Suppressing Spurious Dynamism of Dataflow Circuits via Latency and Occupancy Balancing

Jiahui Xu
ETH Zurich
Zurich, Switzerland

Lana Josipović
ETH Zurich
Zurich, Switzerland

ABSTRACT

Dataflow circuits produced via *high-level synthesis* (HLS) adapt their schedule at runtime to unpredictable data and control outcomes, thus promising superior performance to standard HLS solutions. However, their distributed handshake mechanism is extremely resource-expensive—there is a clear benefit in simplifying or removing it whenever it is unneeded for correctness and performance. Yet, even in such situations, transient and spurious stalls and irregular data exchanges prevent the systematic removal of handshake logic, thus resulting in an unnecessary resource overhead. In this work, we present a scalable strategy based on *linear programming* (LP) that eliminates unnecessary and spurious stalls via latency and occupancy balancing; the data exchange periodicity and predictability in the resulting circuits uncover new handshake logic removal opportunities and enable the formation of simple local controllers to replace it. We show that, in cases where dynamism is unneeded, our circuits qualitatively match those produced by standard HLS tools. Otherwise, our strategy allows us to systematically trade off area and performance to exploit various degrees of dynamism depending on the optimization objective.

CCS CONCEPTS

• **Hardware** → **Datapath optimization**; **Model checking**; • **Computer systems organization** → **Data flow architectures**.

KEYWORDS

High-level synthesis, dataflow circuits, model checking

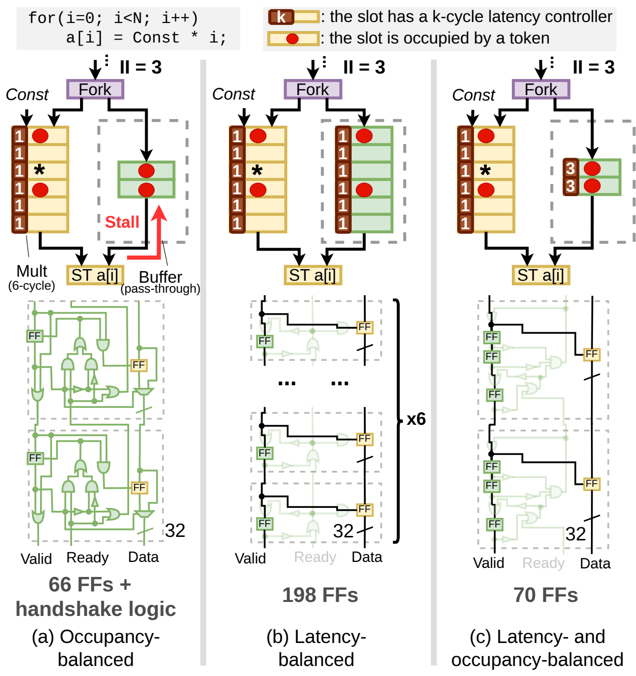
ACM Reference Format:
Jiahui Xu and Lana Josipović. 2024. Suppressing Spurious Dynamism of Dataflow Circuits via Latency and Occupancy Balancing. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626202.3637570

1 INTRODUCTION

Dataflow circuits are an attractive target for C-based *high-level synthesis* (HLS): their distributed handshake communication mechanism allows them to flexibly adapt their execution at runtime and achieve high performance when accelerating programs with unpredictable control flow and irregular memory accesses [17, 19]. Yet, this performance merit is not for free: the handshake logic often

This work is licensed under a Creative Commons Attribution International 4.0 License.

FPGA '24, March 3–5, 2024, Monterey, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0418-5/24/03.
https://doi.org/10.1145/3626202.3637570



```
for(i=0; i<N; i++)
    a[i] = Const * i;
```

Figure 1: Three different buffer placement schemes that achieve the same performance but differ in resource utilization. The dataflow circuit diagrams are shown above, and the corresponding implementations of the *Buffer* are depicted below with the removed logic shown shaded. Figure 1a employs occupancy balancing, Figure 1b employs latency balancing, and Figure 1c employs both latency and occupancy balancing. The latter is the most resource-efficient and what we aim to achieve in this work.

constitutes over 50% of the resource usage [35] and makes dataflow circuits unacceptably resource-expensive.

A promising way to reduce this expensiveness is to restrict the generality of the dataflow handshake logic—if a particular signal is never required due to a predictable and regular data exchange pattern, it can be simplified or removed without compromising the circuit's correctness. Yet, even when implementing perfectly regular workloads, the opportunities to exploit this insight are limited: due to the dynamic nature of dataflow circuits, data items are propagated through the circuit at arbitrary times and may arrive at data joining points out-of-sync. This causes irregular stalls that are, in many cases, irrelevant for performance, but hinder circuit simplification opportunities.

In this work, we present a linear programming–based methodology to systematically eliminate unwanted stalls and uncover new circuit logic optimization opportunities. Our approach strategically adds latencies to orchestrate data transfer and synchronization times. This facilitates the removal of redundant handshake logic and enables us to reason about the time particular data items *occupy* dataflow circuit registers (i.e., buffers); we exploit this information to reduce register count and complexity. On a set of benchmarks obtained from C code, we demonstrate the ability of our strategy to systematically vary the degree of dynamism in the produced dataflow circuits: our solutions either qualitatively correspond, in both area and performance, to classic HLS pipelines achieved by modulo scheduling, or outperform them by maintaining only the dataflow logic necessary for performance benefits. Our strategy unifies the design space of classic HLS with that of HLS-produced dataflow circuits, previously achieved by fundamentally different HLS algorithms and opposing toolchains. Furthermore, we show that our optimization strategy is faster and more scalable than prior dataflow optimization approaches, which makes it more suitable for the stringent development time requirements of HLS.

2 WHAT IS THE BEST WAY TO REMOVE SPURIOUS DATAFLOW STALLS?

Figure 1 illustrates an example of a dataflow circuit (we describe such circuits in detail in Section 3) that computes the value of $Const \cdot i$ (i.e., the datapath of the for-loop above the circuit); the input value of i is replicated by the fork, and sent to a 6-cycle multiplier and a buffer (we will discuss its implementation next); the result of the multiplication and the buffered iterator must be synchronized by the store ($ST\ a[i]$) and deposited to memory as a data-address pair. In the most general case, all shown units communicate with handshake signals to guarantee that data is delivered only if it is valid, and the recipient is ready to receive it; whenever they are unneeded (e.g., a unit never stalls the incoming data), it is advantageous to remove them [35]. For the purpose of this example, assume that every 3 clock cycles, a new value of i becomes available (i.e., the *initiation interval* (II) is 3).

There are several ways to place and size *buffers* while sustaining the II of 3. In Figure 1a, the buffer has 2 *slots* (i.e., it can simultaneously hold at most 2 data items) to maintain the rate of incoming data; yet, it needs to keep its handshake logic (see logic gates in the gate-level diagram below), as data will be stalled in the buffer while waiting for the long-latency *Mult* to compute. In Figure 1b, the latencies of the two paths from fork to store are equalized by adding extra buffers; as data from both paths always arrives at the store at the same time, the buffer is never stalled and can be simplified as shown below (i.e., all logic related to the *Ready* signal can be removed, shown shaded). Yet, the implementation of such a buffer (essentially, a 6-latency shift register) is redundant: as data enters the buffer every 3 cycles and the buffer always holds at most 2 data items, its slots (and, thus, resources) are underutilized. The implementation of Figure 1c combines the best of these two strategies: like the first approach, it contains the minimal number of buffer slots that sustain the desired II. It also equips each slot with a latency of 3 (i.e., data will be stored in the slot for 3 cycles and only then become accessible to the successor) to match the total latency of the multiplier, thus allowing the same logic simplification as the second solution. The resulting buffer implementation is cheaper and simpler, as shown in the lower portion of Figure 1c.

In the rest of this paper, we present a mathematical formulation that systematically achieves buffer configurations such as the one in Figure 1c: it minimizes the buffer requirements while uncovering new handshake logic removal opportunities, without compromising the circuit's performance. The rest of this paper is organized as follows: Section 3 provides the background on dataflow circuits and outlines what others have done to optimize their area and performance. Section 4 describes our latency balancing strategy that qualitatively achieves the circuits of Figure 1b. We then complement it with a strategy to calculate token occupancy in Section 5. We combine these two approaches in Section 6 to achieve buffer configurations such as the one in Figure 1c. In Section 7, we discuss the ability of our approach to systematically trade off area and performance. Finally, we evaluate all these aspects in Section 8.

