Resource Sharing in Dataflow Circuits

LANA JOSIPOVIĆ, ETH Zurich, Zurich, Switzerland
AXEL MARMET, ANDREA GUERRIERI, and PAOLO IENNE, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

To achieve resource-efficient hardware designs, **high-level synthesis (HLS)** tools share (i.e., time-multiplex) functional units among operations of the same type. This optimization is typically performed in conjunction with operation scheduling to ensure the best possible unit usage at each point in time. Dataflow circuits have emerged as an alternative HLS approach to efficiently handle irregular and control-dominated code. However, these circuits do not have a predetermined schedule—in its absence, it is challenging to determine which operations can share a functional unit without a performance penalty. More critically, although sharing seems to imply only some trivial circuitry, time-multiplexing units in dataflow circuits may cause deadlock by blocking certain data transfers and preventing operations from executing. In this paper, we present a technique to automatically identify performance-acceptable resource sharing opportunities in dataflow circuits. More importantly, we describe a sharing mechanism which achieves functionally correct and deadlock-free dataflow designs. On a set of benchmarks obtained from C code, we show that our approach effectively implements resource sharing. It results in significant area savings at a minor performance penalty compared to dataflow circuits which do not support this feature (i.e., it achieves a 64%, 2%, and 18% average reduction in DSPs, LUTs, and FFs, respectively, with an average increase in total execution time of only 2%) and matches the sharing capabilities of a state-of-the-art HLS tool.

CCS Concepts: • **Hardware** → **High-level and register-transfer level synthesis**; **Resource binding and sharing**; *Datapath optimization*; • **Computer systems organization** → Data flow architectures;

Additional Key Words and Phrases: Dataflow circuits, high-level synthesis, resource sharing

ACM Reference format:
Lana Josipović, Axel Marmet, Andrea Guerrieri, and Paolo Ienne. 2023. Resource Sharing in Dataflow Circuits. *ACM Trans. Reconfig. Technol. Syst.* 16, 4, Article 54 (September 2023), 27 pages.
https://doi.org/10.1145/3597614

1 INTRODUCTION

Current industrial and academic HLS approaches [7, 36] rely on static scheduling: at compile time, these tools decide the clock cycles in which each operation will execute and, at the same time, determine the number of functional units to allocate. The goal is to obtain the best possible schedule while reducing the resource requirements by sharing functional units between operations which are used in different clock cycles [6, 30, 38].

Authors' addresses: L. Josipović, ETH Zurich, Department of Information Technology and Electrical Engineering, Gloriastrasse 35, 8092 Zurich, Switzerland; A. Marmet, A. Guerrieri, and P. Ienne, École polytechnique fédérale de Lausanne, School of Computer and Communication Sciences, Rte Cantonale, 1015 Lausanne, Switzerland.

1936-7406/2023/09-ART54 $15.00
https://doi.org/10.1145/3597614

In contrast to standard HLS tools, dataflow or latency-insensitive protocols [8, 11, 13, 34] implement dynamically scheduled circuits, in which components exchange data as soon as all conditions for execution are satisfied. Due to this ability to adapt the schedule at runtime to particular data and control outcomes, dataflow circuits have recently been explored as an efficient HLS approach to handle irregular and control-dominated applications [18]. However, the scheduling flexibility of dataflow circuits makes resource sharing challenging: in the absence of a predetermined schedule, the cycle in which each operation executes is unknown. Hence, dataflow approaches typically employ an individual unit for each operation and result in area-expensive solutions.

The intuition on how to implement sharing in a dataflow context is fairly straightforward: instead of relying on cycle information on operation execution, one could consider statistical information on unit utilization—if a certain unit is, on average, underutilized (i.e., not always busy computing), it may be possible to share it with another underutilized unit. However, on its own, this strategy does not consider two crucial concerns: (1) Sharing may compromise some of the fundamental functional properties of dataflow circuits; one needs to ensure that the resulting circuits are always deadlock-free. (2) Sharing may postpone the execution of some operation with respect to its execution in the original dataflow circuit and, consequently, compromise performance; one needs to evaluate and minimize this performance impact.

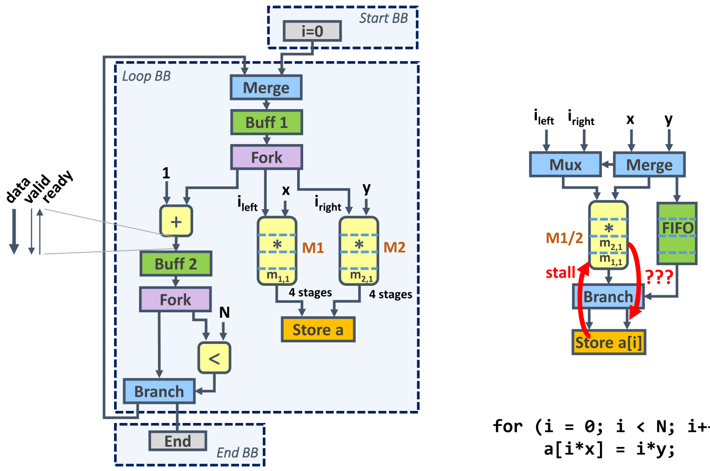
In this paper, we present a complete methodology to implement resource sharing in dataflow circuits. In Section 2, we illustrate the difficulties of performing sharing in the absence of a predetermined schedule. In Section 3, we describe the dataflow circuits we use and provide an intuition on identifying good sharing candidates in this context. In Section 4, we formulate the necessary requirements to ensure deadlock-free circuits with sharing and implement a sharing mechanism accordingly. We then discuss how to minimize the performance impact due to sharing. Section 5 details our hardware implementation, and Section 6 presents our algorithm for sharing resources in dataflow circuits obtained from C code. Finally, we evaluate our approach in Section 7; we show that our technique results in up to 81% DSP reduction with minimal or no impact on execution time compared to dataflow circuits that do not implement sharing.

Our main purpose here is to make dataflow circuits competitive in computational resource usage to standard HLS approaches while profiting from the key advantages of dynamic scheduling. To demonstrate that we have successfully achieved this goal, we compare our circuits with statically scheduled HLS designs and show that they employ the exact same number of computational units realized in DSPs. Additionally, in benchmarks where dynamic scheduling is superior to static scheduling, dataflow circuits with sharing achieve speedups of up to 2.5× over the static solutions.

This work is an extension of our conference paper “Resource sharing in dataflow circuits” [22], presented at the 30th IEEE Symposium on Field-Programmable Custom Computing Machines, 2022. In addition to the content of the published paper, we here describe a mathematical model for dataflow circuits with sharing (Section 5.2); we employ it to analyze the throughput of dataflow circuits with sharing and to ensure that sharing does not come at a performance penalty. Additionally, we provide a mathematical description of the impact of sharing on **initiation interval (II)** and elaborate on the sharing of different types of resources. We extend our evaluation section with a detailed analysis of our sharing structures and their overheads. Our resource sharing methodology is fully integrated into Dynamatic, our open-source HLS compiler, and available together with our benchmarks at dynamatic.epfl.ch.

2 MOTIVATION

To illustrate the challenges of resource sharing in dataflow circuits, consider the example in Figure 1. One should observe that this circuit has no centralized controller—instead, all dataflow units are connected to their predecessor and successor units with handshake signals that regulate



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

Fig. 1. Dataflow circuit and a possible resource sharing implementation. The multiplications could be computed using a single multiplier, with input and output multiplexing logic. Yet, this mechanism on its own does not guarantee that the circuit is deadlock-free nor that its performance is optimal. The multiplication results are indicated as $m_{op,iter}$ (e.g., $m_{2,1}$ is the result of operation M2 from iteration 1).

the flow of data (i.e., tokens); each operation executes as soon as its inputs become available and the corresponding unit is ready. The execution starts when a token enters through the starting point (i.e., Start BB, with iterator $i = 0$); a new loop iteration is triggered as soon as a token reenters the loop body through the cyclic path (in this example, every second clock cycle because of two registers, Buff 1 and Buff 2, on the cyclic path through the merge and the branch). The loop in the figure contains two pipelined, 4-stage multipliers; all other units, apart from the buffers, are combinational. Since a new loop iteration starts every second cycle, the two multiplications could be performed using a single multiplier. An intuitive implementation is shown on the right—the merge and mux steer one set of input tokens at a time into the shared unit (as the figure indicates, these units must communicate to ensure that they always accept the matching operands from their predecessors). The branch at the output ensures that the result is sent to the appropriate successor, depending on the origin of the operand tokens—this information is conveyed to the branch by the input merge through a FIFO.

Surprisingly, this implementation does not guarantee a functional circuit: in this example, the store needs both operands (i.e., both the address and the data) to execute; it therefore stalls the available operand ($m_{1,1}$, i.e., the result of M1 of the first iteration) while it waits for the second operand ($m_{2,1}$, i.e., the result of M2). However, because of the stall of $m_{1,1}$, $m_{2,1}$ will never be able to exit the shared unit and arrive to the store, therefore causing deadlock. Such problems are absent by construction in elementary dataflow circuits [18] where each operation uses an individual unit and only a single token per loop iteration is transferred from one unit to another. However, introducing sharing compromises this property; it is crucial that we develop a sharing mechanism that handles this issue and ensures the absence of deadlock in every possible case.

3 BACKGROUND

In this section, we describe dataflow circuits and illustrate why classic sharing techniques are not applicable in this context. Finally, we discuss how existing performance analysis techniques can be used to identify sharing opportunities in dataflow designs.

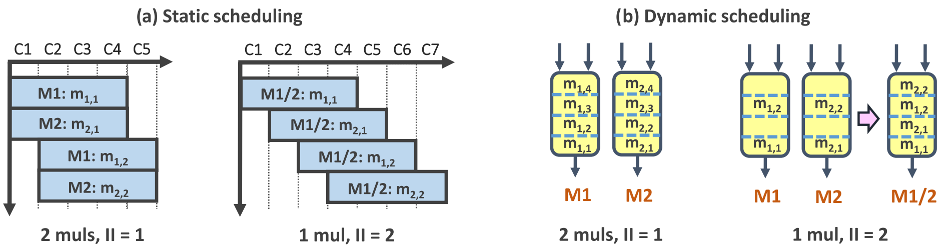


Fig. 2. Resource sharing in static and dynamic scheduling. In static scheduling, resource sharing is decided based on the cycle information on operation execution; in contrast, dataflow circuits can rely on average unit utilization to identify good sharing candidates.

3.1 Dataflow Circuits

Dataflow circuits, also referred to as elastic, latency-insensitive, or data-driven circuits, are built out of units that exchange data (i.e., tokens) using a handshake mechanism [8, 11, 13, 34]. Several authors [3, 12, 18, 25, 31, 32] generate dataflow circuits from high-level programs; we follow an approach that produces *synchronous* dataflow circuits from C code [18]. The circuits we consider organize units into **basic blocks (BBs)**, i.e., straight pieces of code with no conditionals; once a BB is triggered, all its units are guaranteed to receive all their input data and each dataflow edge between the units performs a single token transfer. Control flow statements are implemented between the BBs to form a **control flow graph (CFG)**.

