Buffer Placement and Sizing for High-Performance Dataflow Circuits

Lana Josipović[1], Shabnam Sheikhha[1], Andrea Guerrieri[1], Paolo Ienne[1], and Jordi Cortadella[2]

[1] École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

ABSTRACT

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) and unpredictable memory dependencies. 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. We demonstrate the performance benefits of our approach on a set of dataflow circuits obtained from imperative code.

ACM Reference Format:
Lana Josipović[1], Shabnam Sheikhha[1], Andrea Guerrieri[1], Paolo Ienne[1], and Jordi Cortadella[2]. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375314

1 INTRODUCTION

High-level synthesis (HLS) tools which rely on static scheduling face a fundamental issue when handling irregular applications: they force worst-case assumptions on memory and control dependencies which prevent them from creating high-throughput pipelines. In contrast, dataflow circuits implement dynamic scheduling and can resolve such dependencies as the circuit runs, thus achieving better performance. Some HLS approaches build dataflow circuits out of imperative code [2, 15], but they still rely on crude heuristics and manual tuning to optimize performance.

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 which 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 which achieves maximum design parallelism at the desired clock frequency and with minimal resource cost.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02.
https://doi.org/10.1145/3373087.3375314

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 [6, 8] are a natural method to create synchronous dataflow circuits, capable of taking 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 [9, 13, 15, 27]: 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 [7, 12]: (1) A *merge* propagates a token received from any of the predecessor BBs into its BB body. (2) A *branch* propagates a token from its input to any of the successor BBs based on a condition.

Figure 1a shows a dataflow circuit which 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 basic block 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 to the merges to trigger 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* which 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 of a dataflow circuit must contain at least one buffer, as given in Figure 1a. Yet, in contrast to standard registers, buffers can be placed on *any* channel of the dataflow circuit—this insertion will

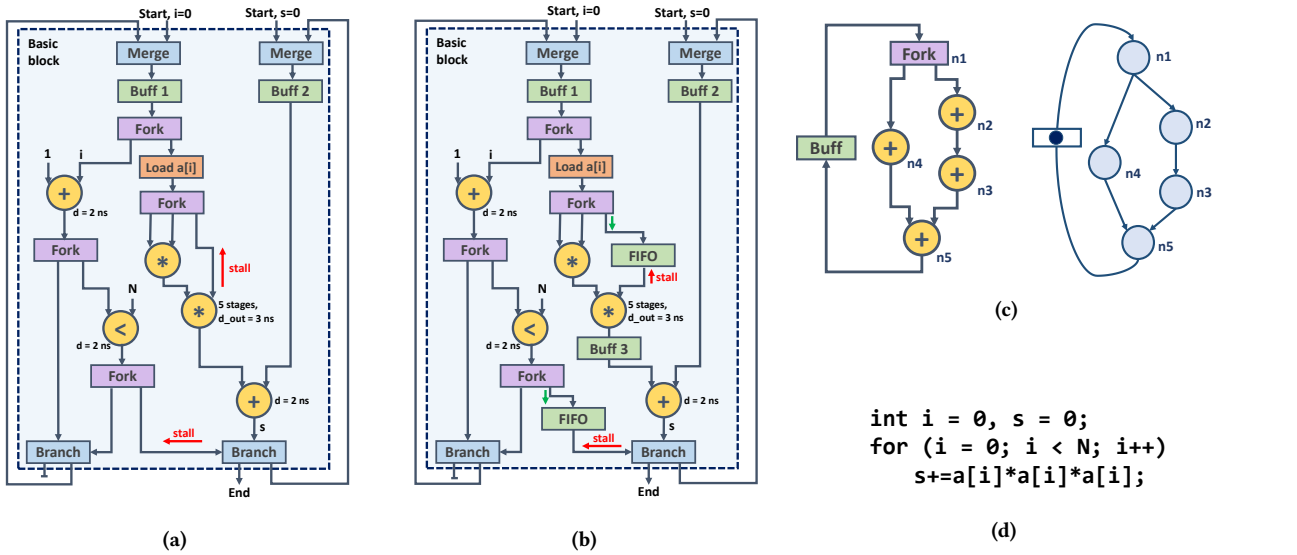


Figure 1: A functionally correct, but non-optimized dataflow circuit (Figure 1a) implementing the code from Figure 1d contains buffers (i.e., registers) placed to break all combinational loops. The optimized circuit (Figure 1b) has buffers placed strategically to control the critical paths. Moreover, the FIFOs in the paths with higher latency mitigate backpressure and allow achieving the ideal loop initiation interval. A choice-free dataflow circuit (Figure 1c) and its representation as a marked graph. Circuits obtained out of high-level code (such as the ones in Figure 1a and 1b) contain choices (i.e., control flow decisions through merges and branches) and cannot be represented as marked graphs.

not compromise the functionality of the circuit due to its latency-insensitivity [3, 15], but may impact its timing and throughput.