3 BACKGROUND AND RELATED WORK

In this section, we describe dataflow circuits and existing methods for their area and performance optimization; we illustrate why these methods are inadequate for taking full advantage of removing redundant handshake logic.

3.1 Dataflow Circuits

Dataflow circuits are built from *units* that communicate with their predecessors and successors via latency-insensitive *channels*, composed of data and handshake signals [10, 17]. Once the control and memory dependencies have been resolved, units exchange data encapsulated in *tokens*; the exchange time is determined *dynamically* during circuit execution. Many works study the generation process of dataflow circuits from high-level code [3, 6, 11, 12, 17]. We here focus on an approach that targets C code [17]; the generated circuits implement single-threaded programs, which is in line with the standard, statically-scheduled, HLS tools [5, 17, 21, 32].

The circuits we consider organize units into *basic blocks* (BBs), i.e., straight pieces of code without internal control flow decisions. They are built from the following units: A *fork* distributes a copy of the incoming token to each of the successors as soon as they are ready to receive it. A *join* synchronizes multiple tokens before sending a token to its successor; it is typically used in arithmetic units to ensure the presence of all inputs prior to computing. A *merge* propagates a token to its single output from one of its two data inputs. A *branch* propagates the received data token to one of its successors, depending on the value of the received condition token. A *buffer* is used to store tokens, break combinational paths, and increase throughput. In general, buffers can be arbitrarily placed inside the circuit without impacting its functionality [4, 17, 20].

3.2 Performance Optimization of Dataflow or Latency-Insensitive Systems

Performance optimization of synchronous and asynchronous dataflow circuits has been extensively studied. *Occupancy balancing*, also referred to as *slack matching* and *bubble insertion*, aims to avoid computation-bottlenecking pipeline stalls [1, 4, 20, 23, 25, 29]. The goal is to identify the number of tokens that can *occupy* a channel and instantiate the appropriate number of buffer *slots* to accommodate them, without stalling the preceding computation. In the context of HLS, a recent work [20] relies on occupancy balancing to optimize the performance of HLS-produced dataflow circuits implementing program loops; it is based on *mixed-integer linear programming* (MILP). The goal is to maximize *throughput* (i.e., the inverse of the loop II) of circuit subgraphs corresponding to the program's control flow graph cycles; these subgraphs are referred to as *choice-free circuits* (CFCs), as they internally contain no control flow choices. As we target HLS circuits as well, our strategy will rely on the same concepts.

The works above qualitatively achieve the solution of Figure 1a: although they can achieve the best throughput with the minimal buffering required to support it, the resulting circuits are not well-suited for handshake logic optimization. Thus, occupancy balancing on its own is not sufficient to achieve our goal of simultaneously tackling performance and logic optimization.

Latency balancing is an alternative way to address pipeline stalls: the idea is to equalize the delays of joining paths, thus ensuring that data arrives at the joining point at the same time. This strategy is usually employed in feed-forward dataflow pipelines [13, 22] or in systems where the simultaneous arrival time of data is a prerequisite for correctness [15]. As suggested in Figure 1b, this approach is more effective in stall removal than occupancy balancing; yet, it may come at a notable buffering overhead. Our work aims to exploit the former benefit and avoid the latter cost.

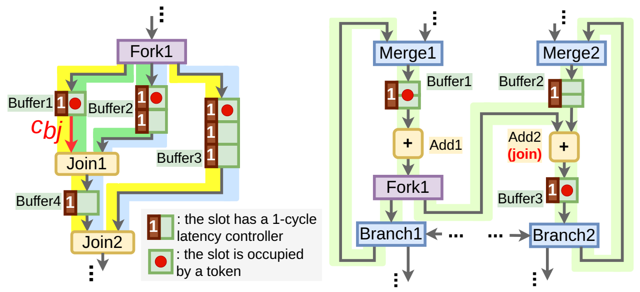
3.3 Eliminating Redundant Handshake Logic

Prior studies [34, 35] have suggested that the absence of stalls implies opportunities for removing redundant handshake logic. They apply formal methods, e.g., model checking, to guarantee the correctness of the underlying circuit transformation and obtain cheaper circuits without performance degradation (i.e., no sequential behavior is altered). However, the success of these approaches significantly depends on the construction and performance optimization strategy of the input circuit—any irregular data exchange and stall will dramatically reduce their advantage. For instance, as discussed earlier, the circuit of Figure 1a exhibits a stall on the buffer—attempting to remove the buffer logic of the circuit in this form will be fruitless. The purpose of our work is to overcome this limitation and systematically produce high-performance circuits that are also amenable to handshake logic removal.

Several approaches noted the benefits of combining static and dynamic scheduling [8, 9]—they aim to benefit from the area efficiency of the prior and the performance advantage of the latter approach. While Cheng et al. [8] introduce static circuit regions into dataflow circuits and Szafarczyk et al. [28] opt for the opposite approach, both efforts ultimately contain general handshake logic that suffers from the overheads illustrated in Figure 1a; thus, they could both benefit from our stall and logic removal strategy. Furthermore, in contrast to these works, our strategy effectively combines the benefits of static and dynamic circuits without requiring sophisticated code restructuring [28] or stitching together solutions produced by different HLS toolchains [8].

4 MINIMIZING STALLS VIA LATENCY BALANCING

In this section, we describe our *linear programming* (LP) formulation for minimizing the presence of stalls in dataflow circuits via latency balancing. Our formulation reasons about *synchronization patterns* of the circuit, i.e., circuit subgraphs in which multiple tokens need to synchronize; our goal is to latency-balance such paths to avoid token stalls. For instance, in the example of Figure 1, our formulation will identify the need to incorporate 6 cycles of latency on the right path, so that tokens arrive at the store simultaneously and stalls never occur. The rest of this section formalizes the notion of synchronization patterns and details our LP model.



(a) Two fork-join patterns (b) A pair of synchronizing cycles

Figure 2: Synchronization patterns that we aim to balance. Figure 2a: Two fork-join patterns *Fork1-Join1* and *Fork1-Join2*; the two pairs of reconvergent paths of pattern *Fork1-Join2* are highlighted in blue and yellow. Figure 2b: A pair of synchronizing cycles that join at *Add2*; all channels related to the pattern are highlighted in green.

The main source of stalls in dataflow circuits is the difference in token arrival time at the inputs of a join. We classify the circuit subgraphs where this occurs into noncyclic *fork-join patterns* and cyclic *synchronizing cycles*. We model the dataflow circuit as a graph, where the vertices and edges are units and channels. A *path* is a sequence of channels joining a sequence of distinct units; a *cycle* is a path whose first and last units are identical.

Definition 1. A pair of *reconvergent paths* inside a dataflow circuit are two paths with the same nodes *only* at the beginning (i.e., a fork) and the end of the paths (i.e., a join); the channels and units on these two paths must belong to the same BB sequence.

The circuit in Figure 2a shows a circuit of a single BB. There is one pair of reconvergent paths between *Fork1* and *Join1* (the pair of paths is shaded in green).

Definition 2. A *fork-join pattern* is the set of *all* reconvergent paths between a pair of a fork and a join that follow the same BB sequence.

Figure 2a has two fork-join patterns: (1) *Fork1-Join1*, which has one pair of reconvergent paths, shaded in green, (2) *Fork1-Join2*, which has two pairs of reconvergent paths, shaded in yellow and blue (the left two paths between *Fork1* and *Join2* are not a pair of reconvergent paths, since they share *Join1* and *Buffer4*).

Definition 3. Two cycles are a pair of *synchronizing cycles* in a dataflow circuit if the following properties hold: (1) The two cycles are disjoint (i.e. they do not have any common units) and belong to the same CFC (defined in Section 3.2). (2) There exists at least one join that is reachable from both cycles without crossing any edge on the cycle in the CFC they belong to.

Figure 2b describes a sub-circuit that (1) contains two *disjoint* cycles of a single CFC, (2) there is a join (*Add2*) that synchronizes the execution of the cycles: the left cycle can reach *Add2* via the channel between *Fork1* and *Add2*, and the right cycle contains *Add2*. It satisfies the definition, thus, it is a pair of synchronizing cycles.