We use the following dataflow units that communicate using a standard handshake protocol [11]: (1) a *merge* sends a token nondeterministically into a BB from one of the predecessor BBs, (2) a *mux* is a deterministic version of a merge with an additional control input to select the input token, (3) a *branch* sends the token to one of the successor BBs, as determined by the BB condition, (4) a *fork* distributes a copy of a token to each of its successors, either simultaneously (*lazy fork*), or whenever they are ready to accept it (*eager fork*), and (5) a *join* synchronizes multiple tokens (e.g., operation operands) before triggering the successor. *Buffers* are used to store data; they are characterized by their capacity (i.e., the number of tokens a buffer can hold) and transparency (indicating whether a buffer adds sequential delay, or is a pass-through element) [23]. To ensure that a circuit is deadlock-free, each cyclic path must always have at least one empty buffer slot; additional buffers may be arbitrarily added without compromising correctness [11, 18], but only with impact on performance.

3.2 Deciding What to Share in a Dataflow Circuit

Standard, statically scheduled HLS tools [7, 36] perform scheduling in conjunction with resource allocation and sharing [38]; depending on the optimization objective, they trade-off area and performance by deciding the cycle in which each operation executes and allocating units accordingly. Figure 2(a) shows two possible schedules for the code from Figure 1: the first achieves the ideal loop pipeline with an II of 1 by scheduling both multiplications in the same cycle, hence employing two multipliers; the second increases the II to 2 and schedules each multiplication on every second cycle, which allows the usage of a single multiplier. It is important to note that the compile-time scheduling dictates the execution time of each operation: by scheduling the two multiplications to start in two consecutive cycles, it enables them to share a single multiplier. Dataflow circuits face the same optimization objectives and area-performance trade-offs; however, there is no predetermined schedule: the exact execution time of each operation is unknown and will only be determined at runtime, during dataflow circuit execution. Thus, one cannot rely on scheduling information to decide how many units to employ. Instead, one can reason about the average computational rate, identify units which are, on average, underutilized, and use this information to implement sharing.

Several authors discussed techniques to analyze the timing of dataflow circuits [4, 5, 23, 28, 29]; some determine the rate at which dataflow units compute and directly provide the information on average unit utilization. We here rely on an approach that maximizes the throughput (i.e., the inverse of the initiation interval, 1/II) of each CFG cycle by appropriately placing and sizing buffers [23]. The approach calculates the average *occupancy* of each unit with tokens, i.e., for a given throughput of a CFG cycle, determines the average number of tokens that each unit holds in the steady state of the cycle execution. We can use this information to identify good candidates for sharing [20]: if the sum of the tokens in two units of the same type is at most equal to the unit latency (i.e., number of sequential stages), it may be possible to use a single unit without damaging the throughput of the CFG cycle.

In the dataflow circuit in Figure 1, the cyclic path contains two buffers, so a new token enters the loop on every second cycle, i.e., the throughput is 1/2. Thus, a new token enters each multiplier on every second cycle as well; in the steady state, each multiplier holds two tokens and has two empty slots (i.e., the occupancy of each multiplier is equal to 2), as shown on the left of Figure 2(b). It is therefore possible to implement the two multiplications using a single multiplier that will accept a new token and start a new multiplication on every cycle—this multiplier will, in the steady state, always be busy computing and its occupancy will be equal to 4 (see right of Figure 2(b)).

Although such analysis ensures that each shared unit receives tokens at a rate at which it can compute, it does not recognize that sharing may postpone a computation: In Figure 1, prior to sharing, both multiplications execute simultaneously (i.e., $m_{1,1}$ and $m_{2,1}$ are computed at the same time by the two multipliers); with sharing, one multiplication is delayed by one clock cycle (in the right of Figure 2(b), $m_{2,1}$ is computed one cycle after $m_{1,1}$). In some cases, such delays may compromise throughput, as we will discuss later. More importantly, as indicated in Section 2, nothing in this analysis guarantees that the dataflow circuit with sharing is deadlock-free. We will address both of these issues in this paper.

4 RESOURCE SHARING IN DATAFLOW CIRCUITS

This section details our methodology for deadlock-free and high-performance resource sharing in dataflow circuits.

4.1 Sharing in Noncyclic Datapaths

Sharing requires steering data into a unit from multiple predecessors and sending the output to the appropriate successor. This behavior is realized on the left of Figure 3, repeating the situation of Figure 1: the input of the shared unit has a merge for one of its operands and a mux for all others; they have as many data inputs as there are shared operations. The merge informs the muxes and the branch which operand it took so that they can choose the corresponding operands and send the result to the correct successor, respectively. The merge and the branch communicate through a FIFO, with as many slots as there are pipeline stages in the unit. Yet, as discussed before, this scheme does not guarantee a functional circuit: a token may be stalled inside the unit and prevent the others from exiting, potentially causing deadlock. In this case, as the successor unit needs to join tokens $m_{1,1}$ and $m_{2,1}$ to compute, $m_{1,1}$ cannot exit the unit until $m_{2,1}$ arrives; however, the exact same token ($m_{1,1}$) is blocking $m_{2,1}$ from ever exiting the unit, therefore infinitely starving the succeeding store and blocking the shared multiplier from processing other tokens.

The mechanism on the right of Figure 3 guarantees that all tokens from a noncyclic datapath (i.e., a single BB or a sequence of nonrepeating BBs) that enter the shared unit are able to exit

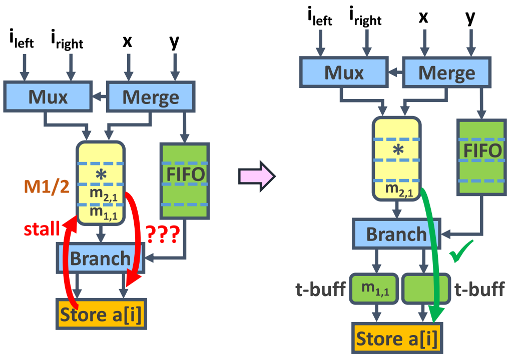


Fig. 3. Hardware for sharing. A naive implementation (left) results in deadlock. Placing transparent buffers (t-buffs) on the branch outputs (right) allows all tokens from a single datapath execution to exit the shared unit and all the computation of the datapath to complete.

it by adding a 1-slot transparent buffer (see Section 3.1), i.e., *t-buff*, at each branch output. In a single BB execution, each dataflow edge transfers a single token; the output edge of the shared unit, on the other hand, does not honor this property (i.e., it transfers as many tokens as there are shared operations), but it sends only one token to each branch output (corresponding to each individual operation in the original circuit). Hence, the *t-buff* is sufficient to ensure that each token can always exit the unit, regardless of the availability of the successor: if the successor is not ready, the *t-buff* will store the token; otherwise, the token will immediately propagate further. No token will be stalled in the unit, nor will it block other tokens in the unit; all successor units of the same BB will be able to receive their data and all BB computation will successfully complete, exactly as if no units were shared.

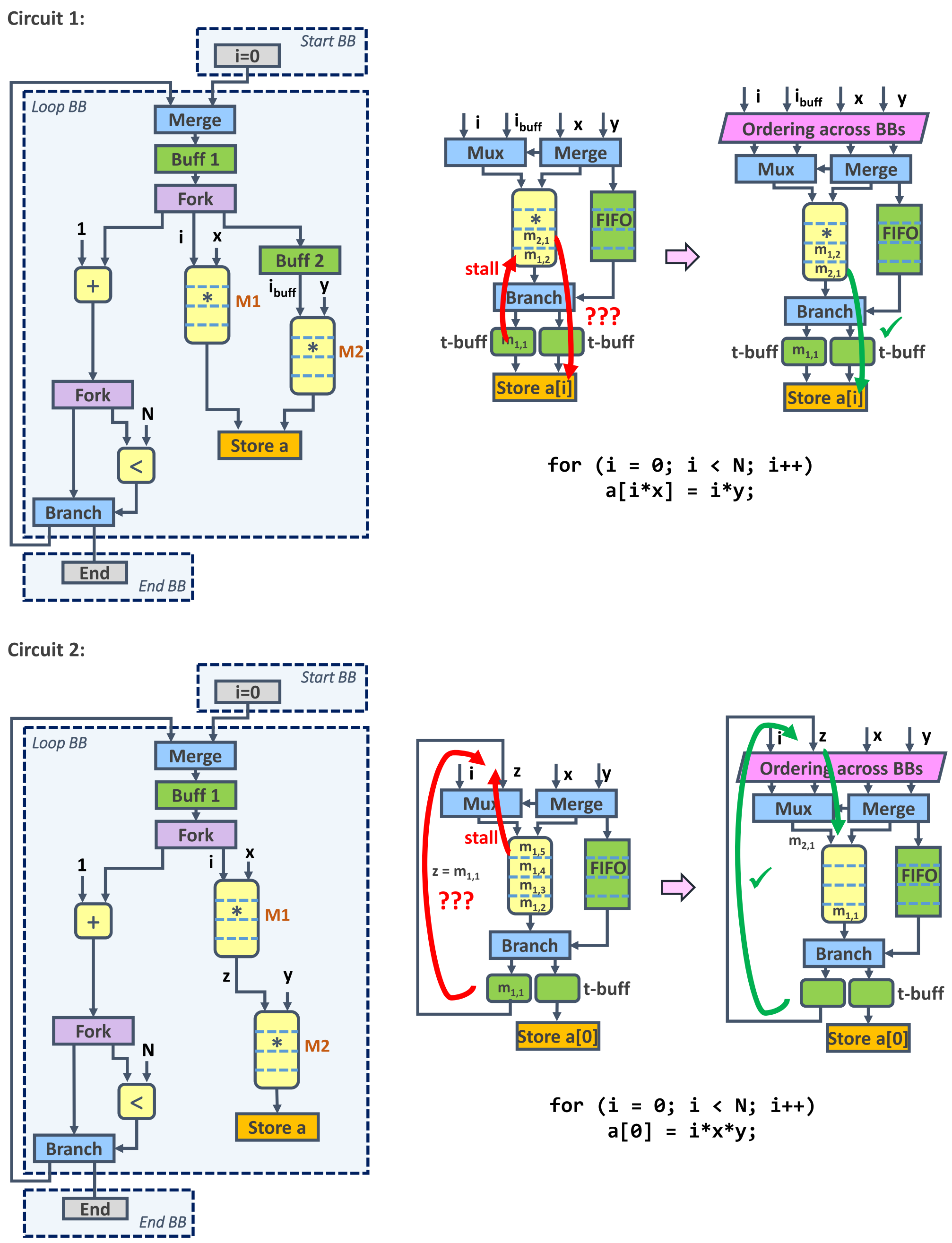
However, this hardware does not guarantee deadlock-free execution in cases where BBs repeat, as in a loop. We discuss this issue in the following section.

4.2 Sharing in General Datapaths

The methodology from the previous section guarantees that the circuit is functional only when sharing within a single BB or a loop iteration; we here extend this implementation to general programs.

