Buffer Placement and Sizing for High-Performance Dataflow Circuits

LANA JOSIPOVIĆ, SHABNAM SHEIKHHA, ANDREA GUERRIERI, and
PAOLO IENNE, École Polytechnique Fédérale de Lausanne
JORDI CORTADELLA, Universitat Politècnica de Catalunya

Commercial high-level synthesis tools typically produce statically scheduled circuits. Yet, effective C-to-circuit conversion of arbitrary software applications calls for dataflow circuits, as they can handle efficiently variable latencies (e.g., caches), unpredictable memory dependencies, and irregular control flow. Dataflow circuits exhibit an unconventional property: registers (usually referred to as "buffers") can be placed anywhere in the circuit without changing its semantics, in strong contrast to what happens in traditional datapaths. Yet, although functionally irrelevant, this placement has a significant impact on the circuit's timing and throughput. In this work, we show how to strategically place buffers into a dataflow circuit to optimize its performance. Our approach extracts a set of choice-free critical loops from arbitrary dataflow circuits and relies on the theory of marked graphs to optimize the buffer placement and sizing. Our performance optimization model supports important high-level synthesis features such as pipelined computational units, units with variable latency and throughput, and if-conversion. We demonstrate the performance benefits of our approach on a set of dataflow circuits obtained from imperative code.

CCS Concepts: • **Hardware** → **High-level and register-transfer level synthesis**; Timing analysis; *Circuit optimization*; • **Computer systems organization** → *Data flow architectures*; • **Software and its engineering** → *Petri nets*

Additional Key Words and Phrases: Dataflow circuits, high-level synthesis, performance optimization, Petri nets

ACM Reference format:
Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2021. Buffer Placement and Sizing for High-Performance Dataflow Circuits. *ACM Trans. Reconfigurable Technol. Syst.* 15, 1, Article 4 (November 2021), 32 pages.
https://doi.org/10.1145/3477053

1 INTRODUCTION

Standard **high-level synthesis (HLS)** tools [6, 43] rely on static scheduling: the clock cycle in which each operation executes is decided during compilation. To increase parallelism, these tools

L. Josipović was supported by a Google Ph.D. Fellowship in Systems and Networking. J. Cortadella was supported by grants MINECO TIN2017-86727-C2-1-R and GENCAT 2017-SGR-786.
Authors' addresses: L. Josipović, S. Sheikhha, A. Guerrieri, and P. Ienne, École polytechnique fédérale de Lausanne, School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland; emails: lana.josipovic@epfl.ch; J. Cortadella, Universitat Politécnica de Catalunya, Department of Computer Science, Jordi Girona Salgado 1–3, 08034 Barcelona, Spain; email: jordi.cortadella@upc.edu.

1936-7406/2021/11-ART4 $15.00
https://doi.org/10.1145/3477053

use techniques such as loop pipelining to overlap statically scheduled loop iterations at a constant loop initiation interval [5, 36, 44]. Yet, in the presence of memory and control dependencies that cannot be determined at compile time, this scheduling approach forces worst-case assumptions and results in suboptimal schedules. In contrast, dataflow or latency-insensitive circuits [7, 14, 16, 41] implement dynamic scheduling and can resolve such dependencies as the circuit runs, thus achieving better performance. Although these circuits are naturally capable of overlapping loop iterations, their pipelining abilities critically depend on the placement and sizing of buffers. Some approaches build dataflow circuits out of imperative code [2, 23], but they still rely on crude heuristics and manual tuning to optimize performance, hence often producing suboptimal circuits.

If dataflow circuits are to play a significant role in the development of HLS, they need to benefit from every optimization opportunity that standard HLS techniques regularly exploit. In this work, we simultaneously tackle two aspects that are crucial for achieving high-performance circuits: constraining the critical path and maximizing throughput. We discuss the difficulties of performing such optimizations in the context of dataflow designs and present a performance optimization model based on marked graph theory that achieves maximum circuit parallelism at the desired clock frequency and with minimal resource cost. We discuss the scalability of our optimization approach and present a set of techniques to reduce the runtime of our performance optimization while still achieving near-optimal results. In addition to our previously published work [25], this article enhances our performance optimization model to support several concepts that are critical in handling typical HLS circuits and crucial to fully benefit from dataflow design, such as computational units with variable latency and throughput as well as if-conversion. We extend our evaluation section to demonstrate the effectiveness of our approach in such situations; our enhancements result in speedups of up to 11.6× compared to previous dataflow solutions and achieve up to 13× higher throughput than the corresponding statically scheduled HLS circuits.

2 BACKGROUND AND MOTIVATION

In this section, we discuss structural aspects of dataflow circuits generated from imperative code and emphasize the importance of buffer placement for obtaining high-performance designs. We describe marked graphs, a particular class of Petri nets, which are the basis for the performance model we introduce in Section 3.

2.1 Dataflow Circuits

Latency-insensitive protocols [7, 14] are a natural method to create synchronous dataflow circuits, capable of making decisions at runtime. Such circuits are built out of *units* that implement latency-insensitivity by communicating with their predecessors and successors through *channels* composed of data lines and paired with handshake control signals: a *token* of data is propagated from unit to unit through a channel as soon as memory and control dependencies allow it—otherwise, it is stalled by the handshake mechanism.

In this work, we rely on existing methodologies for generating dataflow circuits out of high-level code [15, 20, 23, 37]: our circuits consist of an interconnect of subcircuits obtained from **basic blocks (BBs)** (i.e., straight pieces of code separated by control flow decisions). Each BB subcircuit is a directed acyclic graph of dataflow units. The following units implement control flow statements [10, 19]: (1) a *merge* propagates a token received from any of the predecessor BBs into its BB body, and (2) a *branch* propagates a token from its input to one of the successor BBs based on a condition. Each BB contains as many merge units as it has incoming variables and as many branch units as it has outgoing variables [23]. Apart from the dataflow units implementing

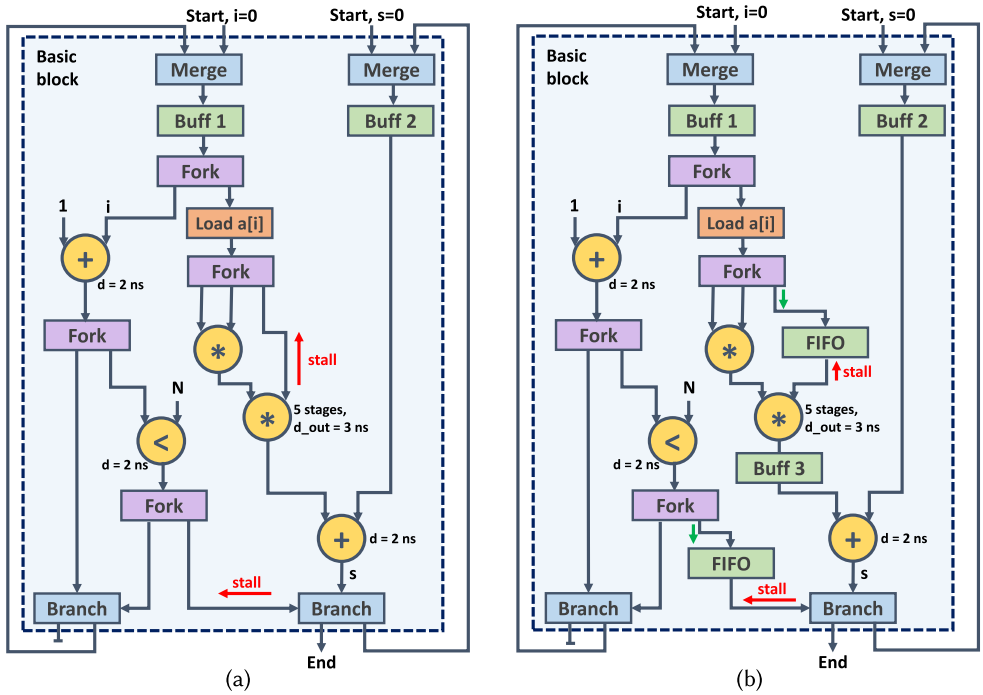


Fig. 1. A functionally correct but unoptimized dataflow circuit (a) implementing the code from (c) contains buffers (i.e., registers) placed to break all combinational loops. The optimized circuit (b) has buffers placed strategically to restrict the critical paths. Moreover, the FIFOs in the paths with higher latency mitigate backpressure and allow achieving the ideal loop initiation interval (in this example, equal to 1). The red (stall) arrows indicate the inability of the successor unit to accept a valid piece of data when it arrives; the green arrows in (b) indicate the ability of valid data to proceed into the FIFOs and to relieve backpressure from the upper fork.

control flow, BB subcircuits typically contain *forks* that replicate a token whenever it needs to be distributed to multiple successors in the same BB.

Figure 1(a) shows a dataflow circuit that calculates the sum of the cubes of N elements of an array. The initial values of the iterator i and the sum s are injected into the single BB of the circuit through their respective merges to trigger the computation start. The iterator is forked to a memory port to access an element of array a, which is sent to the pipelined multipliers to calculate the cube. The result is then added up with s. At the same time, the iterator value is incremented and compared to the loop bound. If the iterator has not reached the bound, the updated values of i and s are sent back through the branches to the merges, which triggers the start of the next loop iteration. Otherwise, the program terminates as the branch outputs the final value of s and the iterator is discarded into a sink.

Dataflow circuits require *buffers* that serve as registers in standard synchronous designs. Buffers store either *tokens* (i.e., valid data) or *bubbles* (i.e., invalid data). As in any circuit, all combinational

cycles[1] of a dataflow circuit must be cut with at least one buffer, as given in Figure 1(a). Yet, in contrast to standard registers, buffers can be placed on *any* channel of the dataflow circuit—this insertion will not compromise the functionality of the circuit due to its latency-insensitivity [3, 23] but will, in general, impact its timing and throughput.

2.2 Buffer Placement Is Crucial for Achieving High-Performance Dataflow Circuits

The circuit in Figure 1(a) is completely functional, as every combinational cycle contains a buffer to break the combinational loop. However, this circuit fails to address two important performance aspects. First, *critical path:* the buffers are placed without any consideration for the combinational delays of the nodes (all non-zero delays are indicated in the figure) and therefore do not restrict the critical path in any way. The critical path of 5 ns is the sum of the output delay of the pipelined multiplier with the delay of the adder. Second, *throughput:* a major performance limitation is caused by backpressure: some paths through the circuit take a longer time to process data and prevent the faster paths from consuming tokens at a higher rate. In Figure 1(a), the token carrying the array value *a[i]* is forked into two pipelined multipliers, but the lower multiplier cannot accept the token until the upper multiplier is done computing (i.e., after five clock cycles). Similarly, the condition token is sent to both branch nodes, but the right branch can accept the condition token only after the two chained five-stage multipliers produce a result. These stalls cause backpressure on their respective forks and prevent the short iterator path on the left from executing quickly: although a new iterator value could be computed on every clock cycle, the token with the updated value is stalled until the previous tokens have been consumed, which effectively increases the initiation interval of the loop, thus preventing loop pipelining.

Figure 1(b) shows a circuit configuration with optimal throughput and the critical path constrained to 4 ns. The additional buffer between the multiplier and the adder lowers the critical path. Inserting FIFOs into the paths with longer latency corresponds to *slack matching* [28] and increases effective parallelism, as accumulating data in the FIFOs allows to trigger the faster paths at a higher rate [23]. In this example, this is the case for the fast iterator path, which can now reenter the loop and trigger the start of a new loop iteration on every clock cycle, hence achieving a perfect pipeline with the initiation interval of 1.

2.3 Marked Graphs

Marked graphs are a class of Petri nets [29] that represent concurrent behavior but never have any *choices* (i.e., conditional execution). Figure 2 shows an example of a **choice-free dataflow circuit (CFDFC)** and its representation in the form of a marked graph. The buffer on the back edge of the circuit contains a token that infinitely loops through the combinational units: a token is forked from unit *n1* into both *n2* and *n4* concurrently, and the tokens from the two parallel paths are joined into a single token in *n5*—the transitions *n3* to *n5* and *n4* to *n5* always occur simultaneously. This concurrency property of marked graphs is the foundation of many linear algebraic techniques for their structural and performance analysis [4, 34, 35]; some address explicitly optimal buffer placement in choice-free dataflow graphs [3].

It is immediately clear that circuits such as the one in Figure 1(a) do not exhibit the choice-free behavior of marked graphs, as each control flow edge between BBs represents a choice: the merges in Figure 1(a) can accept the initial values of *i* and *s* from the starting point of the program, or the updated values sent back from the loop body; each branch can dispatch a value either through the back edge into the loop, or to the end point of the program, as determined by the branch condition.

[1]In the rest of this article, a *cycle* indicates a cyclic path of a data or a control flow graph, whereas we explicitly refer to *clock cycles* in the context of time (i.e., operation frequency of a circuit).