When paths in synchronization patterns differ in latency, they cause stalls that prevent circuit optimization. Thus, our goal is to identify and balance the latency of such patterns. Both examples in Figure 2 exhibit latency imbalance. For instance, the reconvergent path between *Fork1* and *Join1* in Figure 2a is imbalanced; since the left path has a smaller latency, its token has to wait (i.e., *Buffer1* is stalled). Similarly, in Figure 2b, the left cycle can issue data from *Fork1* to *Add2* only every second cycle and its data is stalled in the meantime. The solution to both problems is to insert an additional cycle of latency on the fast path, thus slowing down the data and removing the stall, without degrading circuit performance.

In the rest of this section, we discuss our latency balancing LP model. Its constants and variables are summarized in Table 1.

Synchronization patterns constraints. We compute $Latency(p)$, the total latency on a path p as

$$Latency(p) := (\sum_{u \in p} \mathbf{D}^u + \sum_{c \in p} L^c), \tag{1}$$

where c and u are channels and units on p; L^c and $\mathbf{D}^u$ denote the latency of c and u. For example, in Figure 1, the path *Fork1*, *Mult*, *ST* has a latency of 6.

From a given synchronization pattern *pattern*, we enumerate all pairs of constructs (i.e., reconvergent paths or synchronizing cycles); for each pair $p_1, p_2 \in pattern$, we formulate the following constraint to define the presence of imbalance (indicated by a binary variable $X^{pattern}$):

$$\begin{aligned} \mathcal{M} \cdot X^{pattern} &\geq (Latency(p_1) - Latency(p_2)), \\ \mathcal{M} \cdot X^{pattern} &\geq (Latency(p_2) - Latency(p_1)), \end{aligned} \tag{2}$$

where $\mathcal{M}$ is a very large constant (larger than any two constructs' latency difference). Whenever a pair of constructs has a non-zero latency difference, the right-hand side of one of the constraints must be greater than 1 and its division with $\mathcal{M}$ is a positive fractional number; the only binary solution that satisfies this constraint is $X^{pattern} = 1$. By balancing pattern latencies, we aim to reduce the number of patterns where this occurs (i.e., equalizing the latencies allows $X^{pattern} = 0$). For example, the *Fork1*-*Join1* pattern in Figure 2a has two imbalanced paths (i.e., the left path has a latency of 1 and the right path has a latency of 2). The difference between the latencies will set $X^{pattern} = 1$. Adding a 1-cycle latency to the left path (e.g., setting $L^{c_{bj}} = 1$ for channel c_{bj} colored in red) would balance the structure and relax this constraint.

Dataflow units can have variable latency [16]. In such cases, there is no opportunity to balance the pattern that contains it. For instance, if the multiplier in Figure 1 took either 4 or 6 cycles to compute, one could not find any latency assignment for the right path that balances both computational scenarios—neither a latency of 4 nor of 6 would be useful and the registers implied by the additional latency would only increase the resource consumption. Thus, whenever a pattern contains a unit of variable latency, we fix the value of $X^{pattern}$ as follows

$$X^{pattern} = 1, \tag{3}$$

which indicates that a pattern cannot be balanced and prevents the insertion of redundant latencies.

Input parameters (constants):		
$\mathbf{D}^u$	$\mathbb{N}$	The maximum execution latency of unit u.
$\mathbf{II}_{CFC_i}$	$\mathbb{N}$	The user-required II of CFC_i.
$\mathbf{B}_c$	$\mathbb{N}$	The bitwidth of channel c.
Output variables:		
L^c	$\mathbb{N}$	The number of extra latencies on channel c.
Internal variables:		
$X^{pattern}$	$\mathbb{B}$	If the synchronization pattern *pattern* is not balanced.
S^c	$\mathbb{B}$	If channel c is stalled.
R^c	$\mathbb{B}$	If channel c has $L > 1$, i.e., channel cut.

Table 1: The variables and constants used in the LP model for latency balancing (described in Section 4).

Stall identification constraints. When a pattern is imbalanced, channels that are *related* to it will be stalled. A channel c is related to a fork-join pattern if it belongs to it; similarly, it is related to a pair of synchronizing cycles if it satisfies one of the following properties: (1) c belongs to one of the synchronizing cycles. (2) c belongs to a path from any unit on the synchronizing cycles to any join (reachable from both synchronizing cycles, see Definition 3-(2)) and c is not on any cyclic path $l \in CFC_i, \forall i$. For example, in Figure 2b, all channels highlighted in green are related to the pattern. We formulate the following constraint for each $c \in pattern$ to define the presence of stalls in terms of imbalanced patterns:

$$S^c \geq X^{pattern}. \tag{4}$$

If all *pattern* that contain c are balanced, then the binary variable S^c becomes a free variable. Otherwise, S^c is set to 1, indicating the presence of a stall on c.

Note that Equation 4 sets *all* pattern channels to *stalled* when the pattern is imbalanced; in practice, this is not necessarily the case (i.e., some channels could be stalled and others not). Yet, this simplified consideration has no impact on the balancing quality (i.e., the position and number of inserted latencies depends exclusively on pattern latencies, as specified by Equation 2); furthermore, it allows us to prioritize the optimization of larger patterns (i.e., those with more channels), as our objective function will illustrate.

Cycle time constraints. Adding extra latencies may reduce the number of stalls and, consequently, resources; yet, adding latencies on cyclic paths may compromise the circuit's *cycle time*—the *best possible value* for the II of a CFC CFC_i, defined as the maximum of the sum of latencies on all cycles [1]. For instance, the circuit in Figure 2b has two buffers (i.e., two clock cycles of latency) on the right cycle; thus, cycle time is 2. Adding a third buffer to increase the cycle latency to 3 would increase the best possible II to 3. We formulate the following constraints for each cycle l to ensure that the cycle time does not exceed the desired II:

$$1 \leq Latency(l) \leq \mathbf{II}_{CFC_i}. \tag{5}$$

This prevents the insertion of latencies beyond the predefined II on throughput-critical cycles.

For instance, in Figure 2b, this constraint would allow adding at most a single cycle of latency on the left cycle, but prevent the addition of any latency onto the right cycle. In line with standard performance optimizations for dataflow circuits [20], we apply the cycle time constraints to the CFC that correspond to performance-critical program loops (e.g., the innermost loops of a loop nest) and

Input parameters (constants):		
$\mathbf{D}^u$	$\mathbb{N}$	The maximum execution latency of unit *u*.
$\mathbf{II}_{CFC_i}$	$\mathbb{N}$	The best possible II of CFC_i.
$\mathbf{B}_c$	$\mathbb{N}$	The bitwidth of channel *c*.
$\mathbf{L}^c$	$\mathbb{N}$	The number of extra latencies on channel *c*.
Output variables:		
N^c_{max}	$\mathbb{N}$	Maximal token occupancy on channel *c*.
Internal variables:		
S^c	$\mathbb{B}$	If channel *c* has $N^c_{max} > \mathbf{L}^c$.
$N^c_{CFC_i}$	$\mathbb{R}^+_0$	The token occupancy on channel *c* required by CFC CFC_i.
$N^u_{CFC_i}$	$\mathbb{R}^+_0$	The token occupancy in unit *u* required by CFC CFC_i.

Table 2: The variables and constants used in the LP model for occupancy balancing (described in Section 5).

omit them otherwise, so that we increase the benefits of latency balancing without notable performance degradation.

Latency cost constraints. Adding latency eliminates stalls and saves resources, but implies additional registers; regardless of the controller type (see the gate-level description in Figure 1), the cost increases with the number of bits to store. Thus, it is desirable to place latencies on channels carrying the fewest bits. We characterize each channel c with a decision variable R^c, and define it as

$$\mathcal{M} \cdot R^c \geq L^c \geq R^c, \tag{6}$$

that is, if latency $L^c \geq 1$, $R^c = 1$; otherwise, $R^c = 0$, indicating that no latency will be placed here. R^c is minimized in the objective function and weighted by the bitwidth, as we will see next.

Critical path constraints. We borrow standard techniques [4, 20, 27] to control the critical path in the circuit: we sum up combinational delays across units and channels and break paths with registers to honor a clock period target (i.e., whenever a delay is longer than the target, we set $R^c = 1$ to indicate the presence of a non-zero sequential delay).

Objective function. Our goal is to minimize (1) the stalled channels c (S^c) due to an imbalanced synchronizing pattern and (2) the cost of extra latencies needed to suppress the imbalance. We formulate the cost function as

$$minimize \quad \underbrace{\alpha \cdot \sum_c S^c}_{stalled\ channels} \; + \underbrace{\beta \cdot \sum_c (\mathbf{B}^c \cdot R^c + L^c)}_{extra\ latency\ cost}. \tag{7}$$

We prioritize the minimization of the number of channels that are stalled (i.e., $\sum_c S^c$); thus, the objective function will aim to balance any imbalanced patterns via latency insertion. To add latencies only where necessary, we minimize their cost as a secondary objective. Thus, we set $\alpha \gg \beta$; these parameters could be tuned for other objectives and tradeoffs.

