2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 978-1-6654-8332-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/FCCM53951.2022.9786084

Resource Sharing in Dataflow Circuits

Lana Josipović*, Axel Marmet†, Andrea Guerrieri†, and Paolo Ienne†
*ETH Zürich, Switzerland, †École Polytechnique Fédérale de Lausanne, Switzerland

***Abstract*—To achieve resource-efficient hardware designs, HLS tools share (i.e., time-multiplex) functional units among operations of the same type. This optimization is typically performed together 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. Yet, 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. Furthermore, 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 and we describe a sharing mechanism that achieves deadlock-free dataflow designs. On benchmarks obtained from C code, we show that our approach effectively implements resource sharing: it results in significant area savings (i.e., a DSP reduction of up to 81%) compared to dataflow circuits which do not support this feature and matches the sharing capabilities of a state-of-the-art HLS tool.**

I. Introduction

Standard HLS approaches [29], [7] rely on static scheduling: at compile time, they decide the clock cycles in which each operation will execute and 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 used in different clock cycles [25], [6], [30]. In contrast, dataflow or latency-insensitive protocols [8], [11], [27], [13] 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 applications [16]. Yet, this scheduling flexibility 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.

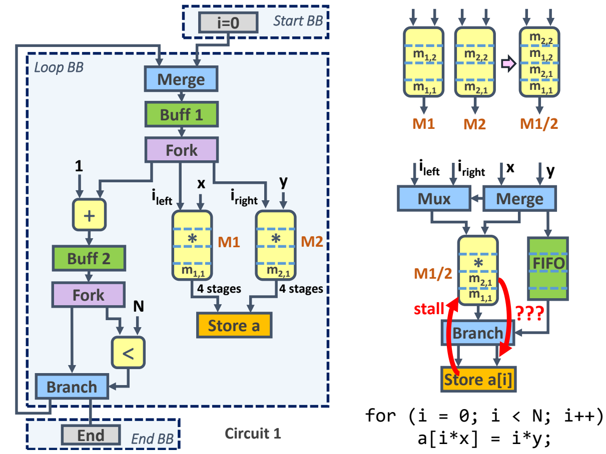

Fig. 1: Dataflow circuit and a possible resource sharing implementation. The multiplications could be computed using a single multiplier, with input and output multiplexing logic. Yet, this mechanism on its own does not guarantee that the circuit is deadlock-free nor that its performance is optimal.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

In this paper, we present a complete methodology to implement resource sharing in dataflow designs. We formulate the necessary requirements to ensure deadlock-free execution and implement a sharing mechanism accordingly. We then discuss how to minimize the performance impact of delays caused by sharing. 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 (i.e., DSP blocks).

II. Motivation

To illustrate the challenges of resource sharing in dataflow circuits, consider the example in Figure 1. This circuit has no centralized controller—all dataflow units are connected to their predecessor and successor units with handshake signals that regulate the flow of data (i.e., tokens); each operation executes as soon as it is ready and its inputs become available.

The execution starts when a token enters through the starting point; 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 has 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 operands, communicated by the 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 [16] 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.

III. Background and Related Work

In this section, we describe dataflow circuits and discuss how existing performance analysis techniques can be used to identify sharing opportunities in dataflow designs.

A. Dataflow Circuits

Several authors [16], [12], [26], [3] generate dataflow circuits from high-level programs; we follow an approach that produces *synchronous* dataflow circuits from C code [16]. 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) *merge* sends a token nondeterministically into a BB from one of the predecessor BBs, (2) *mux* is a deterministic version of a merge with an additional control input to select the input token, (3) *branch* sends the token to one of the successor BBs, as determined by the BB condition, (4) *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) *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) [20]. 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], [16], but only with impact on performance.

B. Resource Sharing in High-Level Synthesis

Standard, statically-scheduled HLS tools [29], [7] perform scheduling in conjunction with resource allocation and sharing [30]; 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 using guarded atomic actions. Nielsen et al. [21] discuss dataflow construct sharing in the Balsa asynchronous hardware description language. However, these works rely on user-given specifications and do not address the correctness and performance aspects of sharing that we consider. 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 in Figure 1; as discussed before, 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 build a local scheduler to decide, at each clock cycle, which input can use the resource. Similarly, Hansen et al. [14] use a centralized FSM for every shared unit in their asynchronous pipelines to regulate the multiplexing of tokens. Both these approaches are applicable only to simple loops without conditionals, where a predetermined sequence of inputs can be encoded into a centralized scheduler. In contrast, this paper presents a sharing method that supports generic HLS constructs and circuits with control flow—it is thus applicable for any dataflow circuit obtained out of high-level code.

C. Deciding What to Share in a Dataflow Circuit

Several authors discussed techniques to analyze the timing of dataflow circuits [20], [4], [24], [23], [5]; 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 [20]. 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 [19]: if the sum of the tokens in two