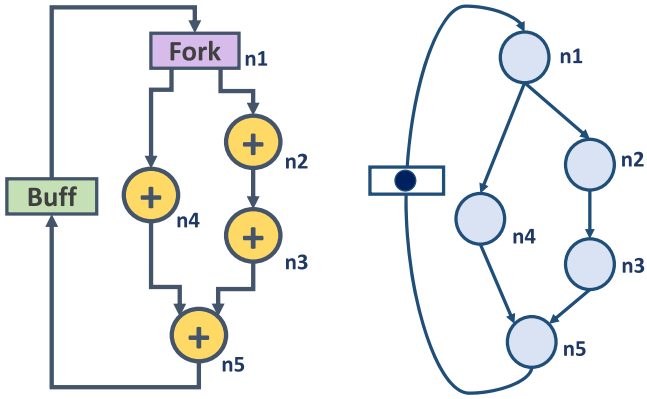


Fig. 2. A CFDFC and its representation as a marked graph. Circuits obtained out of high-level code (e.g., the ones in Figure 1(a) and (b)) contain choices (i.e., control flow decisions through merges and branches) and cannot be represented as marked graphs.

2.4 Key Intuition

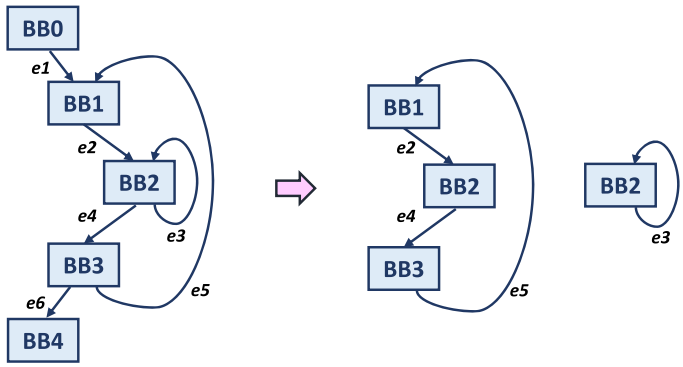
The performance of dataflow circuits such as the ones described in Section 2.1 critically depends on buffer placement and sizing, yet little is known about such optimizations. Yet, the timing properties of marked graphs have been extensively studied [3]. However, these techniques are not applicable in the context of dataflow circuits obtained from high-level code, which inevitably feature control flow and therefore cannot be represented as marked graphs. In this work, we combine the knowledge from marked graph theory with dataflow circuits that implement choices to optimize their performance: our work is based on the observation that choice-free subgraphs with the properties of marked graphs can be extracted out of generic dataflow graphs. We describe an approach to perform this extraction and adapt an existing performance optimization model for marked graphs [3] to target dataflow circuits produced out of high-level code. We extend this model to support several typical HLS features, such as pipelined computational units and if-conversion. Finally, we discuss the optimization of complex dataflow circuits as well as methods for ensuring scalability of our approach. We evaluate our technique on a set of benchmarks obtained out of C code.

3 OPTIMIZING PERFORMANCE

In this section, we describe our strategy for extracting probabilistically most significant choice-free subgraphs of a dataflow circuit. We introduce our performance optimization model for obtaining the optimal buffer placement and sizes such that (1) the required cycle period is satisfied and (2) the throughput of the choice-free circuits is maximized. We begin with the single most important subgraph and then extend the approach to multiple subgraphs.

3.1 Extracting CFDFCs

In this section, we describe our methodology for extracting the most significant CFDFC from a dataflow circuit. We define a CFDFC as a dataflow circuit obtained from a cycle of the **control flow graph (CFG)** (i.e., from a CFG subgraph in which each BB has exactly one input and one output edge). A CFDFC is therefore (1) choice-free (i.e., the CFDFC has no control flow decisions) and (2) strongly connected (i.e., the CFDFC implements a loop of the program); hence, it can be represented as a marked graph. Figure 3 shows a CFG of a nested loop: it contains two cycles that, internally, correspond to two CFDFCs.




```
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
      …
```

Fig. 3. Extracting CFG cycles. The leftmost graph is a CFG of a nested loop with two cycles (shown on the right of the figure), which we identify using an ILP-based approach. We then optimize CFDFCs that correspond to these cycles (i.e., the dataflow units and channels that belong to each CFG cycle, as illustrated in Figure 4).

The performance optimization that we will introduce in Section 3.3 optimizes the most frequently executed CFDFC. We identify this CFDFC by finding the most frequently executed CFG cycle using an **integer linear programming (ILP)** model.

The ILP we employ has the following constants and variables:

- $\mathbf{N}_e$ (constant): Execution frequency of control flow edge e (i.e., the total number of times e executes).
- S_e^E (variable, binary): Indicates whether the control flow edge e belongs to the selected CFG cycle.
- S_b^{BB} (variable, binary): Indicates whether BB b belongs to the selected CFG cycle.
- N (variable): Total number of times the CFG cycle executes.
- $\mathbf{N}_{\max}$ (constant): Upper bound on the number of executions, which has to be at least as large as the execution count of the most frequently executed edge of the CFG.

The following constraint states that the number of times the CFG cycle executes (N) corresponds to the minimum of the execution frequencies of the control flow edges that belong to it. For every edge e of the CFG,

$$N \leq S_e^E \cdot \mathbf{N}_e + \left(1 - S_e^E\right) \cdot \mathbf{N}_{\max}. \tag{1}$$

Here, $\mathbf{N}_{\max}$ ensures that N is not constrained by the execution frequencies of edges that do not belong to the loop.

If a BB is a part of the selected cycle, exactly one of its input and one of its output edges belong to the cycle as well. For every BB b of the CFG,

$$S_b^{BB} = \sum_{e \in In(b)} S_e^E \tag{2}$$

and

$$S_b^{BB} = \sum_{e \in Out(b)} S_e^E. \tag{3}$$

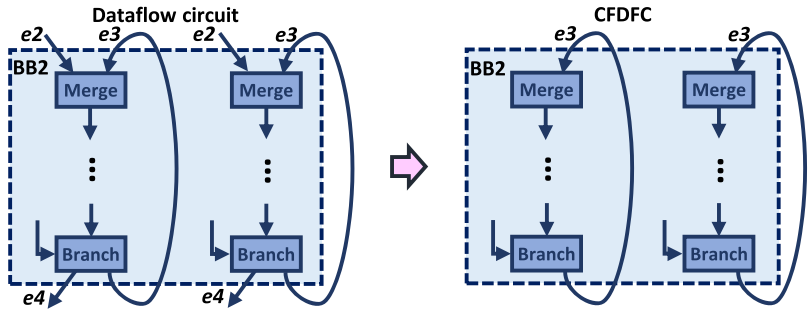


Fig. 4. Obtaining a CFDFC from a dataflow circuit. A CFDFC contains all dataflow units and channels that belong to any of the BBs or edges of the extracted CFG cycle (in this example, BB2 and edge e_3 from Figure 3). Every merge and branch in a CFDFC have a single input and output channel, respectively.

Here, $In(b)$ and $Out(b)$ denote the sets of input and output edges of BB b, respectively. We assume that BBs at the beginning and end of the program have respectively no input and no output edge.

To ensure that only a single cycle is selected, only a single back edge of the CFG may belong to it:

$$\sum_{e \in Back(CFG)} S_e^E = 1. \tag{4}$$

Here, $Back(CFG)$ denotes the set of all back edges of the CFG. Back edges are typically defined as edges that point from a BB to another BB that dominates it (i.e., from a BB inside a loop to the loop header); they can be detected using classical dataflow analysis [1].

We formulate the cost function to obtain the most frequently executed CFG cycle as follows:

$$\max : \quad \sum_{e \in CFG} N \cdot S_e^E. \tag{5}$$

Once this cycle is identified, it is straightforward to find the corresponding CFDFC with its dataflow units and channels. The following properties hold for every unit of the CFDFC: (1) for every merge, only one input channel belongs to the CFDFC (corresponding to the chosen input control flow edge of its BB); (2) for every branch, only one output channel belongs to the CFDFC (corresponding to the chosen output control flow edge of its BB); and (3) for all other units, all input and output channels belong to the CFDFC.

Figure 4 details the extraction of the most significant CFDFC of the program in Figure 3. The ILP will identify the self-loop of BB2 as the cycle with the highest execution frequency (i.e., the ILP result will be $S_{BB2}^{BB} = 1$, $S_{e3}^E = 1$, and all other BBs and edges are not selected). Therefore, for each merge in BB2, we keep only the input channel that originates from BB2 and belongs to edge *e3*; for each branch, we keep only the output channel leading back to BB2. All internal channels and units in BB2 belong to the CFDFC as well.

The approach that we have presented in this section will select one of the innermost loops of the circuit. We will extend our optimization model on multiple CFDFCs in Section 3.4.

3.2 Optimizing Choice-Free Circuits

The mathematical model presented in this article is based on the theory for performance analysis of concurrent systems inherited from timed Petri nets [3, 4, 34, 35]. We apply it to CFDFCs of the dataflow system, which can be represented as marked graphs (with functional units as nodes and channels as edges) to determine the optimal buffer placement and sizes.

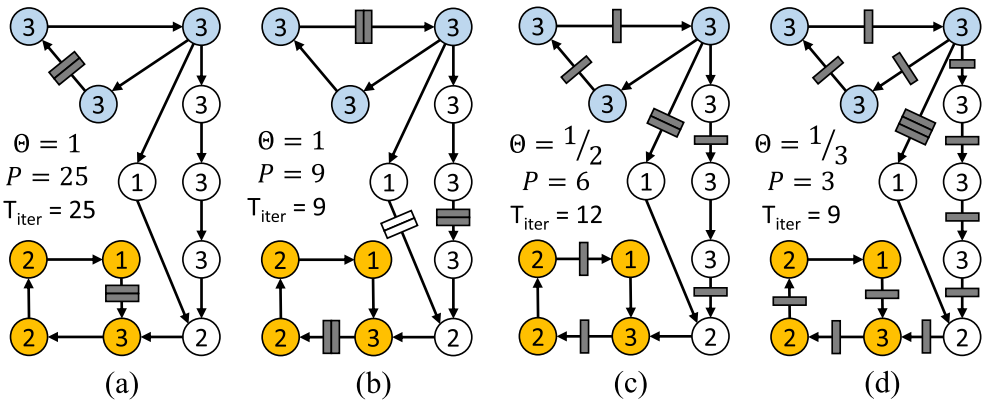


Fig. 5. Performance optimization of a CFDFC. Grey buffers are used for breaking combinational paths. The white buffer is transparent and used for throughput optimization. The circuits in the figure differ exclusively in buffering, which directly affects their timing (i.e., the achievable clock period, throughput, and overall iteration time).

A buffer can hold a token or a bubble—each time a token moves forward, a bubble moves in the opposite direction, similar to electrons and holes in semiconductors [18]. Every cycle of our circuit will always contain *at most one token* [23], whereas bubbles can be freely allocated by adding buffers without affecting functionality; to prevent deadlock, each cycle must contain at least one bubble so that the token and the bubble can always exchange places [3]. The buffers are located on the channels and characterized with two properties: (1) *transparency*, which indicates whether a buffer adds sequential delay onto a path (a nontransparent buffer is used to break the combinational delay and implies a one-cycle latency, whereas a transparent buffer is implemented as a pass-through element and does not increase cycle count), and (2) *capacity* (i.e., number of slots), which is used to regulate throughput. A single-slot, nontransparent buffer corresponds in functionality to a standard register—that is, it holds one data item that it can output on the next clock cycle and breaks a (potentially long) combinational path. A two-slot nontransparent buffer, sometimes referred to as elastic buffer [23], is what we have indicated as *Buff* in Figure 1(a)—it ensures that all combinational cycles are broken and that the cycle can accommodate a token and a bubble. A common FIFO of size N with a combinational path between input and output is here an N-slot transparent buffer.

The buffer configuration of a CFDFC determines its throughput Θ: every cycle of the circuit has a cycle ratio defined as the inverse of the number of nontransparent buffers and the throughput is limited by the cycle with the minimum cycle ratio [35]. As every cycle contains at least a single nontransparent buffer, the throughput equals at most 1 (i.e., $\Theta \leq 1$).

Figure 5 demonstrates the exploration space for the performance optimization of a CFDFC. In this example, there are two cycles that constrain the throughput. Every node (i.e., a functional unit of the dataflow circuit) is labeled with its combinational delay. Figure 5(a) shows a solution with maximum throughput ($\Theta = 1$) by only putting two-slot nontransparent buffers on the cycles, thus satisfying the requirement to accommodate a single token and a single bubble on each cycle. The cycle period **P** is then 25 and a cycle iteration takes 25 time units ($T_{iter} = P/\Theta$). Adding nontransparent buffers and moving the existing buffers reduces the critical path while maintaining the maximum throughput (Figure 5(b)). To prevent the topmost loop from stalling due to backpressure from the noncyclic path, an extra buffer (in white) has been added to one of the paths. Since it is not required to cut combinational paths, it can be implemented without adding any sequential delay (i.e., as a transparent buffer that acts as a FIFO, matching in size the nontransparent buffer on the

right). Constraining the system to work with $\mathbf{P} \leq 8$ requires the addition of nontransparent buffers on the cycles, thus degrading the throughput. Figure 5(c) shows a configuration with two buffers per cycle ($\Theta = 1/2$) and optimal period ($P = 6$) for this throughput, with $T_{iter} = 12$. Surprisingly, under the constraint $P \leq 8$, there is a more efficient configuration with lower throughput. The solution is shown in Figure 5(d) with $\Theta = 1/3$ and $\mathbf{P} = 3$, resulting in nine time units per iteration.