The output values L^c from solving the LP problem indicate the positions and values of latencies to add; we will rely on this notion in the next section when reasoning about token occupancies and in Section 6 when deciding the final buffer configuration.

5 BALANCING TOKEN OCCUPANCY

The LP formulation of the previous section informs us how to balance latencies in a dataflow circuit; yet, as Figure 1b suggests, this information is insufficient to obtain the best buffer configuration: naively inserting pipeline buffers to achieve the desired latency may result in underutilized buffers and wasted resources. In this section, we complement the previous LP with an additional LP that determines the largest token *occupancy*, i.e., the number of tokens that reside in each channel; we will later exploit this information to ensure that our circuits have no redundant and unused buffers.

All variables of our LP model are listed in Table 2; two of its inputs, $\mathbf{L}^c$ and $\mathbf{II}_{CFC_i}$, are provided by the previous LP, whereas the other two inputs are intrinsic circuit properties. We describe our constraints and objective function in the rest of this section.

Channel and unit occupancy constraints. In dataflow circuits, whenever a CFC is active (e.g., a program loop is repeating), each unit and channel receives a token every II cycles. Since every accepted token has to be kept by the channel or unit for $\mathbf{L}^c$ or $\mathbf{D}^u$ cycles, a lower bound for token occupancy in units and channels must be set to support this behavior; otherwise, the pipeline stalls, which will be harmful to both performance and area. For each CFC CFC_i, for each channel $c \in CFC_i$, we formulate the following constraint:

$$\frac{\mathbf{L}^c}{\mathbf{II}_{CFC_i}} \leq \mathbf{N}^c_{CFC_i}. \tag{8}$$

For example, in Figure 1, the right channel between *Fork1* and *ST* (where we should place *Buffer*) has $L = 6$ and $II = 3$, therefore, the occupancy is at least 2. The occupancy bound is set analogously for each unit $u \in CFC_i$:

$$\frac{\mathbf{D}^u}{\mathbf{II}_{CFC_i}} \leq \mathbf{N}^u_{CFC_i} \leq Capacity^u. \tag{9}$$

The only difference is the upper bound $Capacity^u$, which is the capacity of the unit and it must be fixed and finite. The capacity $Capacity^u$ for pipelined units is simply $\mathbf{D}^u$.

Occupancy balancing constraints. For brevity, we denote the total occupancy on a path or cycle p in a CFC CFC_i as

$$Occupancy_{CFC_i}(p) := (\sum_{u \in p} N^u_{CFC_i} + \sum_{c \in p} N^c_{CFC_i}). \tag{10}$$

The token occupancy on reconvergent paths must be identical to sustain the throughput goal (anything else would imply that the two paths process a different number of tokens, which is impossible due to their common origin and joining point). For each CFC CFC_i and each pair of reconvergent paths $p_1, p_2 \in CFC_i$, we enforce

$$Occupancy_{CFC_i}(p_1) = Occupancy_{CFC_i}(p_2). \tag{11}$$

Dataflow circuits limit the number of tokens on a cyclic path to ensure functional correctness [10, 17, 34]. For each CFC CFC_i and each cycle $l \in CFC_i$, we formulate the following constraint:

$$Occupancy_{CFC_i}(l) \leq B. \tag{12}$$

For dataflow circuits generated from sequential programs, $B = 1$, i.e., there must be no more than one token per cyclic path during the steady state of the choice-free circuit [17, 20].

When two CFCs, CFC_i and CFC_j with different IIs share a channel c (e.g., a channel that belongs to both an inner and an outer program loop), they might require c to have different occupancies values (i.e., $N^c_{CFC_i} \neq N^c_{CFC_j}$). To fulfill the requirements of both CFCs, the final channel occupancy N^c_{max} must be greater than any *per-CFC* channel occupancies value (i.e., $N^c_{CFC_i}, \forall i$). For each channel c, each CFC CFC_i such that $c \in CFC_i$, we formulate the following constraint; it defines the required token occupancy on c for sustaining the throughput of all CFC_i:

$$N^c_{max} \geq N^c_{CFC_i}. \tag{13}$$

Note the duality in how conflicts (i.e., situations when dynamism is required) are handled in *occupancy balancing* and *latency balancing*. If multiple synchronization patterns (see Section 4) share any channel or unit and demand *conflicting latency values*, our objective function from Section 4 will be able to balance *at most one of them* (i.e., it will either identify a pattern whose stalls significantly contribute to the cost function or prioritize the second objective function term to favor latency savings). On the other hand, if multiple CFCs demand *conflicting occupancy values* for a channel or unit, the *maximal occupancy* obtained by Equation 13 will satisfy the requirement of all of them, as it specifies the maximum number of tokens that will reside in the channel or unit at any execution point. We will discuss our buffer implementations that consider these two aspects jointly in Section 6.

Objective function. We aim to minimize the sum of N^c_{max} over all channels, weighted by the channel bitwidth $\mathbf{B}^c$, i.e.,

$$minimize \sum_c \mathbf{B}^c \cdot N^c_{max}. \tag{14}$$

The output values N^c_{max} from solving the LP problem of this section indicate the maximum token occupancy of each channel. We use this information in the next section to configure the buffers.

Combining the two LP formulations. A natural question to ask is whether the two LPs that we employ consecutively could be combined into a single LP problem. Although, in principle, possible, a combined strategy is not naturally linear due to the occupancy calculation $\frac{L^c}{\mathbf{II}_{CFC}}$ in Equation 8, as both the dividend and divisor are variables (e.g., an increase of latency L^c may simultaneously change the II, as discussed in Section 4). Such an equation could be linearized (e.g., by discretizing and constraining the values of $\mathbf{II}_{CFC}$ and $\frac{L^c}{\mathbf{II}_{CFC}}$ [2]) at an additional problem complexity. The same considerations hold for unit occupancy (i.e., Equation 9) as well. Instead, we opt to solve the two simpler problems separately; our second LP takes L^c and II as input constants (see Table 2) thus simplifying the occupancy calculation. Although a separate formulation may lead to a globally suboptimal solution (e.g., in terms of buffer count or critical path), our evaluation shows that our circuits are systematically smaller and faster than their counterparts produced via alternative dataflow optimization techniques. Furthermore, our strategy to separate the two concerns is significantly simpler and, thus, faster than strategies that consider them simultaneously, as we will demonstrate in Section 8.

6 PLACING BUFFERS

In this section, we will use Figure 3 as an example to discuss which buffer should be placed on a channel, and how its handshake controller should be implemented based on the combination of the values of the channel latency (L^c) and token occupancy (N^c_{max}), obtained from the LPs in the previous sections.

Case 1: $\mathbf{N}^c_{max} = 0$. If $L^c = 0$, no buffer is needed. If $L^c > 0$, the channel needs latency balancing, which only occurs on channels that do not contribute to any loop (otherwise, circulating tokens would occupy the channel and $N^c_{max} > 0$); thus, they transfer

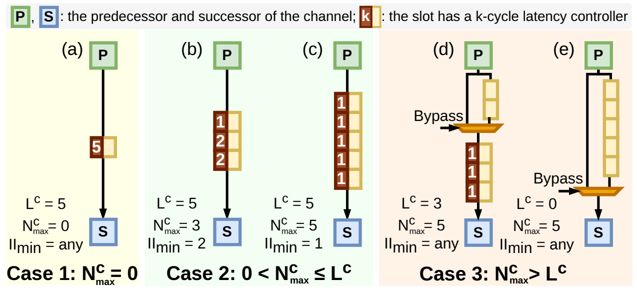


Figure 3: Examples of our buffer configuration scheme based on the relation between channel latency L^c and token occupancy N^c_{max}.