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

The circuit in Figure 1a 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: (1) *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 control 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. (2) *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 1a, the token carrying the array value *a* is forked into two pipelined multipliers, but the lower multiplier cannot accept the token until the upper multiplier is done computing (i.e., after 5 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 5-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, therefore effectively lowering the initiation interval of the loop.

Figure 1b shows a possible 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* [19] and increases effective parallelism, as accumulating data in the FIFOs allows to trigger the faster paths at a higher rate [15] and achieves the ideal loop initiation interval of 1.

2.3 Marked Graphs

Marked graphs are a class of Petri nets [20] which represent concurrent behavior, but never have any *choices*, i.e., conditional execution. Figure 1c shows an example of a *choice-free* dataflow circuit and its representation in the form of a marked graph. The buffer on the back edge of the circuit contains a token which 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, 24, 25]; some address explicitly optimal buffer placement [3].

It is immediately clear that circuits such as the one in Figure 1a do not exhibit the choice-free behavior of marked graphs, as each control flow edge between BBs represents a choice: the merges in Figure 1a 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.

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. On the other hand, 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. In this work, we combine the knowledge from marked graph theory with dataflow circuits which implement choices in order to optimize their performance: our work is based on the observation that choice-free subgraphs with the properties

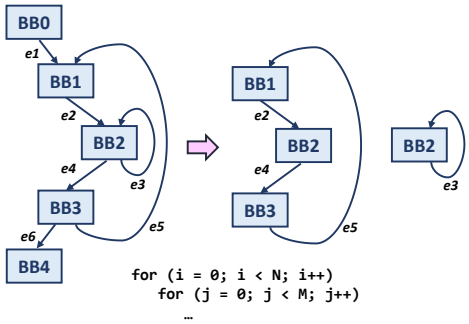


Figure 2: Extracting CFG cycles. The leftmost graph is a control flow graph of a nested loop with two cycles. We optimize choice-free dataflow circuits (CFDFCs) which correspond to these cycles.

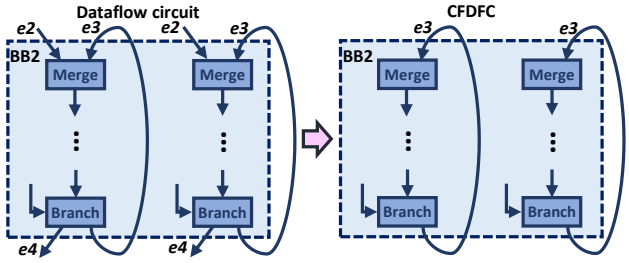


Figure 3: Obtaining a choice-free dataflow circuit (CFDFC) from a dataflow circuit. A CFDFC contains all dataflow units and channels which belong to any of the BBs or edges of the extracted CFG cycle (in this example, BB2 and edge e_3 from Figure 2). Every merge and branch in a CFDFC have a single input and output edge, respectively.

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 pipelined computational units which HLS techniques regularly employ. 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 all subgraphs. Finally, we discuss scalability and present some techniques to limit runtime.

3.1 Extracting Choice-Free Dataflow Circuits

In this section, we describe our methodology for extracting the most significant *choice-free dataflow circuit* (CFDFC) from a dataflow circuit. We define a CFDFC as a dataflow circuit obtained from a cycle of the control-flow graph (CFG). A CFDFC can, therefore, be represented as a marked graph which is (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). Figure 2 shows a control flow graph of a nested loop: it contains two cycles which, internally, correspond to two CFDFCs.

The performance optimization which 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 (variable, binary). Indicates whether the control flow edge e belongs to the selected CFG cycle.
- S_{BB} (variable, binary). Indicates whether a basic block belongs to the selected CFG cycle.
- N (variable). Total number of times the CFG cycle executes.

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:

$$N \leq S_e \cdot \mathbf{N}_e + (1 - S_e) \cdot \mathbf{N}_{\max}, \forall e \in CFG. \tag{1}$$

Here, $\mathbf{N}_{\max}$ represents the upper bound on the number of executions (i.e., the execution count of the most frequently executed edge of the CFG). It ensures that N is not constrained by the execution frequencies of edges which do not belong to the loop.