Figure 4 shows two examples where the mechanism from Section 4.1 does not manage to prevent deadlock: (1) Circuit 1 has a similar problem as discussed before, but occurring across loop iterations: a token from a successive iteration ($m_{1,2}$) blocks the token from the previous iteration ($m_{2,1}$) from exiting the shared unit; at the same time, $m_{1,2}$ cannot proceed before the previous computation completes, so both tokens indefinitely stall. (2) Circuit 2 has a cycle from the output of the shared unit to its input. The unit may fill with tokens and cause deadlock because there is no empty space for the tokens to move (i.e., the property which guarantees the absence of deadlock, outlined in Section 3.1, is violated, as no buffer slot on the cycle is empty): the token in the unit ($m_{1,2}$) needs to move into *t-buff* on the cycle, but the token in the *t-buff* ($m_{1,1}$, i.e., the input z of M2) cannot move back into the unit before another token exits.

Both problems are due to tokens entering the shared unit in an order different than the one specified by the control flow of the program—some tokens enter the unit before all tokens from the preceding BBs and prevent their computations from completing: (1) In circuit 1, instead of consecutively consuming both tokens from the same BB execution, the unit inputs some tokens from the following iteration (i.e., the next BB) which prevent one of the previous tokens from ever



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

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

Fig. 4. Deadlock situations. In the shown dataflow circuits, deadlock can occur due to the reordering of tokens from different BB executions. In circuit 1, the token from the second loop iteration ($m_{1,2}$) precedes the token from the first iteration ($m_{2,1}$) and prevents it from ever exiting the unit. In circuit 2, tokens from multiple iterations may enter the unit before the second multiplication (M2) of the first iteration issues; the unit fills with tokens and no token can move forward. The solution in both cases is to force tokens to enter the unit in the order of BB execution, i.e., all tokens from one BB must enter the unit before the tokens from the successor BBs, as shown in the rightmost figures.

exiting the unit. (2) In circuit 2, the token from the first BB execution (i.e., first iteration) comes from the shared unit itself. Yet, instead of consuming this token to execute the first multiplication of M2, the shared unit keeps taking tokens from the following iterations (coming from the noncyclic path and performing multiplications of M1), therefore filling the unit and preventing the older token from the *t-buff* from propagating further.

The solution to both problems is to send tokens to the shared unit in the order specified by the control flow (i.e., program order): once a BB execution is decided, all its tokens must be consumed by the shared unit before the tokens from the following BB. If all tokens from a BB are injected into the unit before any successive tokens, they are guaranteed to exit the unit (see Section 4.1). Thus, always sending tokens into the unit in order of BB execution guarantees the absence of deadlock for any number of BBs and BB executions.

4.3 Sharing and Performance

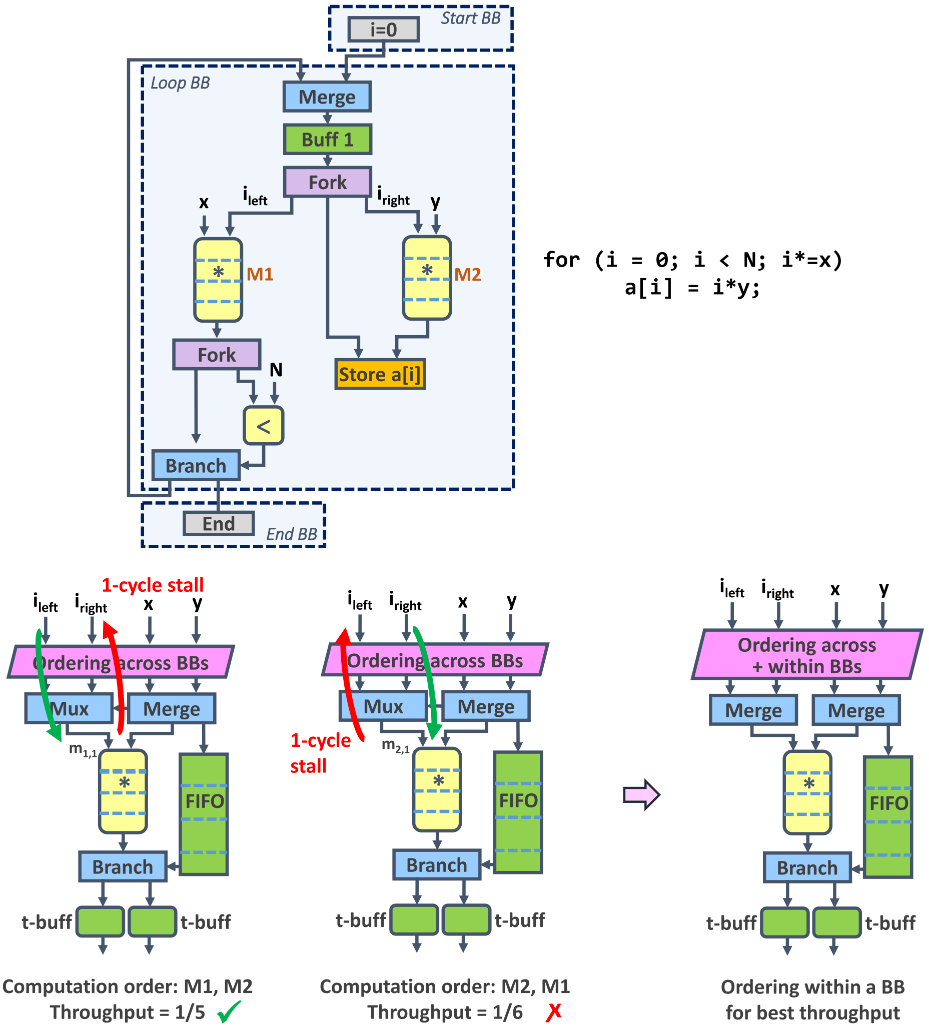
The previous section showed the need to order tokens from different BBs as they enter a shared unit to prevent deadlock. The ordering of tokens from the same BB does not compromise the circuit functioning, but may impact performance.

The buffering of the dataflow circuit needs to account for the operation delays caused by sharing. More importantly, one needs to make sure that the latency of a throughput-critical cycle is not increased. In the dataflow circuit in Figure 5, both multiplications execute simultaneously; M1 is on a loop determining the throughput, equal to 1/5 (because of the buffer and the 4-stage multiplier on the cycle). If the two multiplications share a unit, one of them will be postponed for a clock cycle while the multiplier consumes the inputs of the other. If the delayed computation is M1, the cycle latency increases and, consequently, lowers the throughput to 1/6, as shown in the bottom of the figure.

Therefore, in addition to enforcing the ordering of operations from different BBs, as previously described, one could order operations within each BB as well, as indicated in the bottom right of Figure 5, such that the throughput impact is minimal. We incorporate this notion into our sharing strategy, as we will describe in Section 6: we use the performance analysis from Section 3.2 to choose an ordering which maintains the original throughput as well as to obtain the optimal buffering that accounts for the delays caused by sharing. Note that we now implement a total order of the operations and, thus, the corresponding operands always arrive aligned to the unit; hence, the muxes at the unit inputs (see Section 4.1) can be replaced by merges. We detail our implementation of the ordering logic in Section 5.

4.4 Extending the Ordering Scheme

The ordering rules described so far ensure the absence of deadlock by ordering tokens across BBs (Section 4.2); to ensure the best possible throughput in the presence of such ordering, we order operations within a BB as well (Section 4.3). Interestingly, ordering tokens across BBs may, in particular cases, lower the throughput of a loop, as it may limit the overlapping of operations from different loop iterations. This is the case in circuit 2 in Figure 4: One could, in principle, implement sharing for M1 and M2 with a throughput of 1/2 (i.e., an II of 2) by starting one of the two multiplications on every consecutive clock cycle. However, our strategy from Section 4.2 lowers the throughput to 1/5—as suggested in the rightmost sharing implementation of circuit 2 in Figure 4, the first computation of M2 ($m_{2,1}$) starts four cycles after the start of the first computation from M1 ($m_{1,1}$); the next operation from M1 can start on the cycle after the start of M2. Concretely, our ordering enforces a cycle distance between two consecutive executions of a single operation to be greater than the number of cycles between the start of the first and the start of the last



```
for (i = 0; i < N; i*=x)
      a[i] = i*y;
```

Fig. 5. Performance impact of sharing. The order in which tokens are sent to the shared unit may impact performance by postponing the execution of a throughput-limiting computation (in this example, M1 on the cyclic path). Hence, apart from ordering tokens from different BBs as they enter a shared unit, we also enforce the ordering of tokens of the same BB such that the throughput impact is minimal.

operation within the iteration. The initiation interval of the loop with sharing, II_{shared}, can then be expressed as:

$$II_{shared} = max(II_{orig}, Dist(op_{first,i}, op_{last,i})), \tag{1}$$

where II_{orig} is the II of the loop prior to sharing, and $Dist(op_{first,i}, op_{last,i})$ the clock cycle distance between the first and the last operation of a loop iteration i; if this value is higher than II_{orig}, it will increase the II and lower performance. In fact, because of this throughput degradation, the sharing algorithm which we later introduce would not identify the operations of circuit 2 in Figure 4 as acceptable sharing candidates.

Note that our ordering condition from Section 4.2 is *sufficient* to guarantee the absence of deadlock in any dataflow circuit with the structural properties described in Section 3.1. Yet, this condition is not always *necessary*—our generic ordering mechanism could be replaced by application-specific multiplexing and buffering schemes. For instance, one could relax the ordering constraint so that particular executions from different iterations can overlap—in the example above, allowing an operation from M1 to start before the operation of M2 from the preceding iteration would lower the II. The number of overlapping iterations could be determined based on the cycle distances between operation executions and the achievable II. The buffers around the sharing logic would need to be sized to accommodate tokens from multiple iterations. The technique from Section 4.3 would then be extended to order operations among all iterations which overlap. Naturally, the search space for the appropriate ordering, the complexity of the ordering logic, and the sizes of the buffers around the shared unit would increase with the number of overlapping iterations. Without loss of generality, we limit our ordering to the rules from Sections 4.2 and 4.3. As we later demonstrate, our strategy effectively implements sharing without a throughput penalty in realistic benchmarks.

4.5 Sharing Other Resources

Although we here describe the sharing of functional units (and, as discussed later, focus on reducing their DSP count), our entire sharing strategy is general and applicable to other types of resources as well (e.g., memory blocks, LUTs, buffers). The deadlock avoidance mechanism from Section 4.2 supports any sequential resource type and can be simplified for combinational units: as tokens immediately propagate from input to output, they cannot be stalled within the unit and the transparent buffers at the unit outputs are sufficient to guarantee the absence of deadlock. The same mechanism applies to function sharing as well, as long as the function implementation ensures that tokens exit the units in the same order that they entered (and, thus, exhibit the same ordering properties as any functional unit discussed before). The performance considerations of Section 4.3 (and the accompanying algorithm that we introduce in Section 6) directly apply to any resource type.