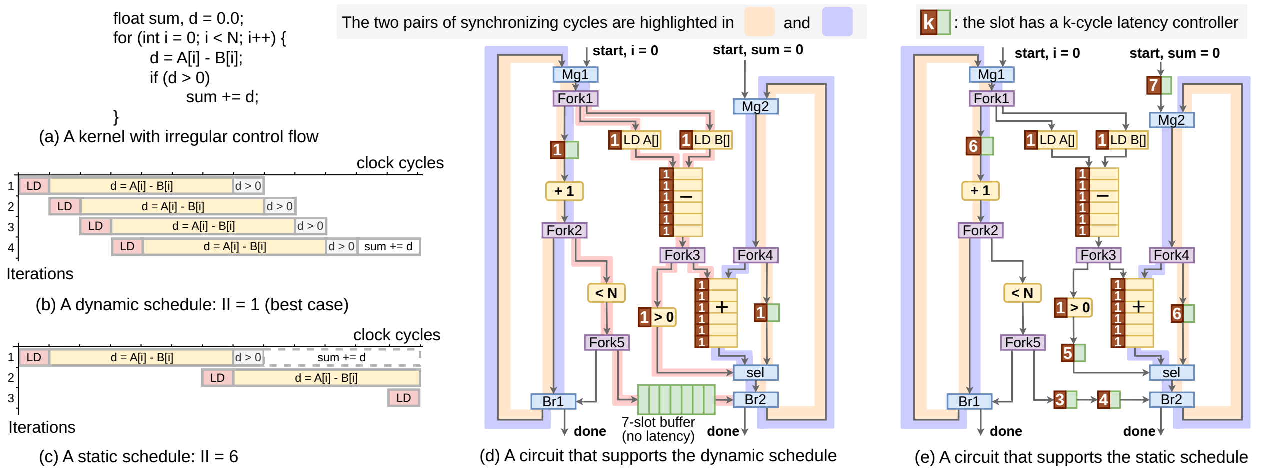
exactly one token [17, 20]. To accommodate this token for L^c cycles, we place a 1-slot buffer with L^c-cycle latency, as shown in Figure 3a.

Case 2: $0 < \mathbf{N}^c_{max} \leq \mathbf{L}^c$. We split an L^c-cycle latency into N^c_{max} buffers; each buffer must have latency $L \leq \mathbf{II}_{min}$, where $\mathbf{II}_{min}$ is the minimum II among the CFCs that c is in. To achieve this, we (1) instantiate N^c_{max} buffer slots and (2) distribute latencies among them as equally as possible by increasing the latency of each buffer by 1 in a round-robin fashion until we run out of latency to distribute, such that none of them is greater than II. In step (2), a solution always exists because Equation 8 guarantees that there are enough slots to distribute the latency. Figure 3b and 3c illustrate how we split a 5-cycle latency into 3 and 5 buffers, respectively.

Case 3: $\mathbf{N}^c_{max} > \mathbf{L}^c$. This scenario occurs when the channel belongs to multiple constructs and has conflicting latency and occupancy values; the objective function of the first LP (Equation 7) opts for the smaller latency of one construct and Equation 13 of the second LP for the larger occupancy of another construct (in line with the discussion of Section 5). To preserve the latency chosen by the former LP, we place L^c slots, each with a latency of 1. To accommodate the occupancy chosen by the latter, we append to it a buffer of $N^c_{max} - L^c$ slots with no latency (i.e., a FIFO with a combinational bypass, also referred to as a transparent buffer [10]). Two such implementations are illustrated in Figures 3d and 3e.

Handshake controller. Dataflow buffers, regardless of their exact implementation, communicate with their predecessor and successor units using the same handshake protocol as the rest of the dataflow circuit. After a token enters a buffer, the buffer's validity is set; it remains visible to the successor from the next clock cycle (as the buffer is, typically, a sequential element) until the successor consumes the token [10, 27]. When the token transfer time is unknown, this handshake flexibility is exactly what we desire; yet, as discussed in Section 2, it is also resource-expensive.

Our buffer placement scheme of this section specifies, for each buffer slot, the exact latency L that the token will reside in the buffer. This offers an opportunity for a simpler and customized buffer implementation. We equip our buffer with a controller that dictates when the buffer should accept a token, how long it should store it, and when it should issue it to the successor. After accepting a token, the controller keeps the buffer invalid for L cycles; in cycle $L+1$, the buffer becomes valid, and the token is, thus, observable by the successor. As soon as the successor is ready to accept the token, it is transferred, and a new token can enter the buffer; otherwise, the buffer stalls any token that may be arriving at its input.



```
float sum, d = 0.0;
for (int i = 0; i < N; i++) {
        d = A[i] - B[i];
        if (d > 0)
                sum += d;
}
```

(a) A kernel with irregular control flow

(b) A dynamic schedule: II = 1 (best case)

(c) A static schedule: II = 6

(d) A circuit that supports the dynamic schedule

(e) A circuit that supports the static schedule

Figure 4: Pivoting from a dynamically-scheduled to a statically-scheduled circuit. Figure 4a: A kernel that contains irregular control flow. Figure 4b: A dynamic schedule that takes advantage of conditional operation skipping. Figure 4c: A schedule produced by a static scheduler. Figure 4d: Dataflow circuit optimized to support the dynamic schedule. Figure 4e: Corresponding dataflow circuit optimized for resources.

The implementation of a buffer with a fixed-latency controller is trivial and exactly as shown in the motivating example of Figure 1c: there is a single register for the data and an L-stage shift register for the forward-going valid signal, enabled by the ready signal of the successor. If the successor is ready to accept the token as soon as the valid signal is set, the corresponding logic can be omitted (shaded in the figure). Such logic simplification opportunities are enabled by balancing the latencies of joining paths and configuring our buffers to control the arrival time of the data accordingly.

7 TRADING DYNAMISM FOR SMALLER RESOURCE UTILIZATION

So far, our strategy has introduced new handshake logic removal opportunities while always prioritizing the circuit performance. Thus, whenever dynamic scheduling is beneficial, our approach maintains the necessary dynamism and the associated resource cost; as we will show in Section 8, in such cases, our circuits are faster but larger than their statically scheduled counterparts. One might prefer to trade off area and performance differently; we here describe how our strategy can be employed to achieve this.

Figure 4a shows a code with unpredictable control flow [17]: the decision *if* $(d > 0)$ depends on the value of d that is only available during runtime. A dynamically scheduled circuit can proceed with the execution if there is no dependency between the following and the current iteration to produce a high-performance schedule as in Figure 4b. In contrast, a statically scheduled circuit has to assume the loop-carried dependency of the long-latency operation *sum+=d* always exists, thereby producing a schedule like Figure 4c.

The two schedules represent two design extremes: a fast and expensive circuit, and a slow and cheap circuit. They are typically produced by fundamentally different HLS strategies (i.e., dynamic HLS, relying on handshake logic to achieve high performance in unpredictable situations, and static HLS, based on modulo scheduling algorithms that pipeline circuits with a *fixed* and conservative II), which makes it challenging to switch between them or explore different intermediate points. Our approach offers an entirely new opportunity for such explorations: our LP formulations are suited to tune the target performance, uncover handshake removal opportunities, and generate circuits with different degrees of dynamism.

The idea is similar to a standard modulo scheduler [36]: instead of optimizing the circuit for a fixed II goal, we iteratively solve the LP problem described in Section 4 to determine the *minimum* **II** such that *all* synchronization patterns are balanced. The search starts with a minimal $\mathbf{II}_{CFC_i}$ (see Table 1) of 1, which is then incremented after each iteration. In addition to the cycle time constraint described in Equations 5, we constrain the latency of each cycle $l \in CFC_i$ using the $\mathbf{II}_{CFC_i}$ value of the current iteration as

$$Latency(l) \geq \mathbf{II}_{CFC_i}, \tag{15}$$

which sets the loop II to be exactly $\mathbf{II}_{CFC_i}$; no other input constants, constraints, or objective function has to be changed. Whenever the solver returns a solution with all synchronization patterns balanced (i.e., $X^{pattern} = 0$ for each *pattern*), the optimizer terminates; after balancing the token occupancy and placing the buffers, we obtain the most performant solution that does not require expensive handshake logic. If any construct contains units with variable latency, removing all stalls is impossible, as the stalls and the handshake signals that implement them are fundamentally required to support the unit's latency variability. In such cases, Equation 3 enforces $X^{pattern} = 1$ and we can terminate the algorithm immediately, without performing this exploration.