If a BB is a part of the selected cyle, exactly one of its input and one of its output edges belongs to the cycle as well:

$$S_{BB} = \sum_{e \in In(BB)} S_e = \sum_{e \in Out(BB)} S_e, \forall e \in CFG. \tag{2}$$

Here, $In(BB)$ and $Out(BB)$ denote the sets of input and output edges of BB, 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:

$$\sum_e S_e = 1, \forall e \in Back(CFG). \tag{3}$$

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 which 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:

$$\max: \quad \sum_e N \cdot S_e. \tag{4}$$

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 3 details the extraction of the most significant CFDFC of the program in Figure 2. The ILP will identify the self-loop of BB2 as the cycle with the highest execution frequency (i.e., the ILP

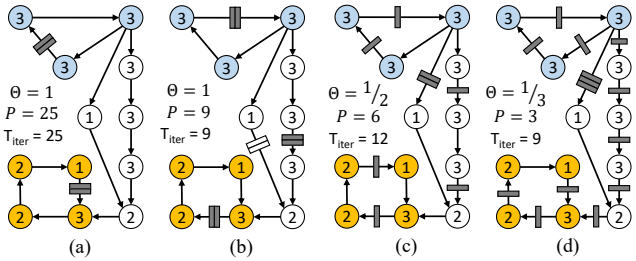


Figure 4: Performance optimization of a choice-free dataflow circuit. Grey buffers are used for breaking combinational paths. The white buffer is transparent and used for throughput optimization.

result will be $S_{BB2} = 1$, $S_{e3} = 1$, and $S = 0$ for all other BBs and edges). Therefore, for each merge in BB2, we keep only the input channel which 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 the 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 3.5.

3.2 Optimizing Choice-Free Circuits

The mathematical model presented in this paper is based on the theory for performance analysis of concurrent systems inherited from timed Petri nets [3, 4, 24, 25]. 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.

A buffer can hold a token or a bubble—each time a token moves forward, a bubble moves in the opposite direction, similarly to electrons and holes in semiconductors [11]. Every cycle of our circuit will always contain *at most one token* [15], whereas bubbles can be freely allocated by adding buffers without affecting functionality [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 1-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 two-slot nontransparent buffer is what we have indicated as *Buff* in Figure 1a—it is sometimes referred to as elastic buffer [15]. 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 minimum cycle ratio [25]. As every cycle contains at least a single nontransparent buffer, the throughput equals at most one (i.e., $\Theta \leq 1$).

Fig. 4 demonstrates the exploration space for performance optimization of a choice-free dataflow circuit. 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. Fig. 4(a) shows a solution with maximum throughput ($\Theta = 1$) by only putting 2-slot nontransparent buffers on the cycles. The cycle period $\mathbf{P}$ 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 (Fig. 4(b)). To prevent the topmost loop to stall due to backpressure, 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 which acts as a FIFO). Constraining the system to work with $\mathbf{P} \leq 8$ requires the addition of nontransparent buffers on the cycles, thus degrading the throughput. Fig. 4(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 Fig. 4(d) with $\Theta = 1/3$ and $\mathbf{P} = 3$, resulting in 9 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 which 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—we here adapt it to generic dataflow graphs. In Section 3.4, we extend the model to support sequential functional units, and in Section 3.5, we generalize it to multiple CFDFCs.

We class constants and variables of the MILP model into three groups: input constants, internal variables, and output variables.

Input constants of the model.

- $\mathbf{P}$ (integer). Target clock period of the circuit.
- $\mathbf{P}_{\max}$ (integer). Upper bound on the clock period 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 $x \xrightarrow{c} y$).

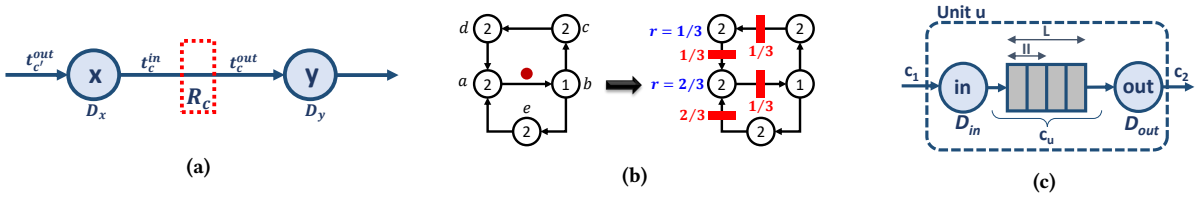


Figure 5: The MILP model for performance optimization. Figure 5a shows the formulation of the path constraints. Figure 5b demonstrates an optimization with throughput and path constraints for $\mathbf{P} \leq 3$. Figure 5c shows an abstract model of a sequential (pipelined) unit.