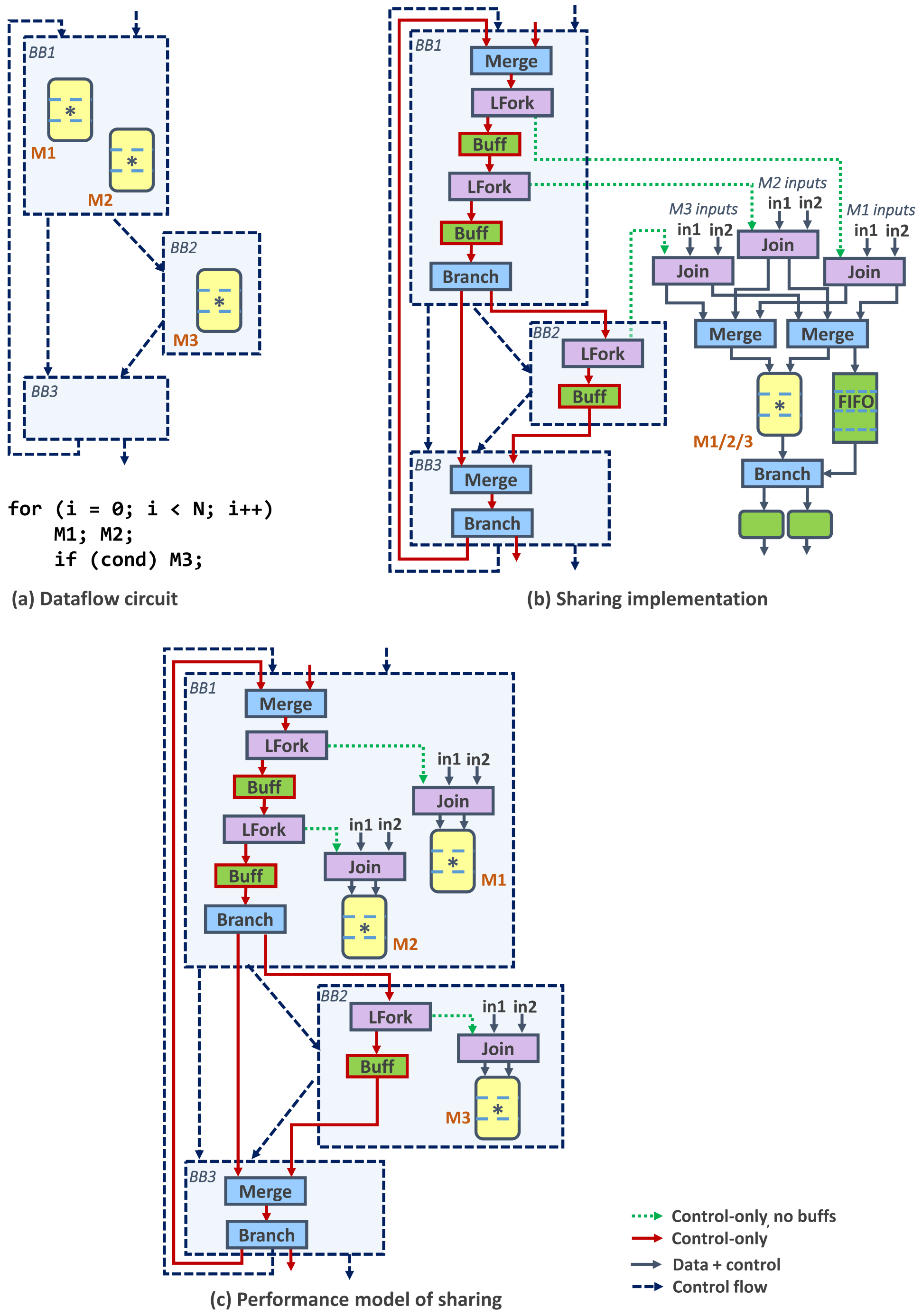
5 ORDERING IMPLEMENTATION

In this section, we detail how to implement the previously-described token ordering when sharing dataflow units.

5.1 Implementation

To implement the desired ordering between operations sharing a unit, we build an in-order dataflow network that strictly mimics the control flow of the program; it propagates a data-less token which triggers the advancement of operands to the shared unit in a predetermined order only when control flow reaches the corresponding BB. Each shared operation is associated with a lazy fork in this network; one of the fork outputs is synchronized, using a join, with the inputs of its shared operation. The fork must be lazy (see Section 3.1) so that a token moves forward and triggers the next fork only after the joined inputs have been sent to the unit. The forks are separated by buffers which introduce a 1-cycle sequential delay, i.e., two forks cannot be active at the same time. Hence, only one set of inputs to the unit will be active at any given clock cycle; this activation order corresponds to the desired operation ordering.

Figure 6(a) shows a CFG of a program with two CFG cycles; it contains three multiplications (M1 and M2 in BB1, and M3 in conditionally executed BB2) which we, in this example, aim to implement using a single multiplier. The resulting dataflow circuit is shown in Figure 6(b) (all irrelevant dataflow units are omitted from the figure for clarity). The in-order network which supplies ordering information to the unit shared between M1, M2, and M3 is shown on the left of the figure—it implements the orderings {M1, M2} in BB1 and {M3} in BB2. When the execution of BB1 starts, the first lazy fork keeps the token until both inputs of M1 become available and



```
for (i = 0; i < N; i++)
    M1; M2;
    if (cond) M3;
```

Fig. 6. Sharing implementation and model. A specialized in-order dataflow network enforces the specified ordering of operations in the shared unit. The same network is used during performance analysis to model delays introduced by sharing.

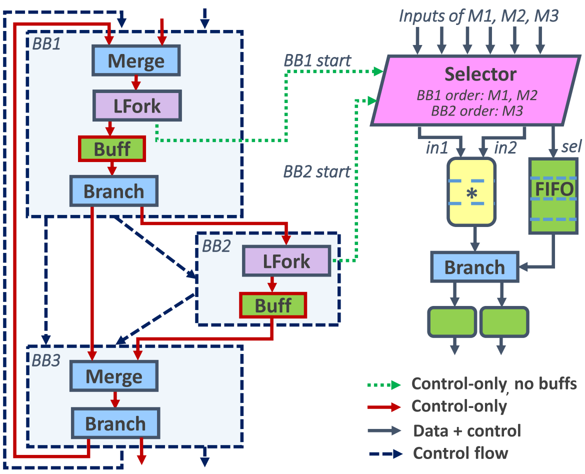


Fig. 7. Optimized sharing implementation. We use the in-order network described in Section 5.3 which, instead of sending an ordering signal per operand, sends a signal per BB; the selector uses this information to enforce a preencoded ordering of operations within each BB.

are consumed by the multiplier; only then does the token move to the next LFork through Buff 1, triggering the execution of M2 at least one clock cycle later. If the control flow decides on the execution of BB2, the in-order network will ensure that M3 executes before M1 and M2 from the next iteration of BB1. Thus, this in-order network effectively implements the functionality of the ordering unit in Figure 5.

5.2 Sharing Model for Performance Analysis

To determine whether sharing affects performance, as discussed in Section 4.3, we need to analyze the throughput achieved in each CFG cycle and compare it to the throughput achieved prior to sharing. The latter is directly obtainable for the original circuit without sharing using the performance analysis [23] mentioned in Section 3.2 which provides us with the throughput of the CFG cycles (e.g., the two cycles in the example in Figure 6). However, the circuit representation in Figure 6(b) is not directly suitable for performance analysis—determining the throughput of each CFG cycle requires every operation to be represented as an individual unit in a particular BB (and, consequently, analyzable as part of a CFG cycle); furthermore, the merge and branch units at the shared unit inputs and output are not immediately compatible with the choice-free behavior that such performance analysis requires [23].

Hence, for the performance analysis, we represent each operation *individually* in its original BB and model the effects of sharing with the in-order network described in the previous section; it connects the individual operators and describes the delays due to the enforced ordering, as shown in Figure 6(c). The performance analysis determines the throughput achievable with this circuit configuration and the corresponding delays. The comparison of the achieved throughput with that of the original circuit indicates whether the sharing and the explored ordering are desirable [16]; we include this aspect in the sharing strategy in Section 6.

5.3 Optimized Implementation

The sharing logic described above may quickly grow in complexity as each shared unit requires its own in-order network with as many lazy forks and buffers as there are shared operations; clearly, it is desirable to unify all networks. Also, as we will later mention, we implement our approach

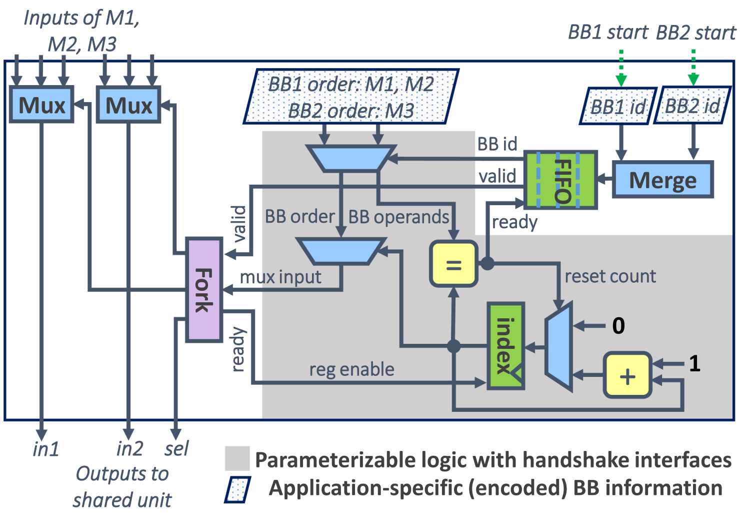


Fig. 8. Implementation of the selector unit. The internal selector logic (grey) selects the appropriate data inputs of the muxes on the left based on the order of BB execution (i.e., the *BB start* signals) and the preencoded operation order for each BB.

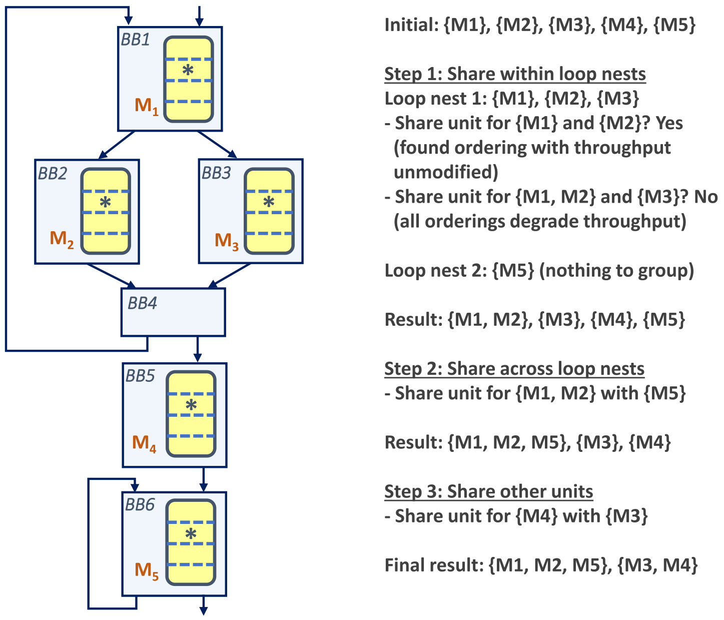
in an existing HLS framework which already produces, for other purposes, an in-order network expressing the dynamic succession of executed BBs [17, 21]. Thus, we adapt our implementation to directly leverage this existing network.