Consider the circuit in Figure 4d, which has two pairs of synchronizing cycles: (1) the iterator computation on the left, with a latency of 1, and the 6-latency sum update on the right (both shaded in blue), and (2) the iterator computation on the left and the 1-latency cyclic propagation of the sum (both shaded in orange). Only one of the two pairs will be executed in a given iteration, depending on the value of d. Assume that we optimize the circuit for the best-case II of 1. To maintain this II, no latencies should be added on the second pair of cycles (to honor Equation 5). This prevents the balancing of the first pair of cycles (i.e., their latencies remain 1 and 6), thus causing stalls in the corresponding imbalanced pattern (the blue cycles and the red paths that connect them). These stalls imply

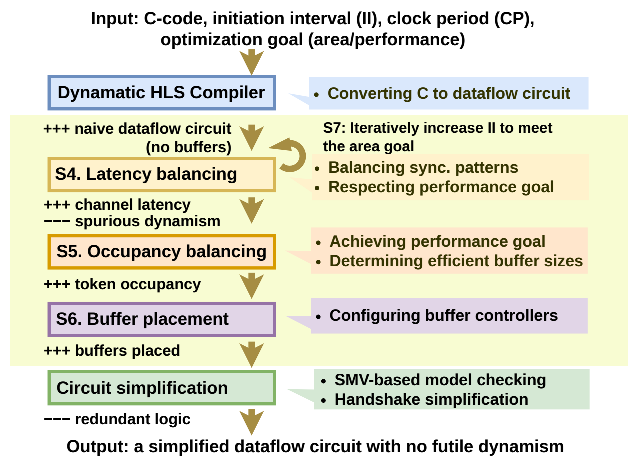

Input: C-code, initiation interval (II), clock period (CP), optimization goal (area/performance)
Dynamatic HLS Compiler
• Converting C to dataflow circuit
+++ naive dataflow circuit (no buffers)
S7: Iteratively increase II to meet the area goal
S4. Latency balancing
• Balancing sync. patterns
• Respecting performance goal
+++ channel latency
−−− spurious dynamism
S5. Occupancy balancing
• Achieving performance goal
• Determining efficient buffer sizes
+++ token occupancy
S6. Buffer placement
• Configuring buffer controllers
+++ buffers placed
Circuit simplification
• SMV-based model checking
• Handshake simplification
−−− redundant logic
Output: a simplified dataflow circuit with no futile dynamism

Figure 5: Our dataflow circuit optimization flow. Our contributions are highlighted with a colored background (*S4, S5, S6, S7*).

that all handshake logic must remain intact to allow the circuit to dynamically switch between the IIs of the slow and the fast cycles, as the schedule in Figure 4b suggests. However, if we relax the II goal from 1 to 6, we can balance *both* pairs of synchronizing cycles by setting the latencies of all cycles to 5. The entire circuit, shown in Figure 4e, is balanced and exhibits no stalls; thus, all its handshake logic can be removed. Ultimately, the circuit in Figure 4e will qualitatively correspond to a statically scheduled circuit and achieve the same schedule of Figure 4c. Interestingly, while static HLS obtains this schedule via modulo scheduling, we reach this point in an entirely different manner (i.e., by removing dynamism from dataflow circuits using the techniques of the previous sections).

8 EVALUATION

In this section, we describe the effectiveness of our strategy in removing spurious transient behaviors of the circuit, thus uncovering new opportunities for removing redundant logic.

8.1 Methodology

The flow of our circuit generation framework is summarized in Figure 5. The research artifact is publicly available [33].

The inputs to our framework are the HLS C code, the desired IIs of all performance-relevant program loops, and the target *clock period* (CP). We use *Dynamatic* [18], an open-source HLS compiler, to generate the unbuffered dataflow circuit from C code. For the circuit produced by Dynamatic, we solve the latency balancing problem (Section 4); it assigns channel latencies to balance synchronization patterns, maintain the target IIs, and control the critical path to achieve the target CP. We incorporate the latency values in the occupancy balancing problem (Section 5) to determine the channel occupancies. We determine the buffer size and controller type from the assigned latency and occupancy (Section 6). The resulting circuit, with its customized buffering and stall-eliminating latencies, is amenable to handshake logic simplification.

Removing handshake logic reduces the generality and flexibility of the handshake protocol; thus, we have to rely on formal verification to guarantee that our optimization does not penalize circuit correctness in any way. We follow Xu et al. [35] for generating verification models from the circuit descriptions. The verification is carried out by a model checker, and each verification goal (i.e., absence of stall) is formatted as a formal property; the checker either produces a proof if the property holds, or a counter-example if it fails. If the verifier concludes that a stall never occurs, the ready signal can be detached from the sender and replaced with a constant-1 signal; otherwise, the channel remains intact.

We use ModelSim [24] to measure the execution time and to verify functional correctness. We report the post-place-and-route area and frequency results using Vivado (v2019.1) [31]; we target a Xilinx Kintex-7 FPGA with a target clock period of 6 ns. We use Gurobi (v10.0.3) [14] to solve the LP problems from Sections 4 and 5. We use the nuXmv model checker [7] to perform the verification tasks. Each verification run is timed out after 16 hours; the concluded properties before timeout are used to simplify the circuit. We performed all experiments on AMD Ryzen 7 CPUs at 1.90 GHz and measured the optimization and verification runtime. We use the same LP solver configuration for comparison with prior strategies, i.e., only occupancy balancing [20].

For comparison with statically scheduled circuits, we generate pipelined circuits using Vivado HLS (v2019.1) [30]. For fairness, we employ the same *arithmetic units* (e.g., floating point arithmetic) used in the circuits produced from Vivado HLS. Since our technique does not address resource sharing of these units, we direct Vivado HLS in the same way by indicating that it should not prioritize this optimization. We note that, despite our efforts to make a fair comparison, the differences in the critical path optimization of the two HLS flows may cause discrepancies in register insertion and, consequently, II; this effect is entirely accidental and not caused by any fundamental difference between static and dynamic HLS. We later will observe and discuss this effect.

Benchmark	Units	Channels	Loop nests	CFCs	Property
fir	43	59	1	1	regular
iir	58	83	1	1	loop-carried dep.
matvec	77	108	1	2	regular
if loop	46	64	1	1	conditional exec.
gsum	80	108	1	2	conditional exec.
gsumif	130	175	1	3	conditional exec.
2mm	303	429	2	6	regular
3mm	376	525	3	9	regular

Table 3: Benchmark characteristics: total numbers of units and channels (prior to buffer placement), number of loop nests, choice-free circuits, and the main property of the loops they contain.

8.2 Benchmark

Table 3 reports the properties and sizes of our benchmarks; they are a collection of HLS kernels for evaluating dynamic scheduling in HLS [8, 17]: (1) *if loop* (the example in Figure 4), *gsum*, and *gsumif* contain irregular and unpredictable conditional execution that prevents standard HLS from achieving high throughput; they have been used to showcase the benefit of dynamic scheduling. (2) *fir*, *iir*, *matvec*, *2mm*, and *3mm* are from standard HLS test suites [26].

8.3 Comparison of Dataflow Circuit Optimization Strategies

Table 4 details our main results. The column *Technique* indicates whether latency, occupancy, or both are used to determine buffer placement: (1) **Occupancy** indicates that the circuit is optimized