Solution (a) is the optimum in terms of area. Solution (b) is the optimum in terms of performance (T_{iter}) that minimizes area. Finally, solution (d) is the optimum in performance under the constraint $\mathbf{P} \leq 8$. The example shows the richness of solutions that can be explored in choice-free dataflow systems by changing exclusively the buffer positions and sizing—we will rely on this property to optimize the performance of our dataflow circuits.

3.3 MILP Model for Performance Optimization

We formulate our performance optimization model as a **mixed-integer linear programming (MILP)** model that determines the channels where buffers need to be placed as well as the buffer sizes. The model is based on the work of Bufistov et al. [3] for optimizing choice-free circuits—here we adapt it to generic dataflow graphs. We first present the model for a single CFDFC; in Section 3.4, we generalize our approach to multiple CFDFCs.

We class constants and variables of the MILP model into three groups: input constants (i.e., values given as input to the MILP), output variables (i.e., the solution of the buffer sizing problem), and internal variables (i.e., intermediate values found by the MILP solver but of little consequence to the user).

Input constants of the model.

- $\mathbf{P}$ (integer): Target **clock period (CP)** of the circuit.
- $\mathbf{P}_{\max}$ (integer): Upper bound on the CP of the circuit, which has to be at least as large as any possible value of $\mathbf{P}$.
- $\mathbf{B}_c$ (binary): Indicates whether channel c is a back edge ($\mathbf{B}_c = 1$) of the dataflow graph.
- $\mathbf{D}_u$ (real): Combinational delay of unit u.

Output variables of the model.

- R_c (binary): Indicates whether a sequential (nontransparent) buffer is present on channel c.
- N_c (integer): The number of slots of the buffer on channel c. The presence of a buffer implies at least one slot (i.e., $R_c \Rightarrow N_c > 0$). However, $N_c > 0$ and $R_c = 0$ indicates that a transparent buffer is present in the channel.

Internal variables of the model.

- Θ (real): Throughput of the CFDFC.
- $\overset{\bullet}{\Theta}_c$ (real): Average occupancy of channel c (token presence).
- $\overset{\circ}{\Theta}_c$ (real): Average emptiness of channel c (bubble presence).
- r_u (real): *Fluid* retiming of tokens across unit u.
- t_c^{in} (real): Arrival time at the the input of channel c (i.e., output of unit x, where $x \xrightarrow{c} y$).
- t_c^{out} (real): Arrival time at the output of the channel c (i.e, input of unit y, where $x \xrightarrow{c} y$).

We now describe the constraints of the MILP, grouped into path, throughput, and buffer sizing constraints.

Path constraints. These constraints ensure that the entire circuit meets the target CP. They are therefore applied to the *complete dataflow graph*. For every channel c of the dataflow graph,

$$t_c^{out} \geq t_c^{in} - \mathbf{P}_{\max} \cdot R_c, \tag{6}$$

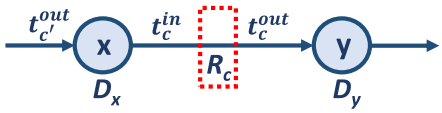


Fig. 6. Path constraints of the MILP model. These constraints ensure that the dataflow circuit meets the target CP by accumulating delays across channels and inserting buffers (indicated with $R_c = 1$) to break the combinational path.

with $t_c^{out} \geq 0$. This constraint, depicted in Figure 6, propagates the combinational arrival time at each channel. In case of the presence of a buffer ($R_c = 1$), the right term is guaranteed to be negative and t_c^{out} becomes zero, essentially disabling the further accumulation of delays through this channel. The constraint requires an upper bound of the maximum cycle period ($\mathbf{P}_{\max}$).

The following constraints model the propagation delay of each unit u of the dataflow graph with a pair of input/output channels $x \xrightarrow{c_1} u \xrightarrow{c_2} y$:

$$\mathbf{P} \geq t_{c_2}^{in} \geq t_{c_1}^{out} + \mathbf{D}_u. \tag{7}$$

The leftmost constraint enforces all delays to meet the cycle period $\mathbf{P}$. For simplicity, we assume that channels and buffers have zero delays. Channel, buffer setup, and clock-to-q delays could be easily incorporated into the model by adding the corresponding constants.

Throughput constraints. Our circuit construction guarantees that there is a single token on each cyclic path of a CFDFC. We initially consider that this token is placed on the back edge—once the buffers are assigned to the edges of the system, the throughput constraints will distribute the token across the cycle edges accordingly. These constraints are only applied to the *choice-free circuit (CFDFC)* obtained using the methodology described in Section 3.1. For every channel $u \xrightarrow{c} v$ in the CFDFC,

$$\dot{\Theta}_c = \mathbf{B}_c + r_v - r_u \tag{8}$$

$$\Theta \leq \dot{\Theta}_c \,/R_c. \tag{9}$$

The first constraint is analogous to the equations of classical retiming [27]; in this case, the variables are real instead of integers (i.e., fluid retiming) and $\dot{\Theta}_c$ represents the average number of tokens in the channel at the steady state of the system. The second constraint indicates that the system throughput is determined by the channel with a minimum average number of tokens among all channels with a nontransparent buffer. This constraint can be easily linearized taking into account that R_c is binary and $\Theta \leq 1$:

$$\Theta \leq \dot{\Theta}_c - R_c + 1. \tag{10}$$

For $R_c = 1$ (i.e., the channel contains a nontransparent buffer), the throughput is limited by the channel occupancy (i.e., $\Theta \leq \dot{\Theta}_c$). Otherwise (i.e., for $R_c = 0$), the throughput Θ is not constrained by the channel since the largest possible throughput value is 1 (and the right side of the equation will be greater or equal to 1).

Figure 7 demonstrates fluid token retiming based on the throughput and path constraints. The path constraints determine the buffer placement to achieve the target period of $\mathbf{P} \leq 3$. The values next to the buffers represent the token occupancies $\dot{\Theta}_c$. They indicate that every channel of the upper loop with a buffer will contain a token every one out of three clock cycles, whereas the buffer in the bottom left channel will contain a token two out of three clock cycles. The values

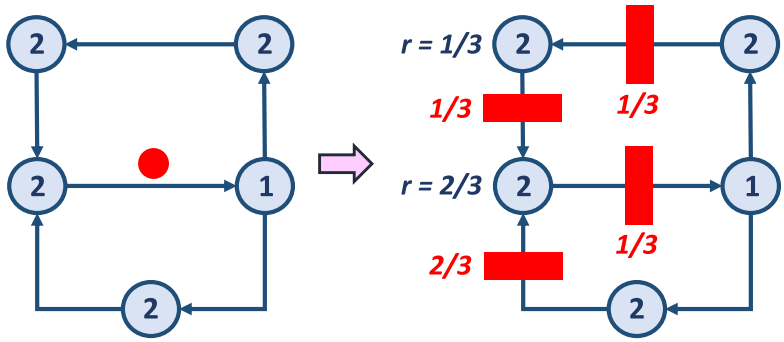


Fig. 7. Token retiming with throughput and path constraints for $\mathbf{P} \leq 3$. The token from the middle channel (left) is retimed during buffer assignment (right) and distributed in channels where the buffers are placed.

next to the units represent the retiming values r that indicate how much of the token must be retimed from the initial position (i.e., the middle channel) to achieve the average occupancies $\overset{\bullet}{\Theta}_c$. All values that are equal to zero are omitted from the figure.

Buffer sizing constraints. Buffer sizing is essential for avoiding backpressure. It corresponds to adding empty buffer slots, which do not affect circuit functionality. The average occupancy of tokens and bubbles will determine the number of buffer slots at every channel of the CFDFC:

$$N_c = \overset{\bullet}{\Theta}_c + \overset{\circ}{\Theta}_c . \tag{11}$$

The constraint for bubble occupancy is dual to that of token occupancy (and can be linearized in the same manner); it ensures that each cycle has at least one bubble, thus avoiding deadlock:

$$\Theta \leq \overset{\circ}{\Theta}_c \ / R_c . \tag{12}$$

Cost function. Subject to the path and the throughput constraints, we maximize throughput Θ for a given CP P while accounting for the minimization of the total number of buffer slots in the channels of the dataflow circuit:

$$\max : \quad \Theta - \lambda \cdot \sum_c N_c, \tag{13}$$

where λ is a small coefficient that gives a lower priority to the minimization of buffer sizes. As already mentioned, the path constraints include the complete dataflow graph, whereas the throughput constraints apply to the most frequently executed CFDFC.

In this work, without loss of generality, we focus on maximizing the throughput of the system. The model that we have presented in this section could easily be employed with different cost functions and optimization objectives (e.g., minimizing the CP or the buffer area cost under a throughput constraint [3]).

3.4 Optimizing Multiple CFDFCs

The model that we have presented so far optimizes only the single, most frequently executed CFDFC of the circuit. In this section, we extend our methodology to multiple CFDFCs.

We apply the ILP from Section 3.1 iteratively to extract one CFDFC after another based on their respective execution frequencies. After finding the most frequently executed CFG cycle, we update the execution frequencies by subtracting the execution values of the extracted CFG edges. Applying the ILP on the CFG while considering only the remaining execution values extracts the

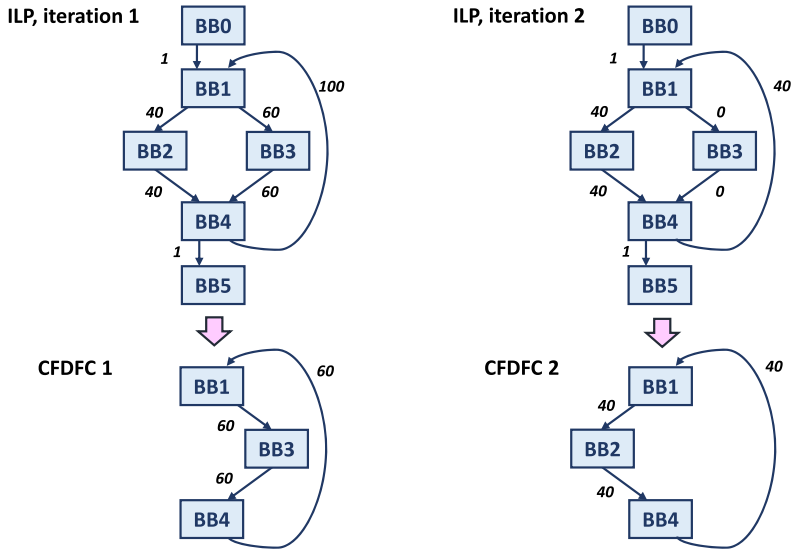


Fig. 8. Extracting multiple CFDFCs. The ILP from Section 3.1 can be iteratively applied while updating the execution frequencies of the CFG edges to extract one CFDFC after another. In the figure, the first extracted cycle (and the CFDFC that it represents) executes 60 times. After subtracting this value from the execution count of each corresponding CFG edge, we extract the next cycle of 40 iterations.

next cycle and its corresponding CFDFC based on its share in the runtime of the program. We illustrate this approach in Figure 8.

It is important to note that our ILP extracts cycles in the order of their importance (i.e., based on their fraction in the application runtime). We could also employ any algorithm for finding cycles in a directed graph [21], yet this approach would require extracting *all* graph cycles and subsequently sorting them based on their execution frequencies (by repeatedly identifying the most significant cycle and then updating the execution frequencies of all remaining cycles). The fact that our ILP simultaneously orders and extracts the cycles makes it possible to terminate the extraction as soon as appropriate criteria have been met (e.g., no remaining edge has an execution frequency above some threshold or the extracted cycles collectively represent a sufficient fraction of the application runtime). As we will discuss in Section 6, having such criteria is of great importance to limit the MILP runtime; moreover, the optimization of all CFDFCs is not always needed to maximize performance.

Optimizing multiple CFDFCs requires the extension of the MILP from Section 3.3 to maximize throughputs of all CFDFCs. For every additional CFDFC, the MILP includes an additional set of throughput and buffer sizing constraints (i.e., Equations (8) through (11)). The cost function to maximize system throughput considers a weighted sum of the individual throughputs Θ of all extracted CFDFCs:

$$\max : \sum_i w_i \cdot \Theta_i - \lambda \cdot \sum_c N_c, \tag{14}$$

where the weight w_i of each throughput is proportional to the frequency of execution of each CFDFC (i.e., an approximation of the runtime fraction of each CFDFC in the program profile).

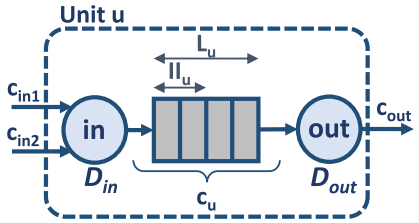


Fig. 9. An abstract model of a sequential (pipelined) unit.

4 MODELING COMPUTATIONAL UNITS AND IF-CONVERSION

The model that we have presented so far (as well as the model by Bufistov et al. [3] on which our work is based) only accounts for combinational nodes (i.e., combinational dataflow units). In this section, we extend the model to pipelined computational units and discuss how to apply it to units with variable II and latency. We then use these insights to model if-conversion, a typical HLS transformation.

