffer from increased critical paths compared to their statically scheduled counterparts [3] which reduces their benefits in applications where dynamic scheduling is useful and makes them undesirable in cases where static and dynamic scheduling achieve similar pipelines. In this section, we thus compare the circuits optimized with our timing strategy with the results of a state-of-the-art HLS tool [27]. It is important to note upfront that dynamic scheduling typically requires significantly more resources than statically scheduled circuits, as others have noted before us [4, 10]. Optimizing dataflow resources is not the focus of this work; we here aim to: (1) show that our ability to meet the CP target is equivalent to that of static HLS, and (2) investigate how the improved performance of dynamic circuits compares to a state-of-the-art static approach.

Figure 7 visualizes the dataflow solutions normalized to the corresponding static designs (all in point (1,1)); Table I details the Vivado HLS results (Static). In the graphs, the matching dataflow solutions are connected with an arrow, such that the arrow base and point show Naive and DVRM designs, respectively. All points left of the vertical dashed line have a lower execution time than Vivado.

While both Static and DVRM solutions meet the CP target (apart from the two cases discussed above), Static solutions are sometimes pipelined in an over-conservative manner: their CP is lower but the cycle count increases due to additional pipeline stages needed to achieve it. In contrast, our solutions are more precise in meeting the CP target. In benchmarks that profit from dynamic scheduling (e.g., *gsum, gsumif*), Naive solutions are superior to Static in overall execution time; our performance optimization lowers the CP and thus further increases the achieved speedups (i.e., the points move further left in the graph). When dynamic scheduling achieves lower throughput than static (due to the sometimes conservative dataflow circuit construction strategy [10] or the additional buffers that their more complex datapaths require), our approach lowers the CP and successfully reduces the execution time difference

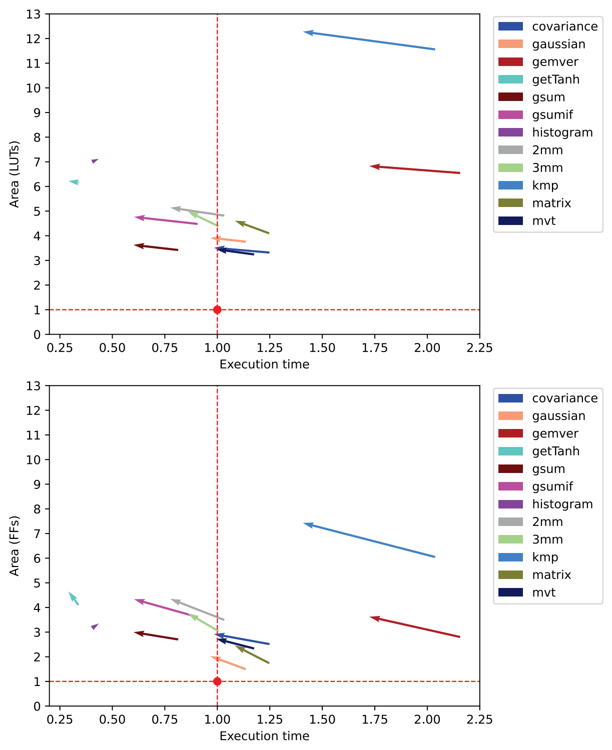


Figure 7: Execution time and area (LUTs, FFs) of dataflow circuits, normalized to the Static solutions. The base and point of the arrows show the Naive and DVRM (our solutions) points, respectively. DVRM enables the majority of the benchmarks to achieve equal or better performance than the Static circuits.

(e.g., *kmp*, *gemver*). Most importantly, in cases where static and dynamic scheduling have similar pipelining capabilities and, consequently, cycle counts, the ability of DVRM to meet the CP target enables the majority of our benchmarks to meet the execution time of the Static designs (e.g., *covariance, mvt* move to the red dashed line) or even to outperform them (e.g., *2mm, 3mm, gaussian* cross the red dashed line). All these results indicate that our approach significantly advances the usability of dynamic scheduling in diverse applications.

D. Results: Effectiveness of the Clock Period Constraints

Table II shows how the achievable CP changes with the inclusion of particular sets of timing constraints from Section VI.

The Naive strategy is what we discussed in Table I and its results are shown as $\mathrm{CP}_{\mathrm{Naive}}$; the following columns gradually include the constraints and delays that correspond to the valid, ready, and mixed domains ($\mathrm{CP}_{\mathrm{DVRM}}$ is our complete optimization model, equivalent to our results from Table I). The CP values gradually improve with the increased number of considered timing domains, as additional combinational paths are identified and optimized. Yet, it is only the final and complete model that consistently reaches the target CP. This shows that the complex combinational paths that span through the different timing domains (see Figure 1) indeed contribute to the critical path of the circuit and confirms the necessity of employing our complete timing model from Section VI.