		Simplification		FPGA Resources				Performance			Optimization Runtime			
Bench-mark	**Technique**	**Simpli-fied**	**#Stall-ed Ch.**	**Slices (red.)**	**LUTs (red.)**	**FFs (red.)**	**DSPs**	**CP (ns)**	**Cycles**	**Exec. time (us)**	**Check time (s)**	**MILP time (s)**	**LP1 time (s)**	**LP2 time (s)**
fir	**Occupancy**	✗	-	162	453	413	3	4.8	1011	4.9	-	0.04	-	-
	Occupancy	✓	3	47 (-71%)	167 (-63%)	111 (-73%)	3	4.9	1011	5.0	2	0.03	-	-
	Latency	✓	0	40 (-75%)	118 (-74%)	73 (-82%)	3	4.8	1010	4.8	2	-	0.05	-
	All	✓	0	38 (-77%)	105 (-77%)	73 (-82%)	3	4.8	1010	4.8	2	-	0.05	0.01
iir	**Occupancy**	✗	-	251	626	643	6	5.0	3007	15.0	-	0.12	-	-
	Occupancy	✓	34	195 (-22%)	505 (-19%)	427 (-34%)	6	4.8	3007	14.4	5	0.13	-	-
	Latency	✓	0	81 (-68%)	198 (-68%)	249 (-61%)	6	4.8	2009	9.6	11	-	0.2	-
	All	✓	0	93 (-63%)	227 (-64%)	235 (-63%)	6	4.8	2009	9.6	11	-	0.19	0.02
matvec	**Occupancy**	✗	-	212	611	444	3	5.2	1092	5.7	-	0.4	-	-
	Occupancy	✓	26	132 (-38%)	378 (-38%)	241 (-46%)	3	4.8	1092	5.2	40	0.42	-	-
	Latency	✓	0	59 (-72%)	167 (-73%)	120 (-73%)	3	4.8	1029	4.9	12	-	0.2	-
	All	✓	0	62 (-71%)	177 (-71%)	107 (-76%)	3	4.8	1029	4.9	10	-	0.19	0.09
if loop	**Occupancy**	✗	-	335	946	1125	4	5.5	1405	7.7	-	0.06	-	-
	Occupancy	✓	23	297 (-11%)	799 (-16%)	967 (-14%)	4	5.1	1405	7.2	11	0.05	-	-
	Latency	✓	17	230 (-31%)	622 (-34%)	752 (-33%)	4	5.0	4558	22.8	4	-	0.08	-
	All	✓	18	281 (-16%)	774 (-18%)	922 (-18%)	4	5.7	1065	6.1	7	-	0.07	0.01
gsum	**Occupancy**	✗	-	780	2278	2575	22	6.0	3232	19.4	-	0.31	-	-
	Occupancy	✓	60	705 (-10%)	1965 (-14%)	2267 (-12%)	22	5.6	3232	18.1	47155	0.32	-	-
	Latency	✓	0	1125 (+44%)	1352 (-41%)	4526 (+76%)	22	4.9	3459	16.9	169	-	0.16	-
	All	✓	0	693 (-11%)	1419 (-38%)	1972 (-23%)	22	5.0	3459	17.3	98	-	0.16	0.23
gsumif	**Occupancy**	✗	-	1056	2955	3404	26	6.2	3181	19.7	-	3.79	-	-
	Occupancy	✓	-	1056 (+0%)	2955 (+0%)	3404 (+0%)	26	6.2	3181	19.7	Timeout	3.77	-	-
	Latency	✓	0	1099 (+4%)	1774 (-40%)	4408 (+28%)	26	5.0	3327	16.6	645	-	0.68	-
	All	✓	0	841 (-20%)	1812 (-39%)	2402 (-28%)	26	5.3	3327	17.6	284	-	0.65	3.9
2mm	**Occupancy**	✗	-	1052	2743	2588	12	5.5	2451	13.5	-	206.19	-	-
	Occupancy	✓	44	531 (-50%)	1308 (-52%)	1094 (-57%)	12	4.8	2451	11.8	1546	199.85	-	-
	Latency	✓	0	351 (-67%)	679 (-75%)	776 (-70%)	12	4.8	2410	11.6	1560	-	11.65	-
	All	✓	0	315 (-70%)	736 (-73%)	645 (-75%)	12	4.9	2410	11.8	1292	-	13.29	10.75
3mm	**Occupancy**	✗	-	931	2390	1920	9	5.4	3372	18.2	-	108.16	-	-
	Occupancy	✓	28	459 (-51%)	1061 (-56%)	748 (-61%)	9	4.8	3372	16.2	827	106.58	-	-
	Latency	✓	0	305 (-67%)	645 (-73%)	509 (-73%)	9	4.8	3938	18.9	1242	-	1.77	-
	All	✓	0	315 (-66%)	690 (-71%)	445 (-77%)	9	4.8	3938	18.9	1021	-	1.9	7.01

Table 4: Resource utilization and performance of dataflow circuits using *our optimization strategy* (All (✓)), compared to circuits with no resource optimization (Occupancy (✗)), as well as circuits optimized exclusively by occupancy balancing (Occupancy (✓)) and latency balancing (Latency (✓)). In all benchmarks, our strategy significantly reduces resource utilization compared with Occupancy (✗). Since our technique also reduces the reachable state-space by regularizing the behavior of the circuit, the model checking runtime of All (✓) is significantly reduced in larger benchmarks (the timeout benchmarks did not return a single proven property). Similarly, the runtime of the two LPs we employ scales better than the MILP approach of Occupancy (✓).

using the methodology of Josipović et al. [20] to generate occupancy-balanced circuits (see Section 3.2). (2) **Latency** indicates that latency balancing is applied, but channel occupancy is not exploited for optimization (i.e., only Section 4). (3) **All** indicates the usage our complete workflow, summarized in Section 8.1, that exploits both latency and occupancy balancing. Column *Simplified* indicates whether model checking is used to simplify the circuit.

Effectiveness in eliminating spurious dynamism. Column *#Stalled Ch.* details the number of channels for which the model checker encounters a stall when exploring all possible circuit executions; those channels *must* maintain general handshake logic for correctness. Even for kernels with perfectly regular computation patterns (i.e., *fir, iir, matvec, 2mm*, and *3mm*), such stalls still exist and prevent the optimization. Our latency balancing strategy is effective in ruling out such situations and consistently removes all stalls, as seen from the entries with **Latency** (✓) or **All** (✓). The only exception is *if loop*: as discussed in Section 7, where latency balancing cannot be achieved without an II penalty; our strategy successfully identified this and the **All** (✓) solution maintained the necessary stalls and logic to respect this II goal. We will later investigate different area-performance tradeoffs on this benchmark.

Improvement in resources. The columns *Slices (red.)*, *LUTs (red.)*, *FFs (red.)*, and *DSPs* detail the FPGA resources; the improvement compared with a fully dynamic, performance-optimized solution (i.e., **Occupancy** (✗)) is recorded in parenthesis. One can notice consistent resource-saving trends of our approach across the table, for benchmarks of different sizes. Stall reduction consistently enables more logic optimization, as evident from the **Latency** (✓) and **All** (✓) designs, which require fewer LUTs compared with **Latency** (✓). When the circuit cannot be ideally pipelined (i.e., the II is limited by loop-carried dependencies or is variable due to irregular control flow), properly sizing the buffers results in significant FF savings; this is evident from the fact that *iir, gsum, gsumif*, optimized using **Latency** (✓), demand more FF than **All** (✓). This points to the need for balancing both latency and occupancy, as we propose in this paper.

Effect on performance. The *Performance* columns in Table 4 detail the clock period (*CP*), the execution time in cycles (*Cycles*), and the wall-clock time (*Exec. time*), which is calculated as *CP* × *Cycles*. As expected, the clock cycle count does not notably change across the design points since they all aim to sustain the same II and, thus, performance. In some situations, latency balancing slightly increases the cycle count despite maintaining the same II as the other solutions; this is because of the additional sequential delays that postpone the execution of some operations. The outlier is *if loop*, where **Latency** (✓) suffers in performance due to the execution variability of this benchmark; other solutions tackle it via occupancy balancing and bypass FIFO insertion (see Section 6), again pointing to the benefits of combining latency and occupancy in circuit optimization.

Runtime of LP solving and logic simplification. The following columns detail the optimization runtimes: *Check time* describes the model checking runtime. *MILP time* reports the MILP runtime of **Occupancy** [20]. *LP1 time* and *LP2 time* correspond to the LP runtime of our latency balancing problem (in Section 4) and occupancy balancing problem (in Section 5), respectively. By having

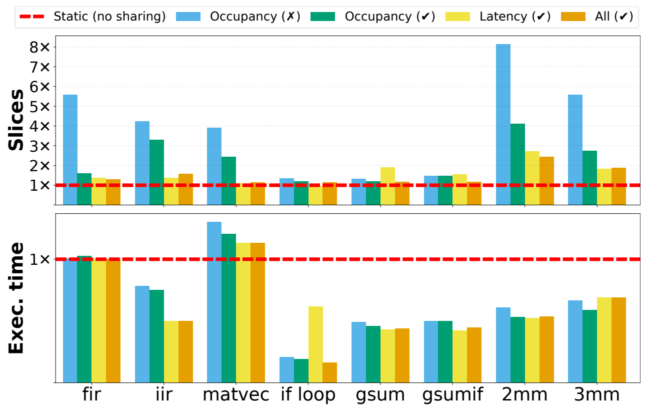


Figure 6: Resources (slices) and execution time (CP × cycles) of dataflow circuits optimized with different techniques, normalized to the corresponding static HLS designs. Compared with static HLS designs, our solutions are systematically Pareto-optimal or negligibly different from static HLS solutions.

two simpler LP formulations, the calculations of channel latency and occupancy of all benchmarks are completed within 30 seconds; this is in strong contrast with the MILP runtime of **Occupancy**, which rapidly increases for larger circuits (e.g., 2mm and 3mm). Additionally, the model checker can better handle circuits after removing unnecessary transient behaviors. This is clear from the *Check time* improvements of **All** (✓), which are especially drastic for larger benchmarks with more variability (*gsum, gsumif*). Thus, our strategy not only makes circuit optimization more effective but also more scalable than prior optimization strategies.