Our simplified implementation is shown in Figure 7. The network on the left of the figure is what already exists in the dataflow circuit: it emits tokens corresponding to the BB succession and, as in our original network, the use of lazy forks separated by buffers ensures that each BB start signal is triggered strictly in order. Essentially, the difference compared to Figure 6(b) is that the *selector* receives a single ordering signal per BB instead of an ordering signal per operand: thus, every time a BB starts, the selector needs to enforce the ordering of the corresponding BB operands (preencoded in the selector unit) before the operands of the subsequent BB.

5.4 Selector Unit

Figure 8 details the selector unit. It contains a FIFO which stores the IDs of the incoming BBs as they arrive in program order and one at a time from the in-order network. The *BB id* at the head of the FIFO selects the preencoded BB ordering information (i.e., a vector with the operand order, *BB order*, and the total number of operands of this BB, *BB operands*). An internal counter enables the appropriate input ports (*mux input*) of the data muxes on the left; a mux port is enabled only after the previous port has sent a token into the unit. A *BB id* is removed from the queue when all its operations have started executing, moving the successor BB to the head of the queue and allowing its tokens to enter the shared unit next.

The size of the encoded ordering information depends on the total number of shared units S, the maximal number of units within a single BB, S_{bb}, and the number of BBs connected to the selector, B: (1) *BB id* requires $\log_2(B)$ bits, (2) *BB order* requires $S_{bb} \times \log_2(S)$ bits, and (3) *BB operands* requires $max(1, \lceil \log_2(S_{bb}) \rceil)$ bits. Typically, only a few operations share a unit and these values are relatively small (e.g., in the example of Figure 8, *BB id*, *BB order*, and *BB operands*, encoded in the dotted boxes, are 1, 4, and 1 bits). The complexity of the multiplexing logic in the selector follows the same trends; it is typically minor in comparison to the 32- or 64-bit data multiplexers (left of the



Initial: {M1}, {M2}, {M3}, {M4}, {M5}

Step 1: Share within loop nests
Loop nest 1: {M1}, {M2}, {M3}
- Share unit for {M1} and {M2}? Yes (found ordering with throughput unmodified)
- Share unit for {M1, M2} and {M3}? No (all orderings degrade throughput)

Loop nest 2: {M5} (nothing to group)

Result: {M1, M2}, {M3}, {M4}, {M5}

Step 2: Share across loop nests
- Share unit for {M1, M2} with {M5}

Result: {M1, M2, M5}, {M3}, {M4}

Step 3: Share other units
- Share unit for {M4} with {M3}

Final result: {M1, M2, M5}, {M3, M4}

Fig. 9. Example execution of our sharing strategy. The execution assumes that units M1 and M3 are on throughput-critical cycles and, therefore, should not be shared.

selector in Figure 8), which are necessary in any sharing implementation and are not an overhead of our particular strategy. We will evaluate the complexity of the selector unit in Section 7.4.

6 PUTTING IT ALL TOGETHER

Algorithm 1 summarizes our resource sharing strategy. It operates on **control-data flow graphs (CDFGs)** of independent loop nests, i.e., different strongly connected components of the global CDFG. Initially, we consider every operation as a separate group (i.e., unit). Our strategy attempts to merge different groups that can share the same physical resource without compromising the throughput of any of the loops as follows:

(1) *Sharing within a loop nest.* For every pair of groups that belong to the same loop nest, we check if their sum of token occupancies (indicated as $\dot{\Theta}$) is at most equal to the unit latency $\mathbf{L}_u$ (line 14 of the algorithm); if so, the units are underutilized (see Section 3.2) and sharing may be possible without compromising performance. If neither of the groups has units on cyclic paths (lines 17–21), the original throughput Θ_s can always be maintained; thus, we topologically order the operations within each BB and employ the performance analysis from Section 3.2 to resize the buffers accordingly (i.e., to account for any operation delay due to sharing). If any of the units is on a cyclic path (lines 22–30), we use the same performance analysis to choose an ordering of operations that does not damage the throughput, i.e., where none of the operations on a throughput-critical cycle is postponed (see Section 4.3). As soon as such an ordering is found, the search terminates; the groups will be merged and the occupancy of the group will be updated (lines 32–36). If all orderings degrade throughput, the merging of the groups is discarded. This process repeats until no further merging can be done. The final ordering within each group corresponds to that found in the last successful merge and the buffer placement and sizing to that determined in the last performance analysis run.

ALGORITHM 1: Sharing strategy.

```
1  // Input: units (all units of the same type)
2  // Input: sets (dataflow units of different strongly connected components of the CDFG)
3  // Output: globalGroups (sets of operations which share a resource)
4  // 1. Sharing within a loop nest
5  forall $s \in sets$ do
6      // Calculate original throughput and buffers in set
7      $\Theta_s$, buffs = runPerformanceAnalysis (s)
8      // Initialize groups to individual units
9      groups (s) = {u | u ∈ units, u ∈ s}
10     // Grouping of units
11     while groups (s) modified do
12         forall $g_1, g_2 \in groups(s), g_1 \neq g_2$ do
13             // If token occupancy sum is at most equal to the unit latency, sharing is possible
14             if $\dot{\Theta}_{g_1} + \dot{\Theta}_{g_2} \leq L_u$ then
15                 finalOrd = null
16                 // Check if any unit on cycle
17                 if !$g_1$.hasCycle and !$g_2$.hasCycle then
18                     // No cyclic paths, sort topologically
19                     finalOrd = sort ($g_1 \cup g_2$)
20                     // Resize buffers
21                     $\Theta_{s,ord}$, buffs = runPerformanceAnalysis (s, ord)
22                 else
23                     // Search for best group ordering
24                     forall ord ∈ possible_orderings ($g_1 \cup g_2$) do
25                         // Check if throughput maintained
26                         $\Theta_{s,ord}$, buffs = runPerformanceAnalysis (s, ord)
27                         if $\Theta_{s,ord} = \Theta_s$ then
28                             // Ordering found, terminate
29                             finalOrd = ord
30                             break
31                 // Valid ordering found: share
32                 if finalOrd != null then
33                     // Merge groups and update ordering
34                     groups(s).update($g_1$, $g_2$, $g_1 \cup g_2$, finalOrd)
35                     // Update occupancy of merged group
36                     $\dot{\Theta}_{g_1 \cup g_2} = \dot{\Theta}_{g_1} + \dot{\Theta}_{g_2}$
37 // 2. Sharing across loop nests
38 globalGroups = {}
39 forall $s \in sets$ do
40     // Merge every distinct group of one loop nest with distinct groups of other nests
41     $i = 0$
42     forall group ∈ groups (s) do
43         globalGroups (i++).add (group (s))
44 // 3. Sharing other units
45 // Merge every remaining unit with any existing group
46 $i = 0$
47 forall $u \in \{u \mid u \in units, \forall s \in sets : u \notin s\}$ do
48     globalGroups (i++ mod globalGroups.size).add (u)
```

(2) *Sharing across loop nests.* In this step (lines 38–43), we merge every distinct group of one loop nest with any distinct group of another (if available and not already merged with another group from the same nest); the operation ordering in each BB remains as determined in the previous step.

(3) *Sharing other units.* We merge units that are not in any loop with any of the existing groups (lines 46–48).

The first step ensures that sharing never damages the throughput of any interconnected loops. The second step does not need throughput analysis as different loop nests execute consecutively—while the final iterations of one loop may overlap with the initial iterations of another, two operations from different loop nests never execute simultaneously in the steady state. The same holds for units that do not belong to any loop.

Figure 9 illustrates this strategy with an example. The given CFG has two loop nests; although not visible in the figure, we assume for the purpose of this example that M1 and M3 are on paths which are critical for throughput and, hence, should not be shared. The first step of the sharing algorithm (*sharing within a loop nest*) considers the two loop nests separately; it attempts to group operations while evaluating the throughput, as shown on the right of the figure. In loop nest 1, two out of three operations can be shared (i.e., according to our assumption for this example, M1 and M3 cannot share a unit as this sharing would degrade the throughput). In loop nest 2, there is only a single operation, M5, thus nothing is shared. The second step of the algorithm (*sharing across loop nests*) groups operations from different loop nests; in this example, M5 from loop nest 2 is grouped with M1 and M2. Finally, the third step (*sharing other units*) adds the operation that does not belong to any loop nest (M4) to the previously determined groups. In this example, the resulting circuit implements five operations using two shared units.

Our strategy minimizes the number of units under a throughput constraint. It is adaptable to other optimization objectives as well, e.g., honoring a resource constraint: if the constraint is tighter than the group count achieved by Algorithm 1, one could continue grouping until it is met; the associated performance penalty could be minimized by exploring different groupings. Our algorithm immediately identifies good sharing candidates (i.e., underutilized units) and performs an ordering exploration only in case of throughput-limiting operations on cycles; it is therefore effective in optimizing complex graphs with a large unit count.

7 EVALUATION

In this section, we evaluate our approach for implementing resource sharing in dataflow circuits obtained from C code.

7.1 Methodology and Benchmarks

We evaluate a selection of floating-point kernels from the PolyBench suite [27] that contain loop nests with different properties (i.e., loop organization, count, and nest levels) and computational patterns, thus offering different sharing opportunities within and across loops and loop nests, as shown in Table 1. Most kernels have long-latency loop-carried dependencies due to pipelined floating-point operations that limit the loop II. Our purpose here is not to show the superiority of dataflow circuits over statically scheduled designs but to investigate their sharing capabilities; nevertheless, we also consider two typical cases where dynamic scheduling excels over standard HLS, i.e., *gsum* and *gsumif*, that conditionally compute floating-point polynomial expressions. The conditional statements incur unpredictable long-latency, loop-carried dependencies that prevent static scheduling from achieving high-throughput pipelines; due to the low throughput, the static solutions can share floating-point units among the conditionally executed operations [9].

We implement our sharing strategy in Dynamatic, an open-source HLS tool [19] that synthesizes C code into synchronous dataflow designs and implements the performance analysis from Section 3.2. Although our sharing technique is applicable to any type of resource and functional unit, our goal here is to minimize the DSP usage without affecting loop throughput; we thus apply

Table 1. Benchmark Characteristics: Their Loop (i.e., CFG Cycle) and Loop Nest Count, Scheduling Property, Number of Sharing Candidates (fadd Indicates Floating Point Adders, and fmul Floating Point Multipliers), and Possible Types of Sharing

Benchmark	**Loops**	**Loop nests**	**Property**	Sharing candidates	Sharing types
atax	3	1	regular	2 fadd, 2 fmul	in nest
bicg	2	1	regular	2 fadd, 2 fmul	in nest
gemm	3	1	regular	3 fmul	in nest
gemver	7	4	regular	5 fadd, 6 fmul	in, across nest
gesummv	2	1	regular	3 fadd, 4 fmul	in nest
2mm	6	2	regular	2 fadd, 4 fmul	in, across nest
3mm	9	3	regular	3 fadd, 3 fmul	in, across nest
mvt	4	2	regular	2 fadd, 2 fmul	in, across nest
gsum	2	1	irregular	5 fadd, 4 fmul	in nest
gsumif	3	1	irregular	7 fadd, 4 fmul	in nest

Algorithm 1 to share every type of floating-point operation realized in DSPs. We use ModelSim to measure the execution cycle counts and for functional verification. We target a Xilinx Kintex-7 FPGA and use Xilinx floating-point operations (encapsulated in wrappers with handshake signals to communicate with other dataflow units). All memory operations connect to dual-port BRAMs. We obtain the clock period and resource usage from Vivado after place and route.