4.1 Modeling Pipelined Units

To be able to handle cases such as the one in Figure 1, our MILP model needs to account for pipelined units. For this purpose, we characterize a pipelined unit u using two classic parameters: latency $\mathbf{L}_u$ and initiation interval $\mathbf{II}_u$. Figure 9 depicts our model.

The pipelined unit is modeled as two combinational units, separated by a channel c_u. The units *in* and *out* are represented with the unit input and output delay, $\mathbf{D}_{in}$ and $\mathbf{D}_{out}$. The channel c_u contains a nontransparent buffer with $\mathbf{L}_u$ slots, as the latency of the unit corresponds to the number of tokens the unit can hold. The delays $\mathbf{D}_{in}$ and $\mathbf{D}_{out}$ participate in the path constraints of the MILP (i.e., Equations (6) and (7)), like those of any other unit. The maximal operating frequency of the unit, $f_{u,max}$, can be neither modified nor optimized by the MILP, and hence we provide it directly as a constraint on the target CP (i.e., $1/f_{u,max} \leq \mathbf{P}$). Unless this constraint cannot be met (i.e., the MILP cannot find a feasible solution for the given target period), it has no impact on the buffer placement and sizing.

The initiation interval of unit u puts a constraint on the average presence of tokens in channel c_u that cannot be greater than $\mathbf{L}_u/\mathbf{II}_u$. Thus, throughput constraints for channel c_u can be written as follows:

$$\dot{\Theta}_u = r_{out} - r_{in} \tag{15}$$

and

$$\Theta \cdot \mathbf{L}_u \leq \dot{\Theta}_u \leq \mathbf{L}_u/\mathbf{II}_u, \tag{16}$$

where r_{in} and r_{out} are the corresponding retiming variables for the input and output combinational units, *in* and *out*.

4.2 Modeling Variable Initiation Interval

The model from Section 4.1 assumes a constant latency $\mathbf{L}_u$ and a constant initiation interval $\mathbf{II}_u$ for each computational unit. Yet, this may not always be the case—here we consider units with a variable initiation interval (i.e., $\mathbf{II}_u = [\mathbf{II}_{u,min}, \mathbf{II}_{u,max}]$). Typical examples of units that exhibit such behavior are read and write ports that connect to memory through a **load-store queue (LSQ)** [22]. The role of the LSQ is to ensure that all memory accesses with dependencies are executed in the correct order—based on the dependencies present in the program, the LSQ may issue and receive data from the memory ports at different rates, hence changing their effective initiation interval.

For instance, if a load has a read-after-write dependence with a previous store, the LSQ will return its data only after the store completes; this effect will temporarily lower the rate with which the load port issues data into the circuit, hence resulting in an increased II. However, if there are no dependencies, the load port can issue data at a high rate (i.e., with a low II, ideally equal to 1). Similarly, computational units with variable latency (which we discuss in detail in the following section) may also exhibit variable II: a longer latency of one computation implies the stall of new incoming data, thus temporarily increasing the unit II and, possibly, impacting the II of the entire circuit. For simplicity, here we discuss separately the concepts of variable II and variable latency, yet our technique is perfectly applicable to units that exhibit both of these properties.

Like any other sequential unit, units with variable II require the formulation of Equation (16), which connects the unit II, $\mathbf{II}_u$, to the CFDFC throughput (i.e., $\Theta \leq 1/\mathbf{II}_u$). A higher $\mathbf{II}_u$ value results in a tighter throughput constraint; hence, modeling a unit with an $\mathbf{II}_u$ value that is larger than the II achieved during execution may conservatively constrain the throughput Θ. Consequently, the resulting buffer configuration may be suboptimal and the achieved circuit performance may be limited. Therefore, the question here is how to choose the appropriate value of $\mathbf{II}_u$ for Equation (16) when the II of a unit is variable.

Consider the circuit in Figure 10, with the timing parameters of each unit listed under (a); all units have fixed latencies, but the II of the multiplier varies between 1 and 2. If we include into the MILP model the higher II, $\mathbf{II}_{mul,max} = 2$, it will constrain the CFDFC throughput to 1/2 and the buffers will be sized accordingly—in this case, the capacity of the transparent buffer before the store will be set to 3. Although this buffer capacity is sufficient to sustain the throughput of 1/2, it will cause backpressure when the multiplier operates with a lower II, hence *always* (i.e., regardless of the actual II that the multiplier achieves) constraining the throughput to 1/2. In contrast, optimizing the circuit for $\mathbf{II}_{u,min} = 1$ (and, consequently, the throughput of 1) will result in a larger buffer capacity (here equal to 6)—the buffer will be fully utilized when the throughput is equal to 1 and underutilized otherwise, but it will never limit the throughput and damage performance.

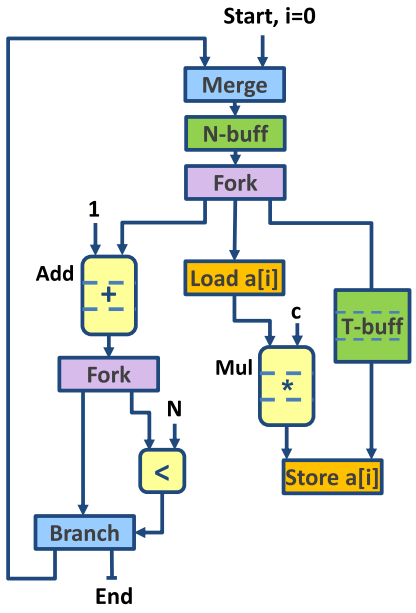
Given that our goal is to maximize throughput (and, consequently, performance), in Equation (16), we model each unit with its minimum initiation interval value, $\mathbf{II}_{u,min}$. Our MILP model from Section 3.3 remains unmodified by this enhancement.

4.3 Modeling Variable Latency

Our circuits may also contain variable-latency units: for instance, computational units that can take a variable number of cycles to compute the result depending on the input data (e.g., variable-latency adders [40] and multipliers [11, 31] that are optimized to compute the result quickly for the majority of input data but prolong the execution time for a small subset of inputs, thus realizing high performance on average) as well as load ports that wait a variable number of cycles for the memory to return the requested data. As in the case of variable II, the modeled latency value may have a significant impact on the circuit throughput and performance.

Consider the example in Figure 10, now with the parameters listed under (b), where now the multiplier latency ranges from one to four cycles. If the MILP model considers the minimum latency of 1, the capacity of the transparent buffer before the store will not suffice to eliminate backpressure when the multiplier latency is higher than 1. In contrast, if the model considers the maximum latency of 4, the MILP will produce a larger buffer that will relieve backpressure for every possible case, hence achieving the best possible performance.

The situation is different with the timing assumptions of bullet (c)—the variable-latency adder is on a cyclic path and its latency may constrain the CFDFC throughput. This effect occurs because there is always a single token on a cycle, as mentioned in Section 3.2; a long-latency unit on a cyclic



a) Variable II
Add: latency = 0, II = 1, Load: latency = 2, II = 1,
Mul: latency = 4, **II = 1-2**
For Mul II = 1 → T-buff size = 6, Θ_{tot}=1/2-1 ✓
For Mul II= 2 → T-buff size = 3, Θ_{tot}=1/2 ✗

b) Variable latency (noncyclic path)
Add: latency = 0, II = 1, Load: latency = 2, II = 1,
Mul **latency = 1-4**, II = 1
For Mul latency = 1 → T-buff size = 3, Θ_{tot}=1/2 ✗
For Mul latency = 4 → T-buff size = 6, Θ_{tot}=1 ✓

c) Variable latency (cyclic path)
Add: **latency = 1-4**, II = 1, Load: latency = 2, II = 1,
Mul: latency = 4, II = 1
For Add latency = 1 → T-buff size = 3, Θ_{tot}=1/2 ✓
For Add latency = 4 → T-buff size = 1, Θ_{tot}=1/5 ✗

Fig. 10. Modeling variable II (case a) and variable latency (cases b and c). When a unit has variable II, it is always desirable to model its best-case (i.e., lowest) II to achieve optimal buffer placement and, consequently, best possible performance. When a unit has variable latency, its model depends on whether the unit lies on a cyclic path; if so, it should be modeled with its minimum latency (so that it does not constrain throughput); otherwise, it is should be modeled with its maximum latency.

path limits the rate with which the token can reenter the loop body, therefore lowering the system throughput. Hence, considering the maximum latency value may create an overly conservative throughput constraint and buffer placement, similar to the effect discussed in the previous section (i.e, the resulting buffers will not be able to sustain higher throughput, achievable when the unit latency is lower). Hence, when a unit is on a cyclic path of the CFDFC, it is desirable to consider its minimal latency; the MILP will optimize the system for the largest achievable throughput and place buffers accordingly.

Therefore, to maximize performance, we model the latency of a variable-latency unit $\mathbf{L}_u = [\mathbf{L}_{u,min}, \mathbf{L}_{u,max}]$ in Equation (16) as follows: (1) if a unit is on a cyclic path of a CFDFC, we consider its minimum latency, $\mathbf{L}_{u,min}$, and (2) if a unit is on a noncyclic path, we consider its maximum latency, $\mathbf{L}_{u,max}$.

It is important to note that a unit may be on a cyclic path through one CFDFC but on a noncyclic path of another (e.g., a loop-carried dependency of an innermost loop may not be on the cyclic path through the outer loop). As each CFDFC has its own set of throughput constraints, independently modeling its units based on Equation (16), each CFDFC can consider the appropriate latency in its own unit constraint following the preceding rules to achieve the best possible throughput. If a unit has both variable II and variable latency, we choose the II following the rules from Section 4.2 and the latency following the rules described in this section—as suggested earlier, these decisions are completely independent. Our approach analyzes each unit separately to make the best possible II

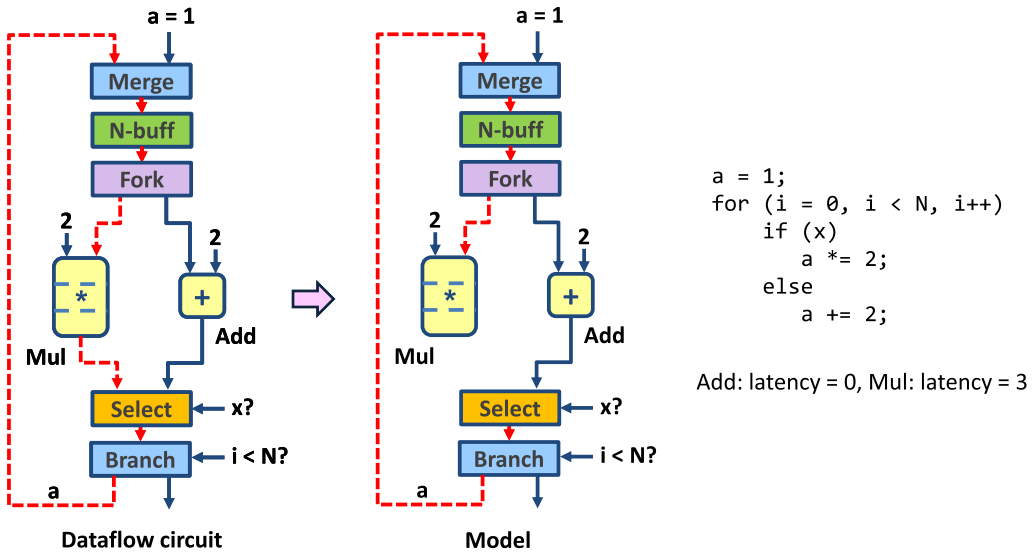


Fig. 11. Modeling if-conversion, implemented using a Select unit. The long-latency cycle (shown in red dashed) through the Select will limit achievable throughput, even if this input is never selected by the unit; hence, we omit this input from the model to place and size buffers for the best-case throughput.

and latency decision per unit and per CFDFC; therefore, it is naturally applicable to any number of units in the dataflow graph.

The rules presented in this and the previous section could easily be adapted to different cost functions and optimization objectives, discussed in Section 3.3. A possible modification would be to consider average latencies and initiation intervals; such an approach may result in lower resources (i.e., it may instantiate smaller buffers than the solutions we present here), but in contrast to our approach, it would not guarantee optimal performance for every possible outcome.

4.4 Modeling If-Conversion

Compilers typically rely on optimizations such as if-conversion to convert conditional branches into predicated instructions [38]; this transformation usually requires a dedicated instruction that chooses one of the input values based on a condition (i.e., a Select instruction). This instruction translates directly into the corresponding Select dataflow unit; it can be implemented as a multiplexer that outputs data as soon as the condition and the chosen input are available, whereas other inputs are simply discarded upon (possibly late) arrival [8, 13].

The performance optimization model that we have discussed so far considers all operations within a BB as choice-free units—all inputs must become available for the unit to trigger. This model is not suitable for a Select unit, which needs only one of its inputs (i.e., the input chosen by the condition) to trigger. If a Select is on a cycle and one of its inputs takes more cycles to compute than the other, our model as described so far would assume the worst-case latency and conservatively model throughput, even if the long-latency input may actually be discarded and the computation can proceed early on. This is the case for the circuit in Figure 11: even if the multiplier input is not selected, the MILP accounts for the long-latency cyclic path (shown in red dashed) and limits the obtainable throughput, exactly as described in the previous section. In the rest of this section, we further extend our model to avoid making conservative throughput assumptions in the presence of a Select unit.