8.4 Comparison with Standard HLS

In this section, we compare our solutions with the standard-HLS-produced circuits.

Where do we stand? Figure 6 illustrates the total resources (in FPGA slices) and the execution time of the produced circuits; each metric is normalized with respect to the corresponding Vivado-HLS-produced designs, shown in red-dashed lines.

While unoptimized and partially optimized solutions exhibit significant resource overheads with respect to static HLS, our final solutions (**All** (✓)) typically have a negligible resource difference, thus pointing to the successfulness of our strategy.

The execution time of our circuits is similar to or lower than that of static HLS. The only increase is in *matvec* due to a difference in the circuit's critical path. The reductions in the other benchmarks are due to different reasons: (1) *iir* is, conceptually, equally pipelineable by static and dynamic HLS, but Vivado HLS's timing model introduces extra registers and, thus, increases II (this effect is orthogonal to our strategy, as mentioned in Section 8.1). (2) *if loop*, *gsum*, and *gsumif* are cases with irregular behaviors where dynamic scheduling fundamentally outperforms static (see Section 7); we now reap this gain at negligible resource cost. Interestingly, in *gsum* and *gsumif*, the benefits of dynamism remain even after removing all stalls. These benchmarks contain a conditional long-latency loop-carried dependency that prevents static pipelining; our circuits take either the short or the long path depending on

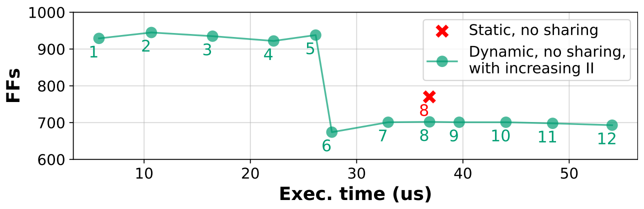


Figure 7: Exploring the design space by relaxing the II bound. Our method successfully produced two Pareto-optimal points that were previously achieved only by fundamentally different HLS strategies.

the condition value. Since we can balance each of these paths individually, tokens are always propagated stall-free. (3) Our *2mm* and *3mm* solutions overlap inner and outer loops more effectively than static HLS, which reduces execution time; they are larger than the static HLS solution due to a conservative IR generation strategy—an aspect orthogonal to our work and tackled by others [12]. We note that our solutions are systematically Pareto optimal.

Area-performance trade-off. So far, we have always aimed for maximal performance; we now evaluate the ability of our approach to explore the design space, as discussed in Section 7. We perform this experiment on the *if loop* of Figure 4 and iteratively solve the latency balancing LP for IIs of 1 to 12. Our results are plotted as the green curve in Figure 7, labeled with the target II; a red cross denotes the best II point generated by static HLS. For II=1, our circuit is the fastest but needs handshake logic due to imbalanced patterns. Increasing the II, initially, does not yield significant changes, as patterns remain imbalanced; it is at II=6 that the two patterns finally match in latencies and all handshake logic is removed, causing a drop in area. Naturally, increasing the II beyond this point is futile: the patterns remain balanced but at no benefit. This demonstrates that our method not only produces a Pareto-optimal solution with the best performance, but also the one with the best area which is the same as that of static scheduling.

9 CONCLUSION

Dataflow circuits have often been criticized for their area expensiveness, as well as the complexity of the underlying area and performance optimization algorithms; due to the former, the results are subpar to standard HLS solutions, whereas the latter makes the HLS process unacceptably slow. To tackle these challenges, we exploit the benefits of two well-known performance optimization strategies—latency and occupancy balancing. We equip the circuit with additional latencies that synchronize data transfers and identify opportunities to remove redundant handshake logic. By analyzing the time particular data items reside in buffers, we reduce the buffer count and simplify their implementation. The resulting circuits are simpler and the time to realize them is shorter than what previous optimization approaches could achieve. Furthermore, by simply tuning the performance goal, our strategy can easily explore a variety of Pareto-optimal HLS solutions. All these benefits make HLS of dataflow circuits more attractive and practical.

ACKNOWLEDGEMENTS

This work has been supported by the Swiss National Science Foundation (grant number 215747) and the ETH Future Computing Laboratory (donation from Huawei Technologies).

REFERENCES

[1] Peter A Beerel, Andrew Lines, Mike Davies, and Nam-Hoon Kim. 2006. Slack matching asynchronous designs. In *12th IEEE International Symposium on Asynchronous Circuits and Systems*. Grenoble, 184–94.

[2] Stephen P. Bradley, Arnoldo C. Hax, and Thomas L. Magnanti. 1977. *Applied Mathematical Programming*. Addison-Wesley Publishing Company.

[3] 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*. Austin, TX, 177–86.

[4] 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*. San Jose, CA, 362–69.

[5] 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), 1–27.

[6] 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* 20, 9 (Sept. 2001), 1059–76.

[7] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Proceedings of the 26th International Conference on Computer Aided Verification*. Vienna, Austria, 334–42.

[8] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, 288–98.

[9] Jianyi Cheng, Join Wickerson, and George A. Constantinides. 2022. Finding and Finessing Static Islands in Dynamically Scheduled Circuits. In *Proceedings of the 30th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Virtual Event, CA, 89–100.

[10] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. Synthesis of Synchronous Elastic Architectures. In *Proceedings of the 43rd Design Automation Conference*. San Francisco, CA, 657–62.

[11] 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*. Vienna, 175–84.

[12] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2022. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 253–61.

[13] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *Proceedings of the 29th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Virtual Event, 81–92.

[14] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[15] Abhishek Kumar Jain, Douglas L. Maskell, and Suhaib A. Fahmy. 2016. Throughput oriented FPGA overlays using DSP blocks. In *Proceedings of the 2016 Design, Automation and Test in Europe Conference and Exhibition*. Dresden, Germany, 1628–33.

[16] 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), 1–19.

[17] 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*. Monterey, CA, 127–36.

[18] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2020. Dynamatic: From C/C++ to dynamically scheduled circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, 1–10.

[19] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2021. Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits. *IEEE Circuits and Systems Magazine* 21, 1 (May 2021), 97–118.

[20] 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*. Seaside, CA, 186–96.

[21] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2022. From C/C++ Code to High-Performance Dataflow Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 7 (July 2022), 2142–55.

[22] Xinheng Liu, Dae Hee Kim, Chang Wu, and Deming Chen. 2018. Resource and Data Optimization for Hardware Implementation of Deep Neural Networks Targeting FPGA-based Edge Devices. In *Proceedings of the 20th International Workshop on System Level Interconnect Prediction*. San Francisco, CA, 1–8.

[23] 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*. London, 272–85.

[24] Mentor Graphics. 2016. ModelSim. https://www.mentor.com/products/fv/modelsim/

[25] 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*. San Jose, CA, 219–25.

[26] Louis-Noël Pouchet. 2012. *Polybench: The polyhedral benchmark suite*. https://web.cs.ucla.edu/~pouchet/software/polybench/

[27] Carmine Rizzi, Andrea Guerrieri, Paolo Ienne, and Lana Josipović. 2022. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 375–83.

[28] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis. In *Proceedings of the 33rd International Conference on Field-Programmable Logic and Applications*. Gothenburg, Sweden, 1–9.

[29] Girish Venkataramani and Seth C. Goldstein. 2006. Leveraging protocol knowledge in slack matching. In *Proceedings of the 25th International Conference on Computer-Aided Design*. San Jose, CA, 724–29.

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

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

[32] Xilinx Inc. 2023. *Vitis HLS*. Xilinx Inc. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[33] Jiahui Xu. 2023. Research Artifact for FPGA '24: Suppressing Spurious Dynamism of Dataflow Circuits via Latency and Occupancy Balancing. https://doi.org/10.5281/zenodo.10307409

[34] Jiahui Xu and Lana Josipović. 2023. Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification. In *Proceedings of the 42nd International Conference on Computer-Aided Design*. San Francisco, CA, 1–9.

[35] Jiahui Xu, Emmet Murphy, Jordi Cortadella, and Lana Josipović. 2023. Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking. In *Proceedings of the 31st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 27–37.

[36] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *Proceedings of the 32nd International Conference on Computer-Aided Design*. San Jose, CA, 211–18.