- 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 clock period. They are therefore applied to the *complete dataflow graph*. For every channel c:

$$t_c^{out} \geq t_c^{in} - \mathbf{P}_{\max} \cdot R_c, \tag{5}$$

with $t_c^{out} \geq 0$. This constraint, depicted in Figure 5a, 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 a unit u with a pair of input/output channels $x \xrightarrow{c_1} u \xrightarrow{c_2} y$:

$$\mathbf{P} \geq t_{c_2}^{in} \geq t_{c_1}^{out} + \mathbf{D}_u. \tag{6}$$

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 cycle. 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 $u \xrightarrow{c} v$:

$$\overset{\bullet}{\Theta}_c = \mathbf{B}_c + r_v - r_u \tag{7}$$

$$\Theta \leq \overset{\bullet}{\Theta}_c \,/\, R_c. \tag{8}$$

The first constraint is analogous to the equations of classical retiming [18]; in this case, the variables are real instead of integers (i.e., fluid retiming) and $\overset{\bullet}{\Theta}_c$ 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 $\Theta \leq 1$:

$$\Theta \leq \overset{\bullet}{\Theta}_c - R_c + 1. \tag{9}$$

Figure 5b 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 $\overset{\bullet}{\Theta}_c$. They indicate that every channel of the upper loop with a buffer will contain a token every 1 out of 3 clock cycles, whereas the buffer in the bottom left channel will contain a token 2 out of 3 clock cycles. The values next to the units represent the retiming values r which 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. Buffer sizing is essential for avoiding backpressure. It corresponds to allocating bubbles (i.e., adding empty buffer slots), which do not affect circuit functionality. The constraint for bubble throughput is dual to that of token throughput:

$$\overset{\circ}{\Theta}_c \geq \Theta \cdot R_c. \tag{10}$$

The average occupancy of tokens and bubbles will determine the number of buffer slots at every channel:

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

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

$$\max: \quad \Theta - \lambda \cdot \sum_c N_c, \tag{12}$$

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 clock period or the buffer area cost under a throughput constraint [3]).

3.4 Modeling Pipelined Units

The model that we have presented so far (as well as the model by Bufistov et al. [3] which our work is based on) only accounts for combinational nodes connected via channels. 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 parameters: latency $\mathbf{L}_u$ (for variable-latency units, either the average or the worst-case latency can be considered) and initiation interval $\mathbf{II}_u$. Figure 5c depicts our model.

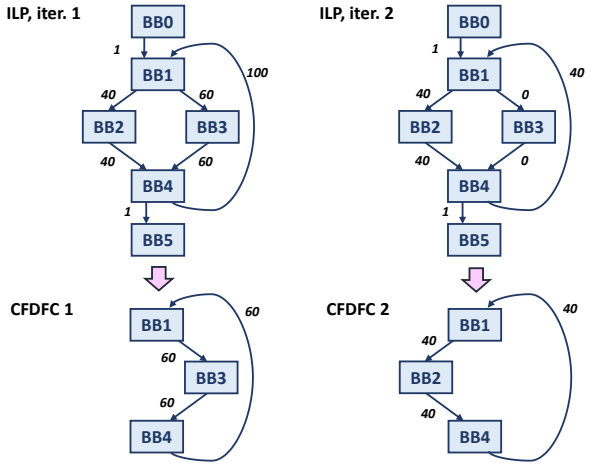


Figure 6: Extracting multiple CFDFCs. The ILP from Section 3.1 can be iteratively applied while updating the execution frequencies of the CFG edges to extract one CFDFC after another. In the figure, the first extracted cycle (and the CFDFC which it represents) executes 60 times. After subtracting this value from the execution count of each corresponding CFG edge, we extract the next cycle of 40 iterations.

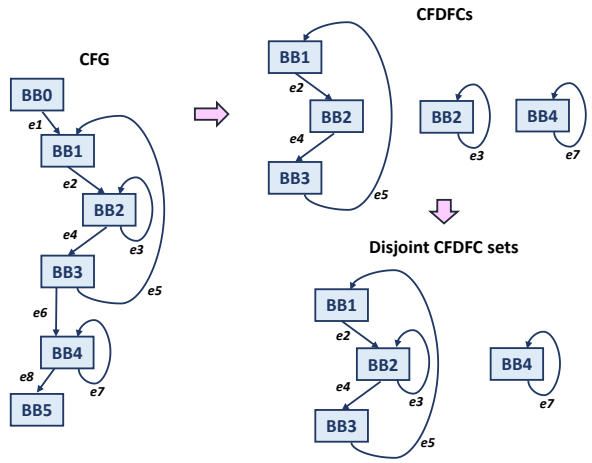


Figure 7: Splitting the circuit into disjoint CFDFC sets to ensure MILP scalability. This circuit represented by the CFG in the figure consist of three CFDFCs which can be grouped into two disjoint sets. Applying the MILP on each set separately reduces the size of each MILP and decreases overall runtime.