Although we use Dynamatic for our implementation, our strategy is perfectly general and applies to any dynamic HLS approach (e.g., that of Elakhras et al. [14], Budiu et al. [3], or Li et al. [25]). Our strategy relies on the performance analysis of Dynamatic to decide what to share—any alternative approach that determines the average loop throughput could be used instead (alternatively, Dynamatic's performance analysis could be employed on dataflow circuits constructed in a different manner, as demonstrated by Elakhras et al. [14]). To enforce operation order at the shared unit input, we use the existing in-order control network of Dynamatic; if not available, this network could be easily added to any dataflow circuit obtained from C code using standard control flow analyses [33].

7.2 Results: Effectiveness of the Sharing Strategy

Tables 2 and 3 compare dataflow circuits that do not implement sharing (i.e., circuits produced by Dynamatic) with the circuits optimized with our sharing strategy. The circuits without sharing (*Naive*) achieve the best possible pipelines (i.e., limited exclusively by the loop-carried dependencies) and a minimal number of cycles. Yet, they employ an individual functional unit for each operation, reflected in their DSP usage. In contrast, our designs (*Shared*) share functional units among multiple operations of the same type, thus significantly reducing the number of employed DSPs. Our strategy ensures that the loop throughput remains unchanged, as evident from the cycle count, which either remains identical or slightly increases. This increase is due to the pipeline latency increase, i.e., some operations using a shared unit execute later than in the original circuit (see Section 4.3) or transient effects when independent loops overlap, i.e., when one loop is ending and another one is starting, sharing temporarily lowers throughput as both loops compete for a shared resource (see Section 6). These effects are perfectly in line with what we described earlier and arguably acceptable for the significant DSP savings.

The minor differences in **clock period (CP)** are largely due to the timing variations caused by FPGA place and route; the interactions of the in-order network and the selector unit from Figure 6(b) sometimes contribute to these variations. These discrepancies are orthogonal to our

Table 2. Resources (i.e., DSPs, LUTs, and FFs) of Dataflow Circuits without Sharing (i.e., *Naive*, Obtained by Dynamatic [19]) and with Sharing (i.e., *Shared*, this Contribution)

Benchmark	**DSPs**			**LUTs**			**FFs**		
	Naive	Shared	ratio	Naive	Shared	ratio	Naive	Shared	ratio
atax	10	5	**0.50**	1970	2076	1.05	2206	1997	0.91
bicg	10	5	**0.50**	1627	1602	0.98	2018	1814	0.90
gemm	11	5	**0.45**	2339	2448	1.05	2500	2491	1.00
gemver	28	10	**0.36**	5580	5433	0.97	6753	5418	0.80
gesummv	18	5	**0.28**	2648	2666	1.01	3163	2528	0.80
2mm	16	5	**0.31**	3785	4200	1.11	4155	4153	1.00
3mm	15	5	**0.33**	3700	3653	0.99	3524	3096	0.88
mvt	10	5	**0.50**	2017	2029	1.01	2253	1878	0.83
gsum	22	5	**0.23**	2235	1989	0.89	2980	1708	0.57
gsumif	26	5	**0.19**	2807	2072	0.74	3865	1976	0.51
average			**0.36**			0.98			0.82

We obtain the resources from Vivado, after place and route. In all benchmarks, our sharing strategy successfully identifies sharing opportunities and reduces the DSP count, which was our primary target.

Table 3. Timing of Dataflow Circuits without Sharing (i.e., *Naive*, Obtained by Dynamatic [19]) and with Sharing (i.e., *Shared*, this Contribution)

Benchmark	**Cycle count**		**CP (ns)**		**Exec. time (μs)**		
	Naive	Shared	Naive	Shared	Naive	Shared	ratio
atax	4140	4459	4.9	4.3	20.3	19.2	0.95
bicg	7909	7910	4.6	4.3	36.4	34.0	0.93
gemm	68827	68827	5.7	4.9	392.3	337.3	0.86
gemver	1817	1899	5.1	5.6	9.3	10.6	1.15
gesummv	7952	8391	5.0	4.9	39.8	41.1	1.03
2mm	16610	17325	5.5	5.6	91.4	97.0	1.06
3mm	24557	24621	5.2	5.5	127.7	135.4	1.06
mvt	15708	15740	4.9	4.9	77.0	77.1	1.00
gsum	2473	2473	5.6	5.8	13.8	14.3	1.04
gsumif	2338	2419	5.3	5.8	12.4	14.0	1.13
average							1.02

We measure the cycle count in simulation and obtain the clock period (i.e., CP) from Vivado, after place and route. As expected, the throughput of the loops remains unchanged (as reflected in the cycle counts, that only marginally increase due to pipeline latency increases).

work and have been extensively discussed in the context of the timing analysis we rely on [23]. In addition to significant DSP reductions, our designs typically require fewer LUTs and FFs, which indicates that the complexity of the sharing mechanism (i.e., selector at unit input, branch at unit output) is minor compared to the shared computational units with their dataflow wrapper logic (i.e., the reduction of the wrapper resources compensates for the sharing mechanism, hence the LUT and FF decrease).

We summarize our main results from Tables 2 and 3 in Figure 10, which shows the execution time (i.e., the product of the CP and the cycle count) and resources (i.e., DSPs, LUTs, and FFs) of our designs, normalized to the naive designs without sharing. All our solutions are Pareto optimal

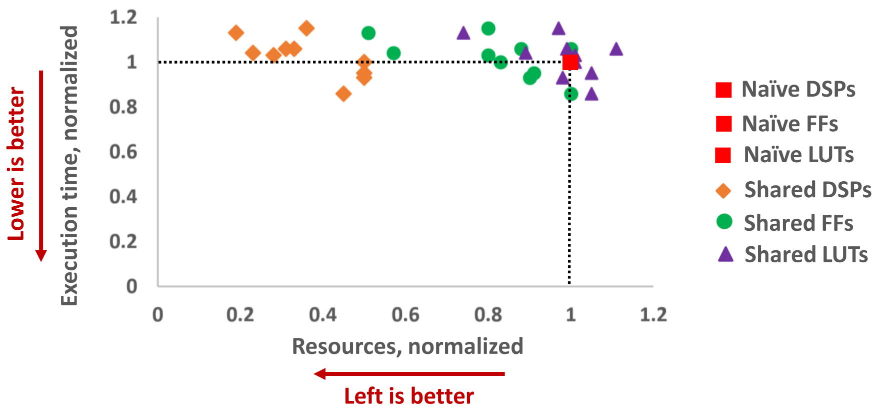


Fig. 10. Execution time and resources of dataflow circuits with sharing, normalized to the designs without sharing. Our main goal is to reduce the DSP count, which we successfully achieved.

in terms of DSPs; some designs even dominate their naive counterpart due to the coincidental reduction in CP. While we opted to identify sharing opportunities that do not affect throughput, our sharing mechanism can be easily extended to further explore the design space and discover other Pareto optimal solutions.

7.3 Results: Comparison with Vivado HLS

In the previous section, we demonstrated that our methodology effectively shares units in dataflow designs. We are now interested in comparing the capabilities of our sharing strategy with that of a standard, statically scheduled HLS tool. It should be noted upfront that, aside from *gsum* and *gsumif*, none of the benchmarks we explore have characteristics that can take advantage of dynamic scheduling. Hence, it is reasonable to expect that our circuits incur resource (i.e., LUT and FF) and timing (i.e., CP) overheads—we already observed these effects in prior work [9, 18]. Our purpose here is to investigate if the unit count (i.e., number of DSPs) achieved by our sharing strategy matches that of state-of-the-art HLS solutions.

We synthesized the benchmarks from Section 7.1 with Vivado HLS [36]; we employ the pipeline directive in all innermost loops and do not impose any resource constraints. Hence, the HLS tool maximizes performance (i.e., throughput) while minimizing the number of units—it shares as many units as possible and achieves the *minimal* DSP count for the best II, which qualitatively matches our strategy from Section 6.

Tables 4 and 5 compare the results obtained by Vivado HLS with dataflow circuits with sharing (i.e., *Shared* results from Tables 2 and 3). The Vivado designs employ the exact same number of DSPs as our solutions, which validates that our strategy successfully identified all sharing opportunities. None of the benchmarks suffer due to the operation ordering across BBs (Section 4.4), which indicates the effectiveness of our approach in a variety of practical cases. As anticipated, the static kernels require fewer LUTs and FFs and achieve a lower CP (typically resulting in a lower total execution time). Our goal here was to share computational resources (i.e., DSPs) as much as static HLS does, which we have successfully achieved.

The dynamic designs that implement the irregular benchmarks (i.e., *gsum* and *gsumif*) Pareto-dominate their static counterparts in execution time by adapting the throughput at runtime to the actual control outcomes (i.e., they require significantly fewer clock cycles to execute, therefore decreasing the total execution time). Whenever a long-latency conditional statement is executed, the throughput is temporarily lowered due to the conditional data dependencies—this lowering

Table 4. Resources (i.e., DSPs, LUTs, and FFs) of Vivado HLS Circuits (i.e., *Vivado*) and our Dataflow Circuits with Sharing (i.e., *Shared*, Repeated from Table 2), Obtained from Vivado, after Place and Route

Bench-mark	**DSPs**			**LUTs**			**FFs**		
	Vivado HLS	Shared	ratio	Vivado HLS	Shared	ratio	Vivado HLS	Shared	ratio
atax	5	5	**1.00**	388	2076	5.35	762	1997	2.62
bicg	5	5	**1.00**	425	1602	3.77	824	1814	2.20
gemm	5	5	**1.00**	458	2448	5.34	837	2491	2.98
gemver	10	10	**1.00**	1032	5433	5.26	1631	5418	3.32
gesummv	5	5	**1.00**	553	2666	4.82	944	2528	2.68
2mm	5	5	**1.00**	598	4200	7.02	963	4153	4.31
3mm	5	5	**1.00**	666	3653	5.48	1104	3096	2.80
mvt	5	5	**1.00**	481	2029	4.22	802	1878	2.34
gsum	5	5	**1.00**	558	1989	3.56	1023	1708	1.67
gsumif	5	5	**1.00**	542	2072	3.82	963	1976	2.05
average			**1.00**			4.86			2.69

The matching DSP counts indicate that our approach successfully identified all sharing opportunities. The LUT and FF overheads of dataflow circuits are expected and orthogonal to our sharing contribution.