It is interesting to note that the behavior of a Select unit corresponds to the behavior of a variable-latency unit; yet, instead of a single unit with varying latency, as we discussed in Section 4.3, the latency variability is now due to the different latencies of the two paths between the Fork and the Select (i.e., the path through the multiplier and the path through the adder). In this case, this variability cannot be resolved at unit level (i.e., adapting the timing parameters of the Select unit would not impact the modeled latency of its incoming paths). Instead, we exclude from the throughput constraints the input channel of the Select unit that is on the long-latency cycle, as indicated in Figure 11; essentially, the model will assume that this input is never taken and the best-case throughput will be computed accordingly.

To achieve the best possible throughput in the presence of a Select unit, we model the Select unit inputs as follows: (1) we include into the throughput constraints each Select input channel that is on a noncyclic path, as this latency will never compromise throughput; (2) if the Select has a single input on a cyclic path, we exclude the corresponding input channel from the throughput constraints; and (3) if the Select has both inputs on cyclic paths, we exclude the input channel on the cycle with more sequential stages. This situation corresponds to the one in Figure 11. Excluding a channel from the throughput constraints (i.e., Equations (8) through (10)) corresponds to predefining the token occupancy value of the channel $\dot{\Theta}_c$ to zero.

Note that this model produces optimal throughput regardless of which input is actually taken during circuit execution, as the buffer sizes are determined based on the highest possible throughput values—the produced buffering will support lower throughputs (potentially caused by the slower input which our model ignored) as well. It is interesting to note that the same effect could be achieved by handling the choices of the Select unit similar to the choices in the CFG graph—that is, by decoupling a CFDFC with a Select unit into two choice-free graphs (with each graph considering only one of the Select inputs). However, this strategy would increase the model complexity (by adding a new set of throughput constraints per input of each Select unit) while producing equivalent results.

5 SCALABILITY

In this section, we discuss the runtime complexity of the MILP model described in Section 3.3 and propose a technique to ensure scalability when optimizing complex circuits.

The ILP for cycle extraction operates on the CFG of the program, which usually covers a limited number of BBs and control flow edges. Hence, this ILP is typically of low complexity and size and it does not impact the overall algorithm runtime—we confirm experimentally this intuition in Section 6. However, the MILP for performance optimization operates on the dataflow graph of the program. While the throughput is optimized locally by applying the throughput constraints on subsets of the circuits (i.e., the frequently executed CFDFCs), the relations for path constraints (i.e., Equations (6) and (7)) extend on the *entire dataflow graph*—they need to ensure that the circuit as a whole meets the target period. The MILP size and runtime are therefore dependent on the overall number of channels and units of the dataflow circuit, which can result in large runtimes when optimizing complex designs.

A possible method to limit the MILP runtime of large applications is to split the dataflow graph into disjoint sets of CFDFCs (i.e., into CFDFCs obtained from strongly connected components of the CFG, which do not share any BBs or edges among each other) and to optimize them *separately* using the MILP. This procedure maximizes the throughput Θ and satisfies the period constraint $\mathbf{P}$ within the CFDFCs of each disjoint set. Afterward, we need to ensure that the *complete circuit* satisfies the period constraint. Hence, we apply the path constraints (i.e., Equations (6) and (7)) on the channels and units that were not covered by any of the CFDFC sets. The buffer placement

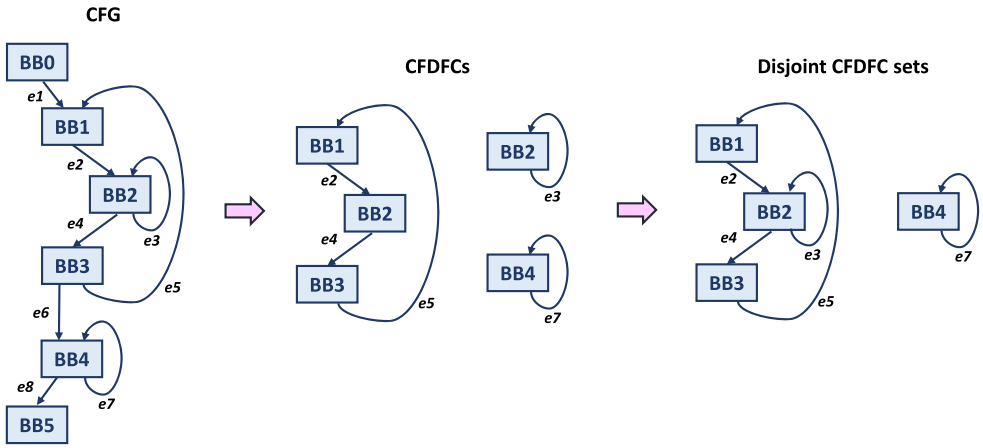


Fig. 12. Splitting the circuit into disjoint CFDFC sets to ensure MILP scalability. This circuit represented by the CFG in the figure consists of three CFDFCs that can be grouped into two disjoint sets. Applying the MILP on each set separately reduces the size of each MILP and decreases overall runtime.

solutions (i.e., the values of R_c) from the CFDFC set optimization are now set as constants to ensure that the combinational paths across set boundaries are appropriately handled. The channels optimized in this step do not need to be subject to any throughput constraints as they are of minimal importance for the overall performance (i.e., they usually belong to paths executed only a single time as the circuit runs); the sizes of all buffers correspondingly inserted can therefore be set to 1 (i.e., $N_c = 1$). The cost function of this final step minimizes the number of inserted buffers:

$$\min : \sum_c R_c. \tag{17}$$

Figure 12 illustrates this approach. The dataflow circuit represented by this CFG contains three CFDFCs—two of them share BBs and need to be optimized together. The third CFDFC (corresponding to BB4 in the figure) can be optimized separately. After solving the MILP for each of the two independent CFDFC sets, their throughput Θ will be maximized and each set will meet the target period $\mathbf{P}$. To ensure that the complete circuit respects $\mathbf{P}$, we subsequently need to optimize the remaining parts of the dataflow circuit (in this case, the channels within BB0 and BB5, as well as those corresponding to edges *e1*, *e6*, and *e8*) using only the path constraints.

In summary, applying the MILP on disjoint CFDFC sets reduces the problem complexity while satisfying the desired CP and achieving the same CFDFC throughput as the global MILP solution. We will show the effectiveness of this approach in Section 6.

6 EVALUATION

In this section, we demonstrate the ability of our optimization technique to maximize throughput under a given CP constraint. First, we compare our optimization approach with a naive buffer placement strategy. We then discuss the runtime of our performance optimization algorithm and present methods to improve it. Next, we investigate the ability of our technique to handle irregular and variable events. We explore the effectiveness of the CP constraint and compare the pipelining capabilities of our circuits with those produced by a standard HLS tool based on static scheduling.

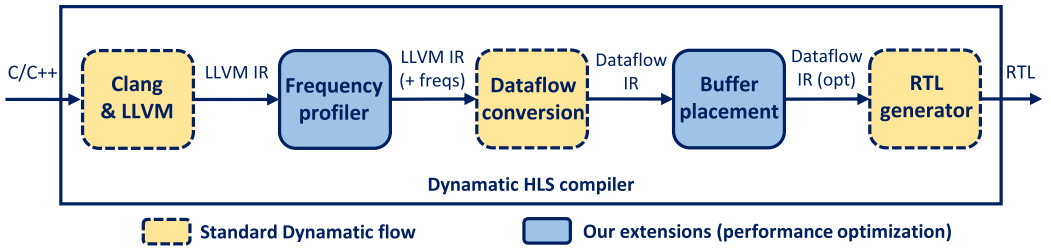


Fig. 13. Dynamatic compiler flow, extended with the performance optimization tool presented in this work. The LLVM IR profiler determines the execution frequencies of the CFG edges. The buffer placement tool uses these frequencies to identify CFDFCs, as discussed in Section 3.1, and inserts buffers to optimize performance using the MILP from Section 3.3.

6.1 Methodology

Our performance optimization model is fully integrated into Dynamatic, an open source HLS tool that produces synchronous dataflow circuits out of C code. The complete tool, with our modifications as well as the benchmarks we explore in this section, is publicly available at dynamatic.epfl.ch. Our modifications to the Dynamatic compiler are illustrated in Figure 13.

We profile the intermediate representation of the benchmarks to obtain the execution frequencies of the CFG edges—we insert counters into the IR code to count the control flow decisions taken in each executed BB and annotate the CFG with the obtained counts. After the IR has been transformed into a dataflow circuit, we use the information on execution frequencies to identify the CFDFCs using the ILP from Section 3.1. We then apply the MILP from Section 3.3 to determine the buffer placement and sizes that satisfy the target CP and maximize the loop throughputs—we employ the cost function from Equation (14). The weights of each throughput term are proportional to the runtime fraction of the corresponding CFDFC in the program profile and the number of units it contains (i.e., for CFDFC i, $w_i = units_i \cdot freq_i / freq_{tot}$). We choose the constant value of $\lambda = 10^{-5}$ to account for the minimization of buffer sizes. The MILP relies on static timing information about the unit delays—we consider exclusively the datapath of each unit. We present our results for two target periods: 4 ns and 3 ns—in the rest of this section, we denote the corresponding results as *MILP 4* and *MILP 3*.

To evaluate our technique, we use ModelSim to measure throughput (represented as the average loop initiation interval, $II = 1/\Theta$) and to verify functional correctness. We target a Xilinx Kintex-7 FPGA and use Vivado to measure the delays of the units. We obtain the CP and the resource usage after placement and routing. We use the CBC mixed-integer programming solver [9] and measure its runtime on an Intel Core i7-8550U CPU (i.e., a standard consumer laptop) at 1.80 GHz.

6.2 Benchmarks

We explore various kernels obtained from the literature [26, 33] and the PolyBench suite [32]. The benchmarks we consider contain pipelined computational units and exhibit different loop properties and organizations, as listed in Table 1: (1) *Sumi3* is the example kernel from Figure 1(c). *FIR* (Finite Impulse Response), *MatVec* (Matrix-Vector Multiplication), and *BiCG* (BiCGStab Linear Solver) are regular kernels implemented as a single loop or loop nest. *IIR* (Infinite Impulse Response) and *Cordic* (Coordinate Rotation DIgital Computer) have loop-carried dependencies that take multiple cycles to compute and therefore limit the achievable loop initiation interval. *Covar* and *Covar (f)* implement the integer and floating point version of the covariance computation, with and without multiple-cycle loop-carried dependencies, respectively. These two benchmarks

Table 1. Benchmark Characteristics: Their Set and CFDFC Count, the Main Property of the Loops That They Contain, and the Total Number of Dataflow Units and channels in the Corresponding Dataflow Circuit (Prior to Buffer Insertion)

Benchmark	Sets	CFDFCs	Property	Units	Channels
Sumi3	1	1	Regular	40	55
Fir	1	1	Regular	43	59
MatVec	1	2	Regular	80	113
BiCG	1	2	Regular	91	136
IIR	1	1	Loop-carried dep.	59	85
Cordic	1	1	Loop-carried dep.	91	129
Covar	3	7	Regular	262	378
Covar (f)	3	7	Loop-carried dep.	262	378
Gemver	4	7	Regular	316	483
CDiv	1	2	Conditional execution	135	201

as well as *Gemver* contain multiple loop nests (i.e., multiple CFDFC sets), as indicated in the table. Finally, *CDiv* calculates a complex quotient of complex numbers—the loop contains a noninlined if-else condition (i.e., it is implemented as two CFDFCs, similar to the example in Figure 8); we assume a data distribution where the if-condition is taken in 55% of the total loop iterations.

6.3 Comparison with Naive Buffer Placement

We demonstrate the performance superiority of our optimized circuits over equivalent designs with buffers placed naively, based on an existing heuristic [23] that cuts every combinational cycle with a single buffer and does not place any FIFOs into the designs.

Tables 2 and 3 show our main results. The circuits produced using the naive strategy (i.e., *Naive*) qualitatively correspond to the circuit in Figure 1(a): they contain the minimal number of buffers to create functional circuits (i.e., circuits with no combinational loops), but there is no way to control the critical path and backpressure significantly lowers throughput. In contrast, the designs optimized using our technique (i.e., *MILP 4* and *MILP 3*) are able to achieve maximum throughput (i.e., the best possible loop II) of the innermost loops. The resource increase (shown in Table 3) is due to the additional buffers that our technique employ, as indicated under *Buffers*. The designs with high throughput require transparent buffers of larger sizes (i.e., FIFOs) to maintain the token rate; those with a lower target CP need more nontransparent buffers to cut the combinational paths but use smaller buffer sizes due to the lowered throughput (consider, for instance, the buffer sizes in the *MILP 4* and *MILP 3* solutions of *Sumi3*). Setting a low target CP degrades throughput (as it requires the insertion of multiple nontransparent buffers on cyclic paths) and, consequently, performance, in all applications but the *IIR*, *Covar*, and *Covar (f)*. In these applications, the throughput is dictated by the pipelined units on the cyclic paths and not influenced by the additional buffers, so the total execution time benefits from the lowered CP. The discrepancies between the target and achieved CP are largely due to the timing variations caused by FPGA place-and-route. Our timing model could be further refined for greater accuracy without any qualitative change (e.g., by including setup delays of the buffers and considering the impact of control paths).