The unit is modeled as a channel c_u. It contains a nontransparent buffer with $\mathbf{L}_u$ slots and has one combinational unit at the input (*in*, with a delay $\mathbf{D}_{in}$) and another at the output (*out*, with a delay $\mathbf{D}_{out}$). The delays $\mathbf{D}_{in}$ and $\mathbf{D}_{out}$ participate in the path constraints of the MILP (i.e., Equations 5 and 6), like any other unit. 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:

$$\dot{\Theta}_u = r_{out} - r_{in}, \tag{13}$$

$$\Theta \cdot \mathbf{L}_u \leq \dot{\Theta}_u \leq \mathbf{L}_u/\mathbf{II}_u, \tag{14}$$

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

3.5 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 next cycle and its corresponding CFDFC based on its share in the runtime of the program. We illustrate this approach in Figure 6.

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 [14], 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 4, 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 7 to 11). The cost function to maximize system throughput considers a weighted sum of the individual throughputs Θ of all extracted CFDFCs:

$$\max : \sum_i w_i \cdot \Theta_i - \lambda \cdot \sum_c N_c, \tag{15}$$

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).

3.6 Scalability

In this section, we discuss the runtime complexity of the MILP model proposed 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 4. 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 5 and 6) extend on the *entire dataflow graph*—they need to

Benchmark	Sets	CFDFCs	Property
Sumi3	1	1	regular
Fir	1	1	regular
MatVec	1	2	regular
BiCG	1	2	regular
IIR	1	1	loop-carried dep.
Cordic	1	1	loop-carried dep.
Covar	3	7	regular
Covar (f)	3	7	loop-carried dep.
Gemver	4	7	regular
CDiv	1	2	conditional execution

Table 1: Benchmark characteristics, i.e., their set and CFDFC count as well as main property of the loops which they contain.

ensure that the circuit as a whole meets the target period. The MILP size and runtime is 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., CFDFCs obtained from parts 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. Afterwards, we need to ensure that the *complete circuit* satisfies the period constraint. Hence, we apply the path constraints (i.e., Equations 5 and 6) on the channels and units which were not covered by any of the CFDFC sets. The buffer placement 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:

$$\min : \sum_c R_c, \tag{16}$$

Figure 7 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 clock period and achieving the same CFDFC throughput as the global MILP solution. We will show the effectiveness of this approach in Section 4.

4 EVALUATION

In this section, we demonstrate the ability of our optimization technique to maximize throughput under a given clock period constraint. We compare our optimization approach with a naive buffer placement strategy, discuss the runtime of our algorithm and methods to improve it, and investigate the effectiveness of the period constraint. Recent papers have directly compared dataflow circuits to commercial statically-scheduled HLS results [15, 16] and this comparison is, therefore, outside of the scope of this paper.

4.1 Methodology

The synchronous dataflow circuits we analyze are obtained from C code using an existing methodology [15]. We profile the intermediate representation of the code 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. We use this information 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 which satisfy the target clock period and maximize the loop throughputs—we employ the cost function from Equation 15. 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 we use Vivado to measure the delays of the units. We obtain the clock period (CP) and the resource usage after placement and routing. We use the CBC mixed-integer programming solver [10] and measure its runtime on an Intel® Core™ i7-8550U CPU (i.e., a standard consumer laptop) at 1.80 GHz.

4.2 Benchmarks

We explore various kernels obtained from literature [17, 23] and the PolyBench suite [22]. 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 1d. *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 which 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 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 4); we assume a data distribution where the if-condition is taken in 55% of the total loop iterations.