Table 5. Timing of Vivado HLS Circuits (i.e., *Vivado*) and our Dataflow Circuits with Sharing (i.e., *Shared*, Repeated from Table 3)

Benchmark	**Cycle count**		**CP (ns)**		**Exec. time (μs)**		
	Vivado HLS	Shared	Vivado HLS	Shared	Vivado HLS	Shared	ratio
atax	5041	4459	3.3	4.3	16.6	19.2	1.15
bicg	9421	7910	3.3	4.3	31.1	34.0	1.09
gemm	91201	68827	3.2	4.9	291.8	337.3	1.16
gemver	2534	1899	3.4	5.6	8.6	10.6	1.23
gesummv	9029	8391	3.4	4.9	30.7	41.1	1.34
2mm	24402	17325	3.3	5.6	80.5	97.0	1.20
3mm	34803	24621	3.3	5.5	114.8	135.4	1.18
mvt	18782	15740	3.3	4.9	62.0	77.1	1.24
gsum	10067	2473	3.4	5.8	34.2	14.3	0.42
gsumif	10047	2419	3.5	5.8	35.2	14.0	0.40
average							1.04

Most of our benchmarks are regular kernels which do not benefit from dynamic scheduling; the exceptions are *gsum* and *gsumif*, where dynamic scheduling significantly outperforms static scheduling. The CP overheads of dataflow circuits are expected and orthogonal to our sharing contribution.

allows the conditional operations to share functional units and reduces the DSP counts to exactly those of the static kernels.

Surprisingly, all our solutions require fewer clock cycles to execute than the static solutions—while this effect is expected for *gsum* and *gsumif*, there is no fundamental reason for the dynamic kernels to execute faster in the other, perfectly regular, benchmarks. There are two explanations: (1) Sometimes, our designs overlap different loops more effectively than Vivado HLS; a similar overlapping could be achieved in Vivado HLS using the dataflow pragma, but this optimization limits resource sharing [35] and prevents us from comparing DSP-optimal pipelined designs. (2) In some cases, the retiming algorithms of Vivado place an additional register on the critical loops and increase the II; we employ a different register placement strategy [23] which does not need this register. These effects are orthogonal to our contribution and have only a quantitative effect on the results; the matching DSP counts of the static and dynamic designs clearly indicate the effectiveness of our sharing approach in achieving the best possible (i.e., minimal) number of functional units.

To sum up, the high-level idea of unit sharing is the same in static and dynamic scheduling; yet, the absence of a fixed, predetermined schedule in dynamically scheduled circuits calls for fundamentally different strategies to achieve these goals: (1) Both scheduling approaches aim to share functional units to minimize their idle time (i.e., each unit should be fully used and constantly busy computing). In static circuits, the information on unit usage is determined based on the scheduled execution times of particular operations—operations that are not scheduled to start simultaneously can share a resource. This information does not exist in dynamically scheduled circuits; instead, sharing is determined based on average computing rates—operations that receive data at a low rate can share a resource. (2) Both sharing implementations require data multiplexing at the shared unit input to ensure correctness and minimal performance damages. In static circuits, this order is enforced via scheduling (i.e., the exact cycle time of the start of each operation that the shared unit performs is fixed at compile time by the HLS tool). In dynamic circuits, the exact execution times are unknown and exact arrival and entry times may differ based on the dynamic variabilities in the circuit execution. Instead, the multiplexing logic enforces an *order* in which data should enter the shared unit; as before, the order is determined based on the average computing rate (i.e., the order should be such that the maximal computing rate is maintained). The fact that our circuits achieve the same sharing capabilities as their static counterparts indicates that our sharing strategy for dynamically scheduled circuits qualitatively matches that of static HLS.

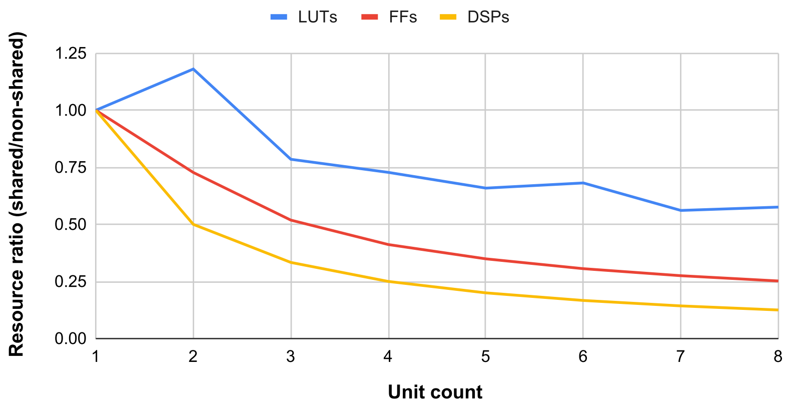
It is important to note that, in certain situations, the throughput of dataflow circuits may change during circuit execution (e.g., in the presence of variable-latency events or irregular control flow). This may cause discrepancies between the average and actual achieved throughput and, consequently, unit occupancy that our sharing strategy relies on. This is completely different than in static HLS where, in the presence of unpredictable events, the HLS tool creates a pessimistic schedule—all execution times are known, but conservative. Modeling the performance of dataflow circuits in the presence of irregular events has been extensively studied in the context of the performance analysis that we use [24]; our sharing results for irregular benchmarks (i.e., *gsum* and *gsumif*) demonstrate its effectiveness even in such situations.

7.4 Results: Sharing Mechanism Analysis

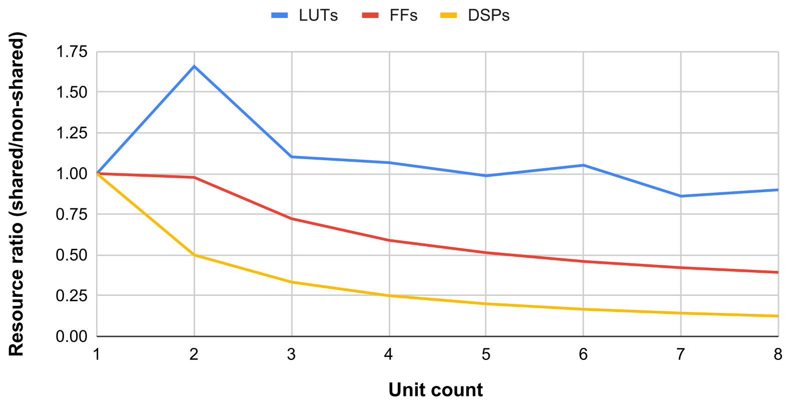
Our sharing mechanism consists of the following components: (1) Selector at the unit input, (2) FIFO on the unit side, and (3) branch and *t-buffs* at the unit output. We are interested in understanding the resource cost of each of these components as well as their implications on the total dataflow circuit resources. To this end, we synthesize the sharing components in isolation targeting various sharing configurations. We vary the number and type of shared units that the mechanism should support: we explore mechanisms for sharing 2 to 8 floating point adders and 2 to 8 floating point multipliers (i.e., the same units that our benchmarks from Section 7.1 contain). We investigate how each component scales with the increase of shared units and we evaluate the resource savings due to sharing.

Our results are reported in Tables 6 to 11 and visualized in Figure 11. We here summarize our main findings:

(1) *DSPs.* As explained earlier, our primary target is DSP reduction; this is consistent with default sharing strategies of standard HLS tools (e.g., Vivado HLS) and a reasonable goal considering the expensiveness and scarcity of DSPs on FPGAs (e.g., the FPGA we employ for our experiments contains only 600 DSPs; in contrast, it has over 160 thousand LUTs [37]).



(a) Sharing floating point adders.



(b) Sharing floating point multipliers.

Fig. 11. Summary of results from Tables 6 to 11. The graphs indicate the resources of dataflow circuits with sharing, normalized to circuits without sharing, for different numbers of floating point adders (a) and multipliers (b). In both cases, all resource savings increase with the number of shared resources, indicating the scalability of our approach.

Our sharing logic does not contain DSPs and sharing removes the DSPs of the floating point units—consistently with our benchmark evaluation, the DSP savings are significant and increase with the number of shared units.

(2) *FFs.* Although we did not attempt to save FFs, our approach systematically reduces the total FF count: sharing removes a large number of pipeline registers in the floating point unit wrappers, and our sharing components require a comparably small number of FFs to encode the ordering information, communicate through the FIFO, and store data in the output *t-buffs*. The FF savings increase with the number of shared units.

(3) *LUTs.* Our sharing components contain multiplexing logic at the shared unit input and output; this is the most significant cost in any sharing mechanism. When only two units are shared, the multiplexing logic dominates the floating point unit LUTs, resulting in an increase in total LUTs. For larger sharing configurations, the LUT count, generally, improves: the savings are more significant for the adder, as it contains a larger number of LUTs that are removed when sharing.

Table 6. DSP Savings when Sharing Float Adders

No sharing		Sharing						
fadd #	**$DSP_{fadd, total}$**	fadd #	DSP_{fadd}	DSP_{sel}	DSP_{FIFO}	DSP_{branch}	**DSP_{total}**	**ratio**
2	**4**	1	2	0	0	0	**2**	**0.50**
3	**6**	1	2	0	0	0	**2**	**0.33**
4	**8**	1	2	0	0	0	**2**	**0.25**
5	**10**	1	2	0	0	0	**2**	**0.20**
6	**12**	1	2	0	0	0	**2**	**0.17**
7	**14**	1	2	0	0	0	**2**	**0.14**
8	**16**	1	2	0	0	0	**2**	**0.13**

The total number of DSPs without sharing corresponds to the DSPs of all employed floating point adders, $DSP_{fadd, total}$. The total number of DSPs for the shared implementation, DSP_{total}, corresponds to the sum of the DSPs of the shared adder, selector, FIFO, and branching logic (branch and *t-buffs*). Our sharing mechanism does not contain DSPs, and sharing removes DSP-based adders, thus, the more adders we share, the more significant the DSP savings.

Table 7. FF Savings when Sharing Float Adders

No sharing		Sharing						
fadd #	**$FF_{fadd, total}$**	fadd #	FF_{fadd}	FF_{sel}	FF_{FIFO}	FF_{branch}	**FF_{total}**	**ratio**
2	**730**	1	365	88	12	66	**531**	**0.73**
3	**1095**	1	365	91	13	99	**568**	**0.52**
4	**1460**	1	365	91	13	132	**601**	**0.41**
5	**1825**	1	365	93	14	165	**637**	**0.35**
6	**2190**	1	365	93	14	198	**670**	**0.31**
7	**2555**	1	365	93	14	231	**703**	**0.28**
8	**2920**	1	365	93	14	264	**736**	**0.25**

The total number of FFs without sharing corresponds to the FFs of all employed floating point adders, $FF_{fadd, total}$. The total number of FFs for the shared implementation, FF_{total}, corresponds to the sum of the FFs of the shared adder, selector, FIFO, and branching logic (branch and *t-buffs*). The number of FFs in the shared logic is smaller than that in the adder; thus, the more adders we share, the more significant the FF savings.

Table 8. LUT Savings when Sharing Float Adders

No sharing		Sharing						
fadd #	**$LUT_{fadd, total}$**	fadd #	LUT_{fadd}	LUT_{sel}	LUT_{FIFO}	LUT_{branch}	**LUT_{total}**	**ratio**
2	**360**	1	180	190	18	37	**425**	**1.18**
3	**540**	1	180	166	19	59	**424**	**0.79**
4	**720**	1	180	247	19	78	**524**	**0.73**
5	**900**	1	180	292	23	98	**593**	**0.66**
6	**1080**	1	180	415	23	118	**736**	**0.68**
7	**1260**	1	180	367	23	137	**707**	**0.56**
8	**1440**	1	180	470	23	156	**829**	**0.58**

The total number of LUTs without sharing corresponds to the LUTs of all employed floating point adders, $LUT_{fadd, total}$. The total number of LUTs for the shared implementation, LUT_{total}, corresponds to the sum of the LUTs of the shared adder, selector, FIFO, and branching logic (branch and *t-buffs*). Our sharing mechanism contains multiplexing logic at the unit input and output; when sharing two adders, this logic introduces a LUT overhead. In all other cases, the LUT count of the shared logic is smaller than that of the removed adders; LUT savings increase with the number of shared units.