6.4 MILP Runtime Analysis

The rightmost column of Table 2 reports the runtime of the MILP for performance optimization. In all of our benchmarks, the runtime of the ILP for extracting the CFDFC was negligible (i.e., less than 1 second) in comparison to the MILP runtime. It is evident from the table that the MILP

Table 2. Timing of Dataflow Circuits Optimized with Our Strategy, *MILP 4* and *MILP 3*, with a Target CP of 4 ns and 3 ns, Respectively, Compared to a Naive Buffer Placement (*Naive*)

Benchmark	Method	CP (ns)	II= 1/Θ	Execution Time (μs)	Speedup	MILP Runtime (s)
Sumi3	Naive	4.3	10	43.0	–	–
	MILP 4	4	1	4.1	**10.6×**	0.1
	MILP 3	3.5	2	7.1	**6.1×**	1.2
FIR	Naive	4.3	6	25.8	–	–
	MILP 4	3.9	1	4.0	**6.5×**	0.1
	MILP 3	3.5	2	7.0	**3.7×**	0.8
MatVec	Naive	5.9	6	31.9	–	–
	MILP 4	4.8	1	4.5	**7.1×**	15.7
	MILP 3	3.9	2	7.3	**4.4×**	25.0
BiCG	Naive	6.0	6	32.4	–	–
	MILP 4	5.9	1	6.4	**5.0×**	1,328.4
	MILP 3	4.1	2	7.7	**4.2×**	2,195.5
IIR	Naive	5.9	6	35.4	–	–
	MILP 4	3.9	5	19.5	**1.8×**	1.1
	MILP 3	3.4	5	17.0	**2.1×**	20.1
Cordic	Naive	5.8	20	116.1	–	–
	MILP 4	4.5	20	87.8	**1.3×**	8.1
	MILP 3	5	20	97.6	**1.2×**	8.1
Covar	Naive	6.8	2, 4, 4	698.5	–	–
	MILP 4	6.5	1, 1, 1	197.5	**3.5×**	3,600
	MILP 3	5.6	2, 2, 2	182.9	**3.8×**	3,600
Covar (f)	Naive	7.1	11, 11, 17	1,833.6	–	–
	MILP 4	7.2	11, 1, 11	1,057.2	**1.7×**	3,600
	MILP 3	5.4	11, 2, 11	865.7	**2.1×**	3,600
Gemver	Naive	7.7	6, 10, 2, 10	180.7	–	–
	MILP 4	7.4	1, 1, 1, 1	23.5	**7.7×**	3,600
	MILP 3	5.5	2, 2, 2, 2	31.3	**5.8×**	3,600
CDiv	Naive	10.5	40, 40	787.9	–	–
	MILP 4	7.5	3, 3	23.3	**33.8×**	25.4
	MILP 3	7.2	5, 5	36.8	**21.4×**	153.9

Under *II*, we indicate the initiation intervals of the innermost loops. Although the MILP always finds a solution, the achieved CP is often larger than the target CP, due to the unavoidable approximation of the pre-synthesis and pre-place-and-route timing model. The rightmost column indicates the MILP runtime (the value of 3,600 indicates a timeout of 1 hour).

runtime significantly depends on the size and complexity of the application—larger applications need a prolonged MILP runtime because the MILP covers the units and channels of the entire dataflow graph, as discussed in Section 5. It is also interesting to note that the runtime depends on the optimization constraints (i.e., the target CP) and the achievable throughput—typically, the inability to achieve the best possible throughput increases the MILP runtime (i.e., MILP 3 solutions typically require a longer runtime than the corresponding higher-throughput MILP 4 solutions).

MILP solvers tend to find an acceptable solution (i.e., a near-optimal cost function value) early on and then spend a long time attempting to improve it. This effect is evident from Figure 14(a), which shows the obtained cost function value (i.e., the sum of the weighted CFDFC throughputs, as given in Equation (14)), relative to the optimal cost function value for a given target CP. The graph depicts only the results of the benchmarks that take longer than 1 second to converge to the optimum value of 1. Although it is clear that the convergence time is lower than the overall MILP

Table 3. Resources (i.e., LUTs, FFs, and DSPs) of Dataflow Circuits Optimized with Our Strategy, *MILP 4* and *MILP 3*, with a Target CP of 4 ns and 3 ns, Respectively, Compared to a Naive Buffer Placement (*Naive*)

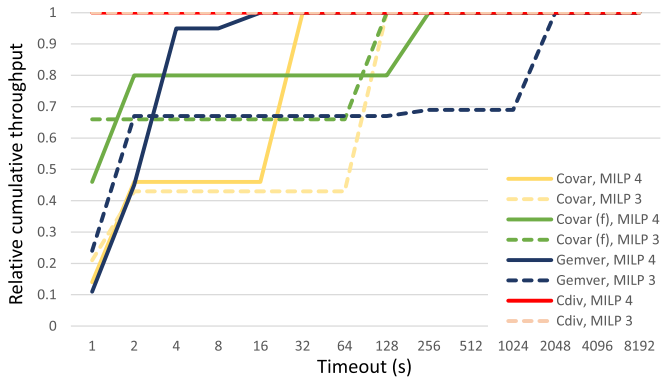
Benchmark	Method	LUTs	FFs	DSPs	Buffers
Sumi3	Naive	287	331	6	3 N2
	MILP 4	402 (+40%)	403 (+22%)	6	8 N1-2, T4, 2 T9
	MILP 3	413 (+44%)	522 (+58%)	6	10 N1-2, 2 T1-2, 2 T5
FIR	Naive	380	384	3	3 N2
	MILP 4	428 (+13%)	504 (+31%)	3	5 N1-2, 2 T6-7
	MILP 3	628 (+65%)	688 (+79%)	3	7 N1-2, 2 T4-5
MatVec	Naive	626	517	3	6 N2
	MILP 4	808 (+29%)	724 (+40%)	3	11 N1-2, 5 T3-8
	MILP 3	947 (+51%)	849 (+64%)	3	16 N1-2, N4, 6 T1-3
BiCG	Naive	831	758	6	6 N2
	MILP 4	1,144 (+38%)	1,157 (+53%)	6	16 N1-3, 7 T1-3, 4 T5-7
	MILP 3	1,140 (+37%)	1,255 (+66%)	6	14 N1-2, 2 N4-5, 8 T1-3
IIR	Naive	648	663	6	5 N2
	MILP 4	745 (+15%)	1,096 (+65%)	6	10 N1-2, 6 T1-2
	MILP 3	772 (+19%)	1,094 (+65%)	6	12 N1-2, 5 T1-2
Cordic	Naive	1,950	2,754	24	7 N2
	MILP 4	2,075 (+6%)	3,086 (+12%)	24	16 N1-2, 9 T1-2
	MILP 3	2,145 (+10%)	3,016 (+10%)	24	17 N1-2, 9 T1-2
Covar	Naive	2,347	1,801	3	23 N2
	MILP 4	3,882 (+65%)	3,024 (+68%)	3	44 N1-3, 16 T1-3, 6 T4-19
	MILP 3	3,953 (+68%)	3,388 (+88%)	3	54 N1-3, 20 T1-3, 3 N4-9
Covar (f)	Naive	3,493	3,795	9	23 N2
	MILP 4	4,298 (+23%)	4,727 (+25%)	9	43 N1-3, 18 T1-3, 3 N6-10, 4 T6-20
	MILP 3	4,558 (+30%)	5,196 (+37%)	9	46 N1-3, 24 T1-3, 2 N5-6, 6 T4-13
Gemver	Naive	3,098	2,903	18	30 N2
	MILP 4	4,076 (+32%)	3,990 (+37%)	18	60 N1-3, 7 T1-3, 8 N5-12, 5 T4-10
	MILP 3	4,066 (+31%)	4,353 (+50%)	18	58 N1-3, 29 T1-3, 3 T4-7, 2 N5-7
CDiv	Naive	14,461	14,081	18	6 N2
	MILP 4	15,197 (+5%)	14,780 (+5%)	18	11 N1, 6 T1-2, 8 T11-26, 8 N13-26
	MILP 3	15,164 (+5%)	14,946 (+6%)	18	12 N1, 8 T1, 16 T6-16

The types of instantiated buffers are shown under *Buffers*, where *N* denotes that a buffer is nontransparent and *T* denotes that a buffer is transparent (e.g., 3 N2-4 indicates the usage of three nontransparent buffers with two to four slots).

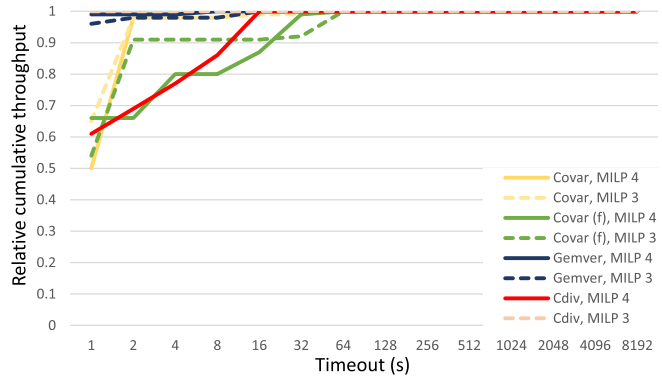
runtime reported in Table 2, it is still nonnegligible in certain cases (e.g., *Gemver* requires at least 30 minutes of MILP runtime to find a good solution).

We investigate the effectiveness of the heuristic from Section 5 to reduce the MILP runtime. We organize the CFDFCs into independent sets and employ the MILP on each set separately. The results we obtain are plotted in Figure 14(b), which compares the obtained cost function result to the optimal result, exactly as in the previous graph. Contrasting the two graphs indicates that this method successfully lowers the time needed for the MILP to converge.

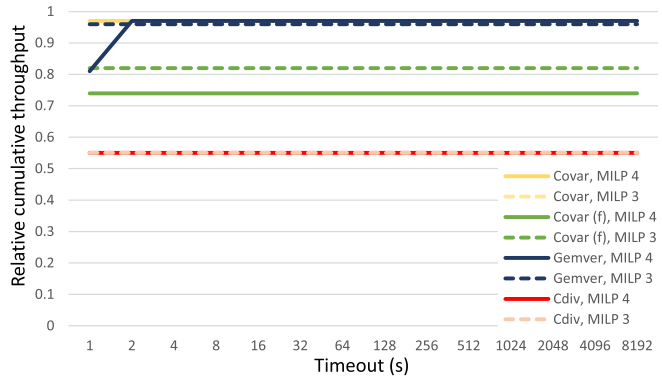
The two versions of the MILP that we have considered so far optimized *all* CFDFCs of the program. Our next experiment is based on the intuition that some CFDFCs do not contribute significantly to the execution time of the application (e.g., the outermost loop of a nested loop)—they can be removed from the cost function without a notable performance penalty. We demonstrate this effect in Figure 14(c), where we compare the cost value of the MILP that optimizes the throughput



(a) Full MILP (i.e., single MILP for entire application).



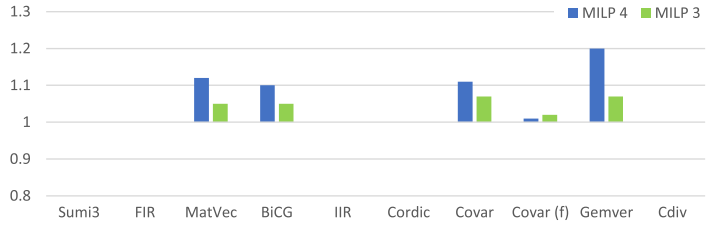
(b) MILP in CFDFC sets.



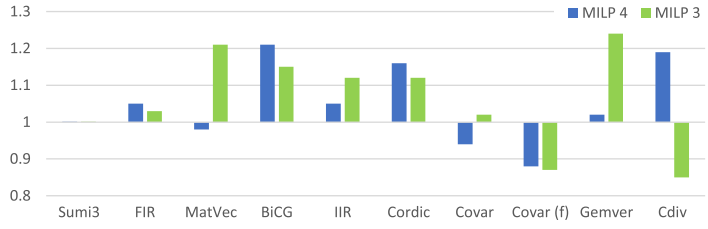
(c) MILP in CFDFC sets, single CFDFC per set.

Fig. 14. Runtime comparison of the full MILP with the MILP applied on individual CFDFC sets, described in Section 5.

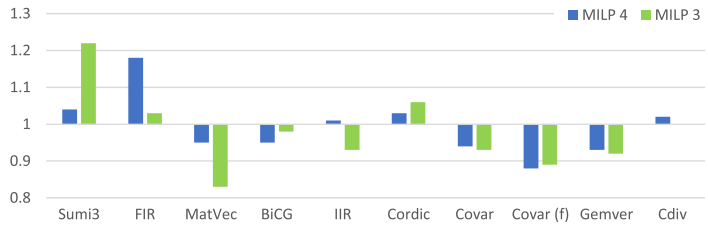
of a *single*, most important CFDFC *per set*, with the optimal MILP cost value, as in the previous graphs. This MILP converges rapidly and, in most cases, obtains a near-optimal value, as the removed CFDFCs contributed to the cost function with a negligible weight factor. However, some applications such as *CDiv* suffer from this simplification: this application has two CFDFCs with similar contributions (i.e., 55% and 45%) to execution time; optimizing the throughput of only one CFDFC lowers the obtainable cost function value and, consequently, application performance.