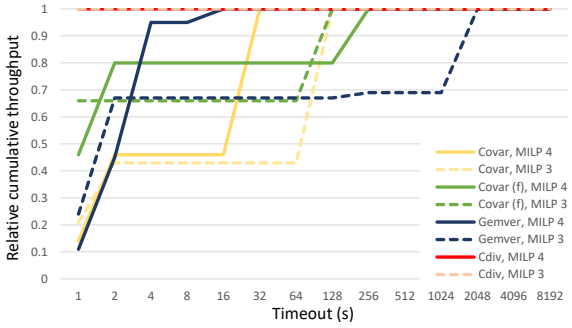
Bench-mark	Method	CP (ns)	II= 1/Θ	Execution Time (μs)	Speedup	LUTs	FFs	DSPs	Buffers	Run-time (s)
	Naive	4.3	10	43.0	–	287	331	6	3 N2	–
Sumi3	MILP 4	4	1	4.1	**10.6×**	402 (+40%)	403 (+22%)	6	8 N1-2, T4, 2 T9	0.1
	MILP 3	3.5	2	7.1	**6.1×**	413 (+44%)	522 (+58%)	6	10 N1-2, 2 T1-2, 2 T5	1.2
	Naive	4.3	6	25.8	–	380	384	3	3 N2	–
FIR	MILP 4	3.9	1	4.0	**6.5×**	428 (+13%)	504 (+31%)	3	5 N1-2, 2 T6-7	0.1
	MILP 3	3.5	2	7.0	**3.7×**	628 (+65%)	688 (+79%)	3	7 N1-2, 2 T4-5	0.8
	Naive	5.9	6	31.9	–	626	517	3	6 N2	–
MatVec	MILP 4	4.8	1	4.5	**7.1×**	808 (+29%)	724 (+40%)	3	11 N1-2, 5 T3-8	15.7
	MILP 3	3.9	2	7.3	**4.4×**	947 (+51%)	849 (+64%)	3	16 N1-2, N4, 6 T1-3	25.0
	Naive	6.0	6	32.4	–	831	758	6	6 N2	–
BiCG	MILP 4	5.9	1	6.4	**5.0×**	1144 (+38%)	1157 (+53%)	6	16 N1-3, 7 T1-3, 4 T5-7	1328.4
	MILP 3	4.1	2	7.7	**4.2×**	1140 (+37%)	1255 (+66%)	6	14 N1-2, 2 N4-5, 8 T1-3	2195.5
	Naive	5.9	6	35.4	–	648	663	6	5 N2	–
IIR	MILP 4	3.9	5	19.5	**1.8×**	745 (+15%)	1096 (+65%)	6	10 N1-2, 6 T1-2	1.1
	MILP 3	3.4	5	17.0	**2.1×**	772 (+19%)	1094 (+65%)	6	12 N1-2, 5 T1-2	20.1
	Naive	5.8	20	116.1	–	1950	2754	24	7 N2	–
Cordic	MILP 4	4.5	20	87.8	**1.3×**	2075 (+6%)	3086 (+12%)	24	16 N1-2, 9 T1-2	8.1
	MILP 3	5	20	97.6	**1.2×**	2145 (+10%)	3016 (+10%)	24	17 N1-2, 9 T1-2	8.1
	Naive	6.8	2, 4, 4	698.5	–	2347	1801	3	23 N2	–
Covar	MILP 4	6.5	1, 1, 1	197.5	**3.5×**	3882 (+65%)	3024 (+68%)	3	44 N1-3, 16 T1-3, 6 T4-19	3600
	MILP 3	5.6	2, 2, 2	182.9	**3.8×**	3953 (+68%)	3388 (+88%)	3	54 N1-3, 20 T1-3, 3 N4-9	3600
	Naive	7.1	11, 11, 17	1833.6	–	3493	3795	9	23 N2	–
Covar (f)	MILP 4	7.2	11, 1, 11	1057.2	**1.7×**	4298 (+23%)	4727 (+25%)	9	43 N1-3, 18 T1-3, 3 N6-10, 4 T6-20	3600
	MILP 3	5.4	11, 2, 11	865.7	**2.1×**	4558 (+30%)	5196 (+37%)	9	46 N1-3, 24 T1-3, 2 N5-6, 6 T4-13	3600
	Naive	7.7	6, 10, 2, 10	180.7	–	3098	2903	18	30 N2	–
Gemver	MILP 4	7.4	1, 1, 1, 1	23.5	**7.7×**	4076 (+32%)	3990 (+37%)	18	60 N1-3, 7 T1-3, 8 N5-12, 5 T4-10	3600
	MILP 3	5.5	2, 2, 2, 2	31.3	**5.8×**	4066 (+31%)	4353 (+50%)	18	58 N1-3, 29 T1-3, 3 T4-7, 2 N5-7	3600
	Naive	10.5	40, 40	787.9	–	14461	14081	18	6 N2	–
CDiv	MILP 4	7.5	3, 3	23.3	**33.8×**	15197 (+5%)	14780 (+5%)	18	11 N1, 6 T1-2, 8 T11-26, 8 N13-26	25.4
	MILP 3	7.2	5, 5	36.8	**21.4×**	15164 (+5%)	14946 (+6%)	18	12 N1, 8 T1, 16 T6-16	153.9

Table 2: Timing and resources of dataflow circuits optimized with our strategy (*MILP 4* and *MILP 3*, with a target CP of 4ns and 3 ns, respectively) compared to a naive buffer placement (*Naive*). Under *II*, we indicate the initiation intervals of the innermost loops. The types of instantiated buffers are shown under *Buffers* (e.g., 3 N2-4 indicates the usage of 3 nontransparent buffers with two to four slots). 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 3600 indicates a timeout of 1 hour).

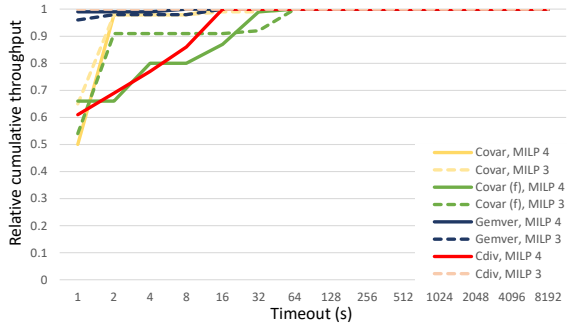
4.3 Comparison with Naive Buffer Placement

We demonstrate the performance superiority of our optimized circuits over equivalent designs with buffers placed naively, based on an existing heuristic [15] which cuts every combinational cycle with a single buffer and does not place any FIFOs into the designs.