Table 9. DSP Savings when Sharing Float Multipliers

No sharing		**Sharing**						
fmul #	**DSP$_{\text{fmul, total}}$**	fmul #	DSP$_{\text{fmul}}$	DSP$_{\text{sel}}$	DSP$_{\text{FIFO}}$	DSP$_{\text{branch}}$	**DSP$_{\text{total}}$**	**ratio**
2	**6**	1	3	0	0	0	**3**	**0.50**
3	**9**	1	3	0	0	0	**3**	**0.33**
4	**12**	1	3	0	0	0	**3**	**0.25**
5	**15**	1	3	0	0	0	**3**	**0.20**
6	**18**	1	3	0	0	0	**3**	**0.17**
7	**21**	1	3	0	0	0	**3**	**0.14**
8	**24**	1	3	0	0	0	**3**	**0.13**

The total number of DSPs without sharing corresponds to the DSPs of all employed floating point multipliers, DSP$_{\text{fmul, total}}$. The total number of DSPs for the shared implementation, DSP$_{\text{total}}$, corresponds to the sum of the DSPs of the shared multiplier, selector, FIFO, and branching logic (branch and *t-buffs*). Our sharing mechanism does not contain DSPs, and sharing removes DSP-based multipliers, thus, the more multipliers we share, the more significant the DSP savings.

Table 10. FF Savings when Sharing Float Multipliers

No sharing		**Sharing**						
fmul #	**FF$_{\text{fmul, total}}$**	fmul #	FF$_{\text{fmul}}$	FF$_{\text{sel}}$	FF$_{\text{FIFO}}$	FF$_{\text{branch}}$	**FF$_{\text{total}}$**	**ratio**
2	**344**	1	172	88	10	66	**336**	**0.98**
3	**516**	1	172	91	11	99	**373**	**0.72**
4	**688**	1	172	91	11	132	**406**	**0.59**
5	**860**	1	172	93	12	165	**442**	**0.51**
6	**1032**	1	172	93	12	198	**475**	**0.46**
7	**1204**	1	172	93	12	231	**508**	**0.42**
8	**1376**	1	172	93	12	264	**541**	**0.39**

The total number of FFs without sharing corresponds to the FFs of all employed floating point multipliers, FF$_{\text{fmul, total}}$. The total number of FFs for the shared implementation, FF$_{\text{total}}$, corresponds to the sum of the FFs of the shared multiplier, selector, FIFO, and branching logic (branch and *t-buffs*). The number of FFs in the shared logic is smaller than that in the multiplier; thus, the more multipliers we share, the more significant the FF savings.

Table 11. LUT Savings when Sharing Float Multipliers

No sharing		**Sharing**						
fmul #	**LUT$_{\text{fmul, total}}$**	fmul #	LUT$_{\text{fmul}}$	LUT$_{\text{sel}}$	LUT$_{\text{FIFO}}$	LUT$_{\text{branch}}$	LUT$_{\text{total}}$	**ratio**
2	**208**	1	104	190	14	37	**345**	**1.66**
3	**312**	1	104	166	15	59	**344**	**1.10**
4	**416**	1	104	247	15	78	**444**	**1.07**
5	**520**	1	104	292	19	98	**513**	**0.99**
6	**624**	1	104	415	19	118	**656**	**1.05**
7	**728**	1	104	367	19	137	**627**	**0.86**
8	**832**	1	104	470	19	156	**749**	**0.90**

The total number of LUTs without sharing corresponds to the LUTs of all employed floating point multipliers, LUT$_{\text{fmul, total}}$. The total number of LUTs for the shared implementation, LUT$_{\text{total}}$, corresponds to the sum of the LUTs of the shared multiplier, selector, FIFO, and branching logic (branch and *t-buffs*). Our sharing mechanism contains multiplexing logic at the unit input and output; in smaller sharing configurations, this logic introduces a LUT overhead; when sharing more units, the LUT count improves. These trends are acceptable for the significant DSP and FF savings that sharing systematically achieves.

All these findings are consistent with our results from Figure 10: sharing results in significant DSP and FF savings, typically accompanied by a LUT saving; in some configurations, sharing results in a minor LUT overhead which is typically acceptable for the significant DSP and FF savings. Furthermore, the fact that the resource reduction increases with the number of shared units indicates the scalability and broad applicability of our sharing approach.

8 RELATED WORK

Standard, statically-scheduled HLS tools [7, 36] perform scheduling in conjunction with resource allocation and sharing [38]; they trade-off area and performance by deciding the cycle in which each operation executes and allocating units accordingly. Dataflow circuits face the same optimization objectives and area-performance trade-offs; however, there is no predetermined schedule and no information on when each operation executes to decide how many units to employ.

Several dataflow-oriented HLS approaches support forms of resource sharing. Bluespec [2] allows the user to specify the appropriate control logic around a shared resource in a dataflow network using guarded atomic actions. Nielsen et al. [26] discuss dataflow construct sharing in the Balsa asynchronous hardware description language. Neither of these works addresses the correctness and performance aspects of sharing that we discuss here; furthermore, our approach automatically achieves the correct sharing logic without any user-given specifications. In the context of dataflow machines [1], a processor-like I-structure manages tokens entering and exiting a shared function; yet, this centralized mechanism is not available in spatial dataflow circuits.

Edwards et al. [13] present a nondeterministic sharing mechanism for dataflow circuits, similar to the one shown in Figure 1; yet, as we have illustrated in this paper, this mechanism is not sufficient to guarantee the absence of deadlock in dataflow circuits obtained out of imperative code. Cortadella et al. [10] describe sharing in elastic circuits and indicate the need to build a local scheduler to decide, at each clock cycle, which input can use the resource, both for avoiding unit starvation and for performance benefits. Similarly, Hansen and Singh [15] employ a local, centralized FSM for every shared unit in their asynchronous pipelines to regulate the multiplexing of tokens at its inputs. However, both these approaches are applicable only to simple loops without conditionals, where a predetermined sequence of inputs can be encoded into a centralized scheduler; therefore, they are not applicable to circuits obtained out of high-level code. In contrast, our method applies to generic HLS constructs and circuits with control flow—we use a distributed network to control the multiplexing of tokens dynamically, based on particular control flow outcomes.

9 CONCLUSIONS

Resource sharing is one of the key optimizations in high-level synthesis; if dataflow circuits are to compete with standard HLS, they need to be able to exploit this optimization opportunity. In this work, we present a resource sharing methodology for dataflow circuits obtained from C code; our key contribution is a sharing mechanism that achieves correct, deadlock-free execution. In addition, we present a method to identify sharing opportunities that do not compromise performance. On a set of benchmarks, we demonstrate the ability of our approach to significantly improve the resource efficiency of dataflow circuits and to match the sharing capabilities of a standard HLS tool. Our sharing mechanism is key to achieve different area-performance tradeoffs in dataflow designs as well as to make them competitive in terms of computational resources (i.e., functional units and the corresponding DSP count) with circuits achieved using standard HLS techniques.

REFERENCES

[1] Arvind and Rishiyur S. Nikhil. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers* 39, 3 (March 1990), 300–18.

[2] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. 2004. High-level synthesis: An essential ingredient for designing complex ASICs. In *Proceedings of the International Conference on Computer-Aided Design*. San Jose, Calif., 775–82.

[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, Tex., 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, Calif., 362–69.

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

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

[7] 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), 24:1–24:27.

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

[9] 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, Calif., 288–98.

[10] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. 2010. Elastic systems. In *Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign*. Grenoble, 149–58.

[11] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference*. San Francisco, Calif., 657–62.

[12] Doug Edwards and Andrew Bardsley. 2002. Balsa: An asynchronous hardware synthesis language. *Comput. J.* 45, 1 (Jan. 2002), 12–18.

[13] 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, Austria, 175–84. https://doi.org/10.1145/3127041.3127055

[14] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2022. Unleashing parallelism in elastic circuits with faster token delivery. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 253–61.

[15] John Hansen and Montek Singh. 2012. Multi-token resource sharing for pipelined asynchronous systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Dresden, 1191–96.

[16] Lana Josipović. 2021. *High-level Synthesis of Dynamically Scheduled Circuits*. Ph.D. Dissertation. EPFL.

[17] 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), 125:1–125:19. https://doi.org/10.1145/3126525

[18] 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, Calif., 127–36.

[19] 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, Calif., 1–10.

[20] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2021. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine* 21, 2 (May 2021), 97–118.

[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. https://doi.org/10.1109/TCAD.2021.3105574

[22] Lana Josipović, Axel Marmet, Andrea Guerrieri, and Paolo Ienne. 2022. Resource sharing in dataflow circuits. In *Proceedings of the 30th IEEE Symposium on Field-Programmable Custom Computing Machines*. New York, 1–9.

[23] 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, Calif., 186–96.

[24] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2021. Buffer placement and sizing for high-performance dataflow circuits. *ACM Transactions on Reconfigurable Technology and Systems* 15, 1 (Nov. 2021), 1–32.

[25] Rui Li, Lincoln Berkley, Yihang Yang, and Rajit Manohar. 2021. Fluid: An asynchronous high-level synthesis tool for complex program structures. In *Proceedings of the 27th International Symposium on Asynchronous Circuits and Systems*. Beijing, China 1–8.

[26] Sune Fallgaard Nielsen, Jens Sparsø, and Jan Madsen. 2009. Behavioral synthesis of asynchronous circuits using syntax directed translation as backend. *IEEE Transactions on Very Large Scale Integration Systems* 17, 2 (Feb. 2009), 248–61.

[27] Louis-Noël Pouchet. 2012. *PolyBench: The Polyhedral Benchmark Suite*. http://www.cs.ucla.edu/pouchet/software/polybench.

[28] C. V. Ramamoorthy and Gary S. Ho. 1980. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering* 6, 5 (Sept. 1980), 440–49.

[29] Chander Ramchandani. 1974. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Ph.D. Dissertation. Massachusetts Institute of Technology.

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

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

[32] John Teifel and Rajit Manohar. 2004. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proceedings of the 10th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. Crete, Greece, 17–27.

[33] Linda Torczon and Keith Cooper. 2011. *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.

[34] Muralidaran Vijayaraghavan and Arvind. 2009. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 7th International Conference on Formal Methods and Models for Codesign*. Cambridge, MA, 171–80.

[35] Xilinx Inc. 2018. *Vivado Design Suite User Guide: High-Level Synthesis*. Xilinx Inc. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.

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

[37] Xilinx Inc. 2021. *7 Series Product Selection Guide*. Xilinx Inc. https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/7-series-product-selection-guide.pdf#K7.

[38] 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, Calif., 211–18.

Received 18 November 2022; revised 2 March 2023; accepted 1 May 2023