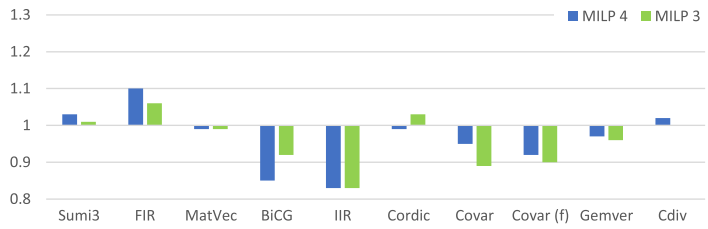
(a) Cycle count ratio (MILP in sets/full MILP).



(b) Execution time ratio (MILP in sets/full MILP).



(c) LUT count ratio (MILP in sets/full MILP).



(d) FF count ratio (MILP in sets/full MILP).

Fig. 15. Comparison of solutions obtained by applying the MILP on individual CFDFC sets with the optimal MILP solutions (i.e., solutions obtained by employing the MILP on the entire circuits).

The results of our runtime analysis can therefore be summarized as follows: (1) it seems possible to rely on timeouts to find good solutions in reasonable runtime, (2) the heuristic from Section 5 helps in further reducing the MILP runtime, and (3) not all CFDFCs play an important role in achievable application performance. It is possible to simplify the MILP to account for this fact and to further reduce its runtime at a negligible penalty.

6.5 Comparison of MILP Solutions

To complement our runtime analysis from the previous section, we evaluate the quality of solutions obtained in the following manner: (1) we choose a timeout of 1 minute to terminate the MILP, (2) we split the CFDFCs into sets to employ the heuristic from Section 5, and (3) in the cost function of

Table 4. Irregular Benchmark Characteristics: Their Set and CFDFC Count, the Main Property of the Loops That They Contain, and the Total Number of Dataflow Units and channels in the Corresponding Dataflow Circuit (Prior to Buffer Insertion)

Benchmark	Sets	CFDFCs	Property	Units	Channels
Histogram	1	1	Irregular memory accesses	51	70
Matrix Power	1	2	Irregular memory accesses	81	110
If loop add	1	1	Irregular control flow	46	64
If loop mul	1	1	Irregular control flow	46	64

each set, we include all CFDFCs, starting from the one with the highest weight, until there is at least an order of magnitude difference in the cost term weight between the last one included and the first one not included. This ensures that the most relevant CFDFCs of each set are optimized (e.g., the innermost loops of our benchmarks; in *CDiv*, this approach includes the throughput optimization of both the if and the else branch).

Figure 15 shows the cycle count, total execution time, and resource consumption of the solutions obtained in such a manner, relative to the optimal MILP solutions from Tables 2 and 3. In applications that have a single CFDFC per set, the obtained cycle count is equal to the optimal because our heuristic covers the entire application; in others (i.e., applications with nested loops), the count slightly increases because the throughput of the outer loops is not optimized. The total execution time varies due to the changes in obtained frequency (largely caused by FPGA place-and-route, as discussed earlier). In most cases, these solutions require fewer resources than the optimal MILP solutions—as the throughputs of certain loops are not optimized, fewer FIFOs are instantiated. All of these variabilities are expected and in an acceptable range for the significant MILP runtime reduction that this heuristic approach offers.

In summary, to easily exploit different runtime and solution tradeoffs, Dynamatic currently provides the following optimization options: (1) solving a single and complete MILP on the entire dataflow graph, (2) decoupling the graph into independent sets where the MILP is applied separately, and (3) decoupling the graph while optimizing only the most relevant loops (as described in this section). The default compiler option is (3), as it achieves near-optimal results at an extremely affordable runtime; the user can optionally trade off runtime and result quality by enabling one of the other optimization options using an intuitive directive-based environment. The appropriate optimizations and analysis (e.g., decoupling the graph and analyzing the performance impact of each loop) are then performed automatically by the compiler, without requiring any user modifications.

6.6 Variable Latency, II, and If-Conversion

In this section, we investigate the effectiveness of our method to model units with variable latency and II, as well as if-conversion. The benchmarks we evaluate, whose properties are summarized in Table 4, exhibit data-dependent and variable behavior: *Histogram* and *Matrix Power* have memory access patterns that cannot be determined at compile time and hence require an LSQ at the memory interface; the LSQ exhibits variable latency and II depending on the runtime-determined data dependencies. *If loop add* and *If loop mul* have a potential dependency across loop iterations; the data to send to the following iteration is selected using a Select unit based on a data-dependent condition, determined during program execution. These kernels are typical examples of situations where dataflow circuits excel in contrast to statically scheduled HLS circuits [23].

We follow the rules from Sections 4.2, 4.3, and 4.4 to model the behavior of these kernels and compare them to the same kernels where buffers are placed naively. Our results are shown in Table 5. As the average II and execution time in all kernels depend on the input data, we

Table 5. Timing and Resources of Kernels That Contain Computational Units with Variable Latency and II, as Well as If-Conversion

Benchmark	Method	CP (ns)	II = 1/Θ	Execution Time (μs)	Speedup	LUTs	FFs	DSPs	MILP runtime (s)
Histogram	Naive	6.1	2.0–12.0	12.2–73.2	–	16,627	3,529	2	–
	MILP 4	6.3	1.0–12.0	6.4–75.6	**1.9–1.0×**	16,879	3,562	2	0.1
Matrix Power	Naive	6.0	3.5–11.7	8.1–26.7	–	16,870	3,696	5	–
	MILP 4	6.2	2.0–11.5	4.9–27.2	**1.7–1.0×**	16,955	3,744	5	23.2
If loop add	Naive	4.9	12.0–20.0	58.8–98.0	–	903	1,284	4	–
	MILP 4	5.0	1.0–10.0	5.1–50.1	**11.6–2.0×**	960	1,318	4	0.2
If loop mul	Naive	5.0	12.0–16.0	60.0–80.0	–	858	1,091	5	–
	MILP 4	5.2	1.0–6.0	5.3–31.3	**11.4–2.6×**	892	1,127	5	0.2

We compare kernels optimized with our strategy (*MILP 4*) to those with a naive buffer placement (*Naive*). The II and execution time are shown as a range from the best-case to the worst-case behavior, as determined by data dependencies. The rightmost column indicates the MILP runtime.

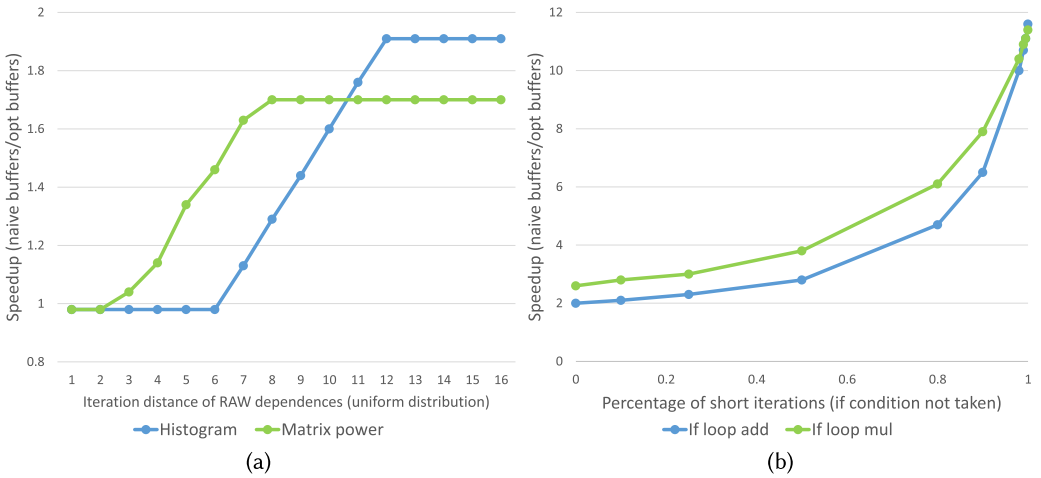


Fig. 16. Speedup of the optimized kernels with respect to the naive kernels for varying data and control dependencies. (a) We change the number of iterations between dependent read and write accesses. (b) We change the percentage of short loop iterations (i.e., the percentage of loop iterations where the long-latency if-condition is not taken).

indicate these values as a range from their minimum to their maximum value. In the kernels with the LSQ, the best-case scenario (i.e., with the smallest II and execution time) corresponds to a situation where there are no RAW dependencies among loop iterations; in the kernels with the Select unit, the same is achieved when there are no loop-carried dependencies. The other extreme is obtained when every pair of loop iterations has a RAW dependency and a loop-carried dependency, respectively. All other possible data points fall in the middle of the shown ranges.

In Figure 16, we show the speedup (i.e., execution time ratio) of the optimized kernels with respect to the naive kernels for a varying number of data or control dependencies (i.e., the data points that are within the execution time range shown in Table 5). In Figure 16(a), we explore the execution time of *Histogram* and *Matrix Power* for a varying number of iterations between dependent read and write accesses (e.g., distance 2 indicates that a load reads a value that is stored into memory two iterations ago). In Figure 16(b), we explore *If loop add* and *If loop mul* while changing the percentage of short loop iterations—that is, the iterations where the if-condition is not taken and there is no loop-carried dependency.

Table 6. Exploration of the Effectiveness of the CP Constraint on a Tree of Combinational Adders

Target CP (ns)	Achieved CP (ns)	II = 1/Θ	Execution Time (μs)	LUTs	FFs
–	9.1	2	15.7	1,578	1,665
8	7.7	1	7.7	1,632	1,742
6	5.7	1	5.7	1,661	1,810
4	4.1	1	4.1	1,853	2,084
3	3.6	2	7.2	2,188	2,696

As Table 5 and Figure 16 illustrate, the naive technique achieves only limited pipelining, whereas the kernels optimized with our approach are able to achieve the highest possible throughput. In these examples, the achievable throughput is variable and dependent on the actual data dependencies: when a large number of dependencies is present, the II increases to honor them; still, the optimized kernels typically achieve higher throughput than the naive kernels, which suffer from backpressure and suboptimal buffering even in those cases. As the number of dependencies decreases (i.e., the iteration distance between RAWs or the percentage of short iterations grows), the kernels become more pipelineable; the optimized designs benefit from the appropriate buffering to achieve better performance and larger speedups. When there are no dependencies, all optimized kernels achieve the best possible II. Although the techniques for ensuring scalability, presented in Section 5 and analyzed in Section 6.4, are generally applicable in benchmarks with irregular and variable behavior, they are not required in the explored benchmarks, as the MILP runtime is already low (as reported in the rightmost column of Table 5).

The resource trends in Table 5 follow the ones we discussed in Section 6.3 and depend on the number and the capacity of the inserted buffers; the overheads of our technique are minor and probably acceptable for the significant performance benefits, which indicates that our technique is critical to truly capture the variable and dynamic behavior of dataflow circuits.

6.7 Effectiveness of the CP Constraint

In this section, we further explore the capabilities of our model to control the critical path. We analyze the effects of the CP constraint on an unrolled accumulator, implemented as a binary tree of adders with 16 inputs; this example gives more room for CP exploration than the benchmarks from the previous section. We present the results in Table 6. The naively obtained CP corresponds to the combinational path through the entire adder tree. Lowering the constraint inserts buffers between different tree stages. Although the achieved CP tracks well the constraint in most cases, the maximum frequency cannot be reached. This effect is most likely due to the control paths that are not included in our timing model and become dominant with tighter CP constraints. Our future work will refine the timing model to account for these effects as well.

6.8 Throughput Comparison with Statically Scheduled HLS

In the previous sections, we have shown that our performance optimization technique effectively lowers the II (i.e., maximizes throughput) and increases dataflow circuit performance in comparison to naively buffered dataflow circuits. In this section, we compare the pipelining capabilities of our approach with that of a standard statically scheduled HLS tool.

We synthesized the benchmarks from Sections 6.3 and 6.6 with Vivado HLS while employing the pipelining directive in the innermost loops. The tool optimizes the synthesized circuits for a minimal II, hence qualitatively matching the strategy of our performance optimizer. We employ

Table 7. Throughput Comparison with Static HLS (i.e., Pipelined Vivado HLS Designs)

Benchmark	Property	Naive II	MILP 4 II	Static II
Sumi3	Regular	10	1	1
Fir	Regular	6	1	1
MatVec	Regular	6	1	1
BiCG	Regular	6	1	1
IIR	Loop-carried dep.	6	5	6
Cordic	Loop-carried dep.	20	20	22
Covar	Regular	2, 4, 4	1, 1, 1	1, 1, 1
Covar (f)	Loop-carried dep.	11, 11, 17	11, 1, 11	10, 1, 10
Gemver	Regular	6, 10, 2, 10	1, 1, 1, 1	1, 1, 1, 1
CDiv	Conditional execution	40, 40	3, 3	1
Histogram	Irregular memory accesses	2–12	1–12	13
Matrix Power	Irregular memory accesses	3.5–11.7	2–11.5	13
If loop add	Irregular control flow	12–20	1–10	10
If loop mul	Irregular control flow	12–16	1–6	7

The Property column summarizes the relevant loop property. Columns Naive II and MILP 4 II repeat the results from Tables 2 and 5; here we compare them to the corresponding static HLS solutions.

the same arithmetic units and RAM memory interfaces as in our dataflow circuits; therefore, we can directly compare the achieved IIs.