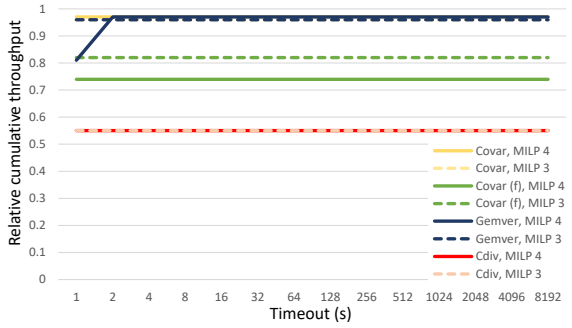
Table 2 summarizes our results. The circuits produced using the naive strategy (i.e., *Naive*) qualitatively correspond to the circuit in Figure 1a: they contain the minimal number of buffers to create functional circuits (i.e., circuits with no combinational loops), but there is no way to control the 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 is due to the additional buffers which our technique employs, 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).



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



(b) MILP in CFDFC sets.



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

Figure 8: Runtime comparison of the full MILP with the MILP applied on individual CFDFC sets, described in Section 3.6.

4.4 MILP Runtime Analysis

The rightmost column of Table 2 reports the runtime of the MILP for performance optimization. In all our benchmarks, the runtime of the ILP for extracting the CFDFC was negligible (i.e., less than 0.1 *s*) in comparison to the MILP runtime. It is evident from the table that the MILP 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 3.6.

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 8a, which shows the obtained cost function value (i.e., the sum of the weighted CFDFC throughputs, as given in Equation 15), relative to the optimal cost function value for a given target CP. The graph depicts only the results of the benchmarks which take longer than 1 *s* to converge to the optimum value of 1. While it is clear that the convergence time is lower than the overall MILP 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 3.6 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 8b 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.

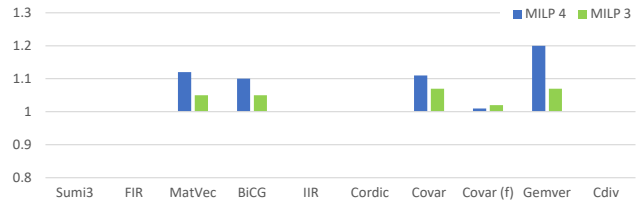
The two versions of the MILP which we have considered so far optimized *all* CFDFCs of the program. Our next experiment is based on the intuition that some CFDFCs do not contribute significantly to the execution time of the application (e.g., the outermost loop of a nested loop)—they can be removed from the cost function without a notable performance penalty. We demonstrate this effect in Figure 8c, where we compare the cost value of the MILP which optimizes the throughput 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.

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 3.6 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.

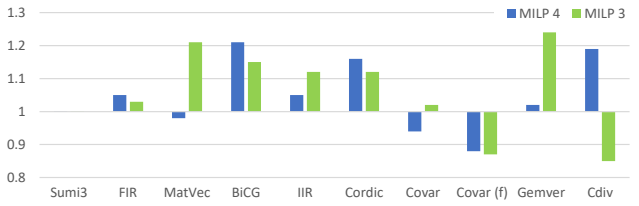
4.5 Comparison of MILP Solutions

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 3.6, and (3) in the cost function of 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).

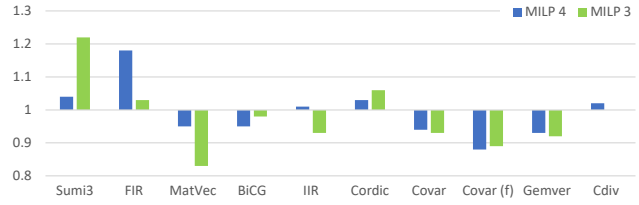
Figure 9 shows the cycle count, total execution time, and resource consumption of the solutions obtained in such a manner, relative to the optimal MILP solutions from Table 2. In applications which have a single CFDFC per set, the obtained cycle count is equal to the optimal because our heuristic covers 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



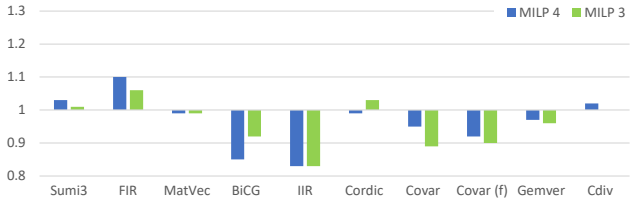
(a) Cycle count ratio (MILP in sets/full MILP).