Benchmark	$\mathrm{CP}_{\mathrm{Naive}}$	$\mathrm{CP}_{\mathrm{DV}}$	$\mathrm{CP}_{\mathrm{DVR}}$	$\mathrm{CP}_{\mathrm{DVRM}}$	$\mathrm{MILP}_{\mathrm{Naive}}$	$\mathrm{MILP}_{\mathrm{DVRM}}$
gsum	5.22	4.64	4.35	3.78	1	8
gsumif	5.59	5.64	4.63	3.66	11	180
getTanh	6.17	6.36	6.51	5.29	1	2
histogram	4.56	5.26	5.16	4.94	1	1
2mm	4.71	5.22	3.74	3.53	361	361
3mm	4.44	4.44	3.87	3.73	541	529
mvt	4.12	3.84	3.74	3.47	6	13
covariance	4.44	4.35	3.97	3.50	183	197
gaussian	3.59	3.64	3.61	3.42	73	64
matrix	3.91	4.10	3.59	3.39	180	103
gemver	4.44	4.82	4.05	3.55	18	17
kmp	6.39	5.24	5.22	3.95	324	201

TABLE II: CP (ns) achieved by different timing models. Naive and DVRM are as in Table I, and DV and DVR are intermediate points of our approach that gradually include different timing domains. The results confirm that a complete (DVRM) model is needed to achieve accurate CPs. The MILP columns show the MILP runtime (s); our runtimes do not deviate significantly from the Naive approach.

E. Results: MILP Runtime

Our optimization strategy introduces new MILP constraints and variables, thus increasing the size of the MILP problem compared to that of Dynamatic. We here explore the impact of these additions on the MILP runtime.

The final two columns of Table II indicate the MILP runtimes of Naive ($\mathrm{MILP}_{\mathrm{Naive}}$) and our approach ($\mathrm{MILP}_{\mathrm{DVRM}}$). The only case where the runtime notably increases is *gsumif*, as the new variables increase the problem size and complexify the solution search. In the majority of cases, the MILP runtime does not change significantly or even reduces; the typically minor variations are mostly accidental and due to particular solutions that the different constraints enforce (e.g., the additional constraints of DVRM may enforce a particular buffer placement, thus reducing the exploration space and, consequently, MILP runtime). The fact that our MILP formulation consistently sustains a reasonable runtime suggests its applicability to a wide variety of real-life benchmarks.

VIII. Conclusions

In this paper, we present a timing model for dataflow circuits that enables accurate buffer placement for critical path (i.e., operating frequency) regulation. Our model captures complex interactions between the datapath and the distributed dataflow control logic; it successfully identifies all combinational delays of the circuit and provides a reliable frequency estimate. We incorporate our timing model into a performance optimizer that maximizes circuit throughput for a given clock period constraint and show that our methodology achieves significant critical path reductions (and, consequently, performance speedups) over prior performance optimization techniques. In benchmarks where dynamic scheduling is useful, our strategy increases its performance benefits; in other cases, it makes dynamic circuits competitive with static circuits. Thus, our approach is a promising step in making dynamic scheduling broadly usable in the context of high-level synthesis.

References

[1] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proceedings of the 43rd Design Automation Conference*, San Francisco, Calif., Jul. 2006, pp. 657–62.

[2] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–76, Sep. 2001.

[3] 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, Calif., Feb. 2018, pp. 127–36.

[4] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, Calif., Feb. 2020, pp. 288–98.

[5] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2013, pp. 171–80.

[6] L. Josipović, A. Guerrieri, and P. Ienne, "From C/C++ code to high-performance dataflow circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2142–55, Jul. 2022.

[7] 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, Calif., Feb. 2020, pp. 1–10.

[8] *http://www.llvm.org*, The LLVM Compiler Infrastructure, 2018. [Online]. Available: http://www.llvm.org

[9] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," *Proceedings of the 27th International Conference on Computer-Aided Design*, pp. 434–41, Nov. 2008.

[10] 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, Calif., Feb. 2020, pp. 186–96.

[11] 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, Calif., Nov. 2013, pp. 211–18.

[12] 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, Calif., Jul. 2006, pp. 433–38.

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

[13] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed Petri nets," Massachusetts Institute of Technology, Tech. Rep. Project MAC Technical Report 120, Feb. 1974.

[14] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets." *IEEE Transactions on Software Engineering*, vol. 6, no. 5, pp. 440–49, Sep. 1980.

[15] J. Campos, G. Chiola, J. M. Colom, and M. Silva, "Properties and performance bounds for timed marked graphs," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 39, no. 5, pp. 386–401, May 1992.

[16] 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, Calif., Nov. 2013, pp. 219–25.

[17] 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, Calif., Nov. 2006, pp. 724–29.

[18] 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, Calif., Nov. 2007, pp. 362–69.

[19] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 125:1–125:19, Sep. 2017.

[20] 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, Tex., Mar. 2005, pp. 177–86.

[22] Mentor Graphics, "ModelSim," 2016. [Online]. Available: https://www.mentor.com/products/fv/modelsim/

[23] *Vivado Design Suite*, Xilinx Inc., 2020. [Online]. Available: https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis

[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, North Carolina, October 2014.

[26] "DVRM benchmark suite," Jun. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6759150

[27] *Vivado High-Level Synthesis*, Xilinx Inc., 2018. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html