Table 7 summarizes the loop IIs of naively optimized dataflow circuits (*Naive* design points from Tables 2 and 5), dataflow circuits optimized with our MILP (*MILP 4* design points from Tables 2 and 5, with a target CP of 4 ns), and Vivado HLS points for the same target CP of 4 ns (indicated as *Static* in this table). In most regular benchmarks, the II of the *MILP 4* designs matches that of the static designs. Benchmarks with a long-latency loop-carried dependence exhibit a minor discrepancy in II due to the differences in the timing models of the two HLS approaches: in *IIR* and *Cordic*, Vivado HLS inserts additional registers on the long-latency critical loops, whereas our approach does not; the exact opposite happens in *Covar (f)*. The II difference in *CDiv* is due to a difference in the HLS compilation process: while Vivado HLS inlines the if-else condition and achieves a single loop body, pipelineable with II=1, Dynamatic keeps the if-else structure of the CFG, as shown in Figure 8; the corresponding dataflow circuit contains restrictive synchronization logic that prevents complete loop pipelining. This effect is due to the dataflow circuit construction strategy of Dynamatic, detailed in prior work [23], and completely orthogonal to our contribution.

Our performance optimization strategy enables us to fully benefit from the flexibility of dynamic scheduling in irregular benchmarks (*Histogram*, *Matrix Power*, *If loop add*, and *If loop mul*); as discussed before, the corresponding dataflow circuits are perfectly pipelined thanks to our strategy and their II varies with the actual data and control decisions. In contrast, static scheduling makes conservative assumptions on data and control dependencies, thus always resulting in a high II and low performance.

We demonstrated in Tables 2 and 5 that our dataflow circuits do not always meet the target CP; in contrast, Vivado HLS designs successfully achieve the target of 4 ns in all explored cases. As suggested before, this issue is not fundamental for our approach; just like in Vivado HLS, our timing models could be further refined to account for backend particularities of FPGA synthesis, placement, and routing. Dataflow circuits exhibit notable resource overheads compared to the corresponding static solutions: in benchmarks that do not require an LSQ at the memory interface (i.e., all benchmarks apart from *Histogram* and *Matrix Power*), we measured an average increase

of 5× in LUT utilization and 2.8× in FF utilization. These values further increase in benchmarks that use an LSQ due to its architectural complexity: *Histogram* requires 16.9× LUTs and 4× FFs, and *Matrix Power* requires 13.1× LUTs and 2.8× FFs of the corresponding static solutions. These observations are orthogonal to our contribution (i.e., providing a systematic methodology to optimize dataflow circuit performance) and consistent with previous findings; we refer the interested reader to prior work [22–24] for a detailed discussion on the area-performance tradeoffs of static and dynamic scheduling.

In summary, all of these results point to the effectiveness of our performance optimization strategy: in regular benchmarks, our pipelining capabilities qualitatively correspond to those of a state-of-the-art HLS tool, whereas in irregular benchmarks, our dataflow circuits achieve flexible high-throughput pipelines that outperform their static counterparts.

7 RELATED WORK

In HLS, timing optimizations are crucial for achieving high-performance circuits. In standard, statically scheduled HLS, such optimizations are typically performed in conjunction with modulo scheduling [5, 36, 44]: the aim is to create pipelines with the best possible loop initiation intervals under the given clock and resource constraints. Dataflow circuits [2, 23] are fundamentally different—their schedules are not predetermined at compile time but devised as the circuit runs. Yet, as in standard HLS, the ultimate goal is to create timing-efficient, high-throughput pipelines—we investigate this objective in this work.

Latency-insensitive protocols [7, 14] have been extensively used to create synchronous and asynchronous dataflow circuits; their timing properties can be analyzed using Petri net theory [3, 4, 34, 35]. Several approaches in asynchronous dataflow design have explored slack matching (i.e., adding pipeline buffers to prevent stalls). Venkataramani and Goldstein [39] present a heuristic to avoid performance bottlenecks by inserting buffers to balance reconverging paths in asynchronous circuits. Najibi and Beerel [30] describe slack matching for asynchronous circuits with conditional computation and communication, where the conditions correspond to different circuit operation modes; they employ a MILP based on Markov chains, a wide class of Petri nets, to balance asynchronous pipelines while reducing the number of slack-matching buffers. In contrast to these works, our model considers retiming and slack matching simultaneously—we target *synchronous* dataflow circuits, so the CP must be optimized in conjunction with the throughput. Furthermore, our method accepts generic control flow schemes that commonly appear in high-level languages (e.g., nested loops) and accounts for typical HLS features (e.g., pipelined computational units and if-convertible control flow).

Standard HLS tools support task-level pipelining (referred to as dataflow optimization [42]), which allows functions and loops to overlap in execution to increase throughput and concurrency. The tasks are connected via channels, implemented as ping-pong buffers or FIFOs, that allow the consumer function or loop to start operation before the producer task has completed. The buffers are typically sized in a conservative manner so that they have the capacity to hold all data exchanged between tasks. Task-level pipelining is usually applicable only to tasks that do not have bypass, feedback, or conditionals between each other. In our work, we explore finer-grain dataflow design (i.e., scheduling individual loop and function datapaths); as we have demonstrated in our results, our approach successfully supports cyclic behavior and conditionals and is able to compute the required FIFO sizes even in those cases. Other HLS efforts also explore coarse-grain dataflow and the related buffer sizing problems. Cheng and Wawrzynek [12] describe sequential programs as networks of processes in which hardware accelerators exchange data via FIFOs. To avoid deadlock, they analyze the static schedule of each accelerator and size the FIFOs accordingly.

Yet, this approach does not always provide a global optimal solution for cases with multiple deadlock-causing cycles. Govindarajan et al. [17] target large-grain, multi-rate actor graphs and present an approach to minimize buffer storage while executing at the optimal computation rate. In contrast to our contribution, this approach does not consider constraining the CP and it does not guarantee the optimality of the buffer placement.

8 CONCLUSION

In this work, we present a performance optimization model for dataflow circuits obtained out of C code. Our MILP model is based on the theory of marked graphs and allows for resource-optimal buffer placement and sizing, with the purpose of maximizing throughput at the desired clock frequency. In addition to the exact model formulation, we propose a computationally efficient heuristic that achieves near-optimal results; its ability to handle large benchmarks makes our approach applicable to real-world and complex workloads. On benchmarks obtained out of C code, we demonstrate the ability of our approach to achieve high-throughput, pipelined dataflow circuits. We show that our approach effectively handles different HLS features such as pipelined computational units, variable-latency memory interfaces, and if-converted code. We believe that optimization techniques such as this one are the key to making dynamic scheduling truly competitive with existing HLS techniques.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Longman.

[2] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. 2005. Dataflow: A complement to superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software.* 177–186.

[3] Dmitry Bufistov, Jordi Cortadella, Mike Kishinevsky, and Sachin Sapatnekar. 2007. A general model for performance optimization of sequential systems. In *Proceedings of the International Conference on Computer-Aided Design.* 362–369.

[4] J. Campos, G. Chiola, J. M. Colom, and M. Silva. 1992. Properties and performance bounds for timed marked graphs. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 39, 5 (May 1992), 386–401.

[5] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications.* 1–8.

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems* 13, 2 (Sept. 2013), Article 24, 27 pages.

[7] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* CAD-20, 9 (Sept. 2001), 1059–1076.

[8] Mario R. Casu and Luca Macchiarulo. 2009. Adaptive latency insensitive protocols and elastic circuits with early evaluation: A comparative analysis. *Electronic Notes in Theoretical Computer Science* 245 (Aug. 2009), 35–50.

[9] GitHub. n.d. CBC Mixed-Integer Linear Programming Solver. Retrieved September 20, 2021 from https://github.com/coin-or/Cbc.

[10] Satrajit Chatterjee, Michael Kishinevsky, and Umit Y. Ogras. 2012. xMAS: Quick formal modeling of communication fabrics to enable verification. *IEEE Design & Test of Computers* 29, 3 (June 2012), 80–88.

[11] Shin-Kai Chen, Chih-Wei Liu, Tsung-Yi Wu, and An-Chi Tsai. 2013. Design and implementation of high-speed and energy-efficient variable-latency speculating booth multiplier (VLSBM). *IEEE Transactions on Circuits and Systems* 60, 10 (Sept. 2013), 2631–2643.

[12] Shaoyi Cheng and John Wawrzynek. 2016. Synthesis of statically analyzable accelerator networks from sequential programs. In *Proceedings of the International Conference on Computer-Aided Design.* 126–133.

[13] Jordi Cortadella and Mike Kishinevsky. 2007. Synchronous elastic circuits with early evaluation and token counterflow. In *Proceedings of the 44th Design Automation Conference.* 416–419.

[14] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference.* 657–662.

[15] Doug Edwards and Andrew Bardsley. 2002. Balsa: An asynchronous hardware synthesis language. *Computer Journal* 45, 1 (Jan. 2002), 12–18.
[16] Stephen A. Edwards, Richard Townsend, and Martha A. Kim. 2017. Compositional dataflow circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 175–184. https://doi.org/10.1145/3127041.3127055
[17] Ramaswamy Govindarajan, Guang R. Gao, and Palash Desai. 2002. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 31, 3 (July 2002), 207–229.
[18] Mark R. Greenstreet and Kenneth Steiglitz. 1990. Bubbles can make self-timed pipelines fast. *Journal of VLSI Signal Processing* 2, 3 (Nov. 1990), 139–148.
[19] Greg Hoover and Forrest Brewer. 2008. Synthesizing synchronous elastic flow networks. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 306–311.
[20] Hans M. Jacobson, Prabhakar N. Kudva, Pradip Bose, Peter W. Cook, Stanley E. Schuster, Eric G. Mercer, and Chris J. Myers. 2002. Synchronous interlocked pipelines. In *Proceedings of the 8th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 3–12.
[21] Donald B. Johnson. 1975. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing* 4, 1 (March 1975), 77–84.
[22] Lana Josipović, Philip Brisk, and Paolo Ienne. 2017. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems* 16, 5s (Sept. 2017), Article 125, 19 pages. https://doi.org/10.1145/3126525
[23] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 127–136.
[24] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative dataflow circuits. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 162–171.
[25] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer placement and sizing for high-performance dataflow circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 186–196.
[26] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel programming for FPGAs. *ArXiv e-prints* arXiv:1805.03648 (May 2018).
[27] Charles E. Leiserson and James B. Saxe. 1991. Retiming synchronous circuitry. *Algorithmica* 6, 1–6 (June 1991), 5–35.
[28] Rajit Manohar and Alain J. Martin. 1998. Slack elasticity in concurrent computing. In *Proceedings of the 4th International Conference on the Mathematics of Program Construction*. 272–285.
[29] Tadao Murata. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77, 4 (April 1989), 541–580.
[30] Mehrdad Najibi and Peter A. Beerel. 2013. Slack matching mode-based asynchronous circuits for average-case performance. In *Proceedings of the 32nd International Conference on Computer-Aided Design*. 219–225.
[31] Mauro Olivieri. 2001. Design of synchronous and asynchronous variable-latency pipelined multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9, 2 (April 2001), 365–376.
[32] Louis-Noël Pouchet. 2012. Polybench: The Polyhedral Benchmark Suite. Retrieved September 20, 2021 from http://www.cs.ucla.edu/pouchet/software/polybench.
[33] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press.
[34] C. V. Ramamoorthy and Gary S. Ho. 1980. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering* 6, 5 (Sept. 1980), 440–449.
[35] C. Ramchandani. 1974. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Technical Report Project MAC Technical Report 120. Massachusetts Institute of Technology, Cambridge, MA.
[36] B. Ramakrishna Rau. 1996. Iterative modulo scheduling. *International Journal of Parallel Programming* 24, 1 (Feb. 1996), 3–64.
[37] Jens Sparsø. 2009. Current trends in high-level synthesis of asynchronous circuits. In *Proceedings of the 16th IEEE International Conference on Electronics, Circuits, and Systems*. 347–350.
[38] Linda Torczon and Keith Cooper. 2011. *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.
[39] Girish Venkataramani and Seth C. Goldstein. 2006. Leveraging protocol knowledge in slack matching. In *Proceedings of the 25th International Conference on Computer-Aided Design*. 724–729.
[40] Ajay K. Verma, Philip Brisk, and Paolo Ienne. 2008. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*. 1250–1255.
[41] Muralidaran Vijayaraghavan and Arvind. 2009. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign*. 171–180.

[42] Xilinx Inc. 2018. Vivado Design Suite User Guide: High-Level Synthesis. Xilinx Inc. Retrieved September 20, 2021 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.
[43] Xilinx Inc. 2018. Vivado High-Level Synthesis. Xilinx Inc. Retrieved September 20, 2021 from http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[44] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *Proceedings of the 32nd International Conference on Computer-Aided Design*. 211–218.

Received July 2020; revised May 2021; accepted July 2021