(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).

Figure 9: 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).

Target CP (ns)	Achieved CP (ns)	II = 1/Θ	Execution Time (μs)	LUTs	FFs
–	9.1	2	15.7	1578	1665
8	7.7	1	7.7	1632	1742
6	5.7	1	5.7	1661	1810
4	4.1	1	4.1	1853	2084
3	3.6	2	7.2	2188	2696

Table 3: Exploration of the effectiveness of the clock period (CP) constraint on a tree of combinational adders.

the optimal MILP solutions—as the throughputs of certain loops are not optimized, fewer FIFOs are instantiated. All these variabilities are expected and in an acceptable range for the significant MILP runtime reduction which this heuristic approach offers.

4.6 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 3. 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 which 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.

5 RELATED WORK

In high-level synthesis, 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, 26, 29]: the aim is to create pipelines with the best possible loop initiation intervals under the given clock and resource constraints. Dataflow circuits [2, 15] 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 [6, 8] have been extensively used to create synchronous and asynchronous dataflow circuits; their timing properties can be analyzed using Petri net theory [3, 4, 24, 25]. Several approaches in asynchronous dataflow design have explored slack matching, i.e., adding pipeline buffers to prevent stalls. Venkataramani et al. [28] present a heuristic to avoid performance bottlenecks by inserting buffers to balance reconverging paths in asynchronous circuits. Najibi et al. [21] describe slack matching for asynchronous circuits with conditional computation and communication, where the conditions correspond to different circuit operation modes. In contrast to these works, our model considers retiming and slack matching simultaneously—we target *synchronous* dataflow circuits, so the clock period 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).

6 CONCLUSIONS

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 which achieves near-optimal results. On benchmarks obtained out of C code, we demonstrate the ability of our approach to achieve high-throughput, pipelined dataflow circuits. We believe that optimization techniques such as this one are the key to making dynamic scheduling truly competitive with existing HLS techniques.

ACKNOWLEDGMENTS

Lana Josipović is supported by a Google PhD Fellowship in Systems and Networking. Jordi Cortadella is supported by grants MINECO TIN2017-86727-C2-1-R and GENCAT 2017-SGR-786.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, first edition, 1986.

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

[3] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar. A general model for performance optimization of sequential systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 362–369, San Jose, Calif., Nov. 2007.

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

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

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-20(9):1059–76, Sept. 2001.

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

[8] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 657–62, San Francisco, Calif., July 2006.

[9] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, Jan. 2002.

[10] J. Forrest, T. Ralphs, S. Vigerske, LouHafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, H. G. Santos, rlougee, and M. Saltzman. coin-or/cbc: Version 2.9.9, July 2018.

[11] M. R. Greenstreet and K. Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI Signal Processing*, 2(3):139–148, Nov. 1990.

[12] G. Hoover and F. Brewer. Synthesizing synchronous elastic flow networks. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 306–11, Munich, Mar. 2008.

[13] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *Proceedings of the 8th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, Manchester, Apr. 2002.

[14] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, Mar. 1975.

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

[16] L. Josipović, A. Guerrieri, and P. Ienne. Speculative dataflow circuits. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 162–71, Seaside, Calif., Feb. 2019.

[17] R. Kastner, J. Matai, and S. Neuendorffer. Parallel programming for FPGAs. *ArXiv e-prints*, arXiv:1805.03648, May 2018.

[18] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, June 1991.

[19] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Proc. 4th International Conference on the Mathematics of Program Construction*, pages 272–285, London, June 1998.

[20] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–80, Apr. 1989.

[21] M. Najibi and P. A. Beerel. Slack matching mode-based asynchronous circuits for average-case performance. In *Proceedings of the 32nd International Conference on Computer-Aided Design*, pages 219–225, San Jose, CA, Nov. 2013.

[22] L.-N. Pouchet. *Polybench: The polyhedral benchmark suite*, 2012.

[23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, third edition, 2007.

[24] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Trans. Software Eng.*, 6(5):440–449, Sept. 1980.

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

[26] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64, Feb. 1996.

[27] J. Sparsø. Current trends in high-level synthesis of asynchronous circuits. In *Proceedings of the 16th IEEE International Conference on Electronics, Circuits, and Systems*, pages 347–50, Yasmine Hammamet, Dec. 2009.

[28] G. Venkataramani and S. C. Goldstein. Leveraging protocol knowledge in slack matching. In *Proceedings of the 25th International Conference on Computer-Aided Design*, pages 724–729, San Jose, CA, Nov. 2006.

[29] Z. Zhang and B. Liu. SDC-based modulo scheduling for pipeline synthesis. In *Proceedings of the 32nd International Conference on Computer-Aided Design*, pages 211–218, San Jose, CA, Nov. 2013.