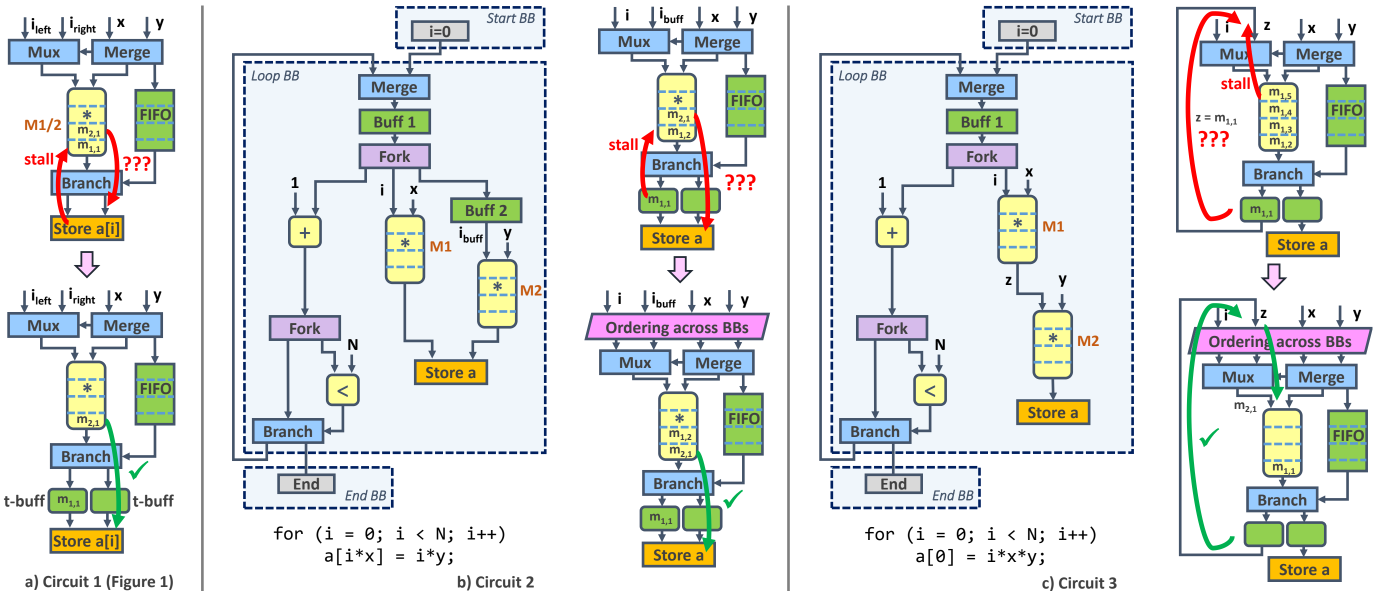

Fig. 2: Figure 2a shows the sharing hardware of the circuit in Figure 1; placing transparent buffers on the branch outputs avoids deadlock by allowing all tokens from a single datapath execution to exit the shared unit and all the computation of the datapath to complete. In circuits 2 and 3 (Figure 2b and 2c), deadlock occurs due to the reordering of tokens from different loop iterations. The solution 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.

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 in the top right of the figure. 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.

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 top right of Figure 1, $m_{2,1}$ 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 II, nothing in this analysis guarantees that the dataflow circuit with sharing is deadlock-free. We will address both of these issues in this paper.

IV. Resource Sharing in Dataflow Circuits

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

A. 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 top of Figure 2a, 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 bottom of Figure 2a 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 it by adding a 1-slot transparent buffer (see Section III-A), 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.

B. Sharing in General Datapaths

The methodology above 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.

Figures 2b and 2c show two examples where the mechanism from Section IV-A does not manage to prevent deadlock: (1) Circuit 2 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 3 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 III-A, 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 2, instead of consecutively consuming both tokens from the same BB execution, the unit inputs some tokens from the following iteration (i.e., next BB) which prevent one of the previous tokens from ever exiting the unit. (2) In circuit 3, 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 IV-A). 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.

C. 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 3, 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.

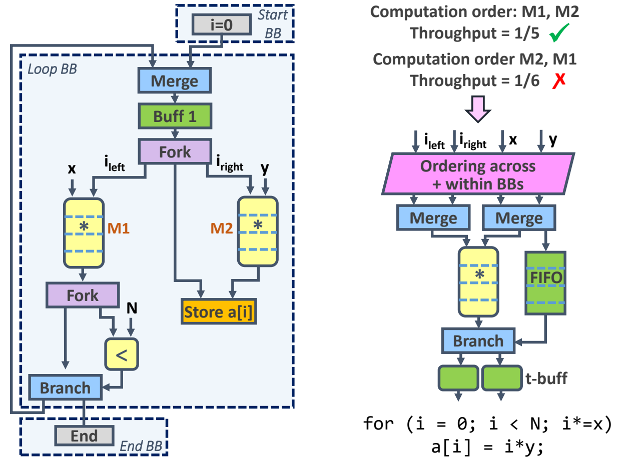

Fig. 3: Performance impact of sharing. The order in which tokens are sent to the shared unit may impact performance by postponing the execution of a throughput-limiting computation (in this example, operation M1 on the cycle). Hence, apart from ordering tokens from different BBs as they enter a shared unit, we also enforce the ordering of tokens of the same BB which has the minimal performance impact.

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 suggested on the right of Figure 3, such that the throughput impact is minimal. We incorporate this notion into our sharing strategy in Section VI: we use the performance analysis from Section III-C 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 IV-A) can be replaced by merges. We will detail our implementation of the ordering logic in Section V.

D. Extending the Ordering Scheme

The ordering rules described so far ensure the absence of deadlock by ordering tokens across BBs (Section IV-B); to ensure the best possible throughput in the presence of such ordering, we order operations within a BB as well (Section IV-C). Interestingly, ordering tokens across BBs may, in particular cases, lower the throughput, as it may limit the overlapping of operations from different loop iterations. This is the case in circuit 3 in Figure 2c: 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 IV-B lowers the throughput to 1/5—as suggested on the right of Figure 2c, the first computation of M2 ($m_{2,1}$) starts 4 cycles after the start of the first computation from M1 ($m_{1,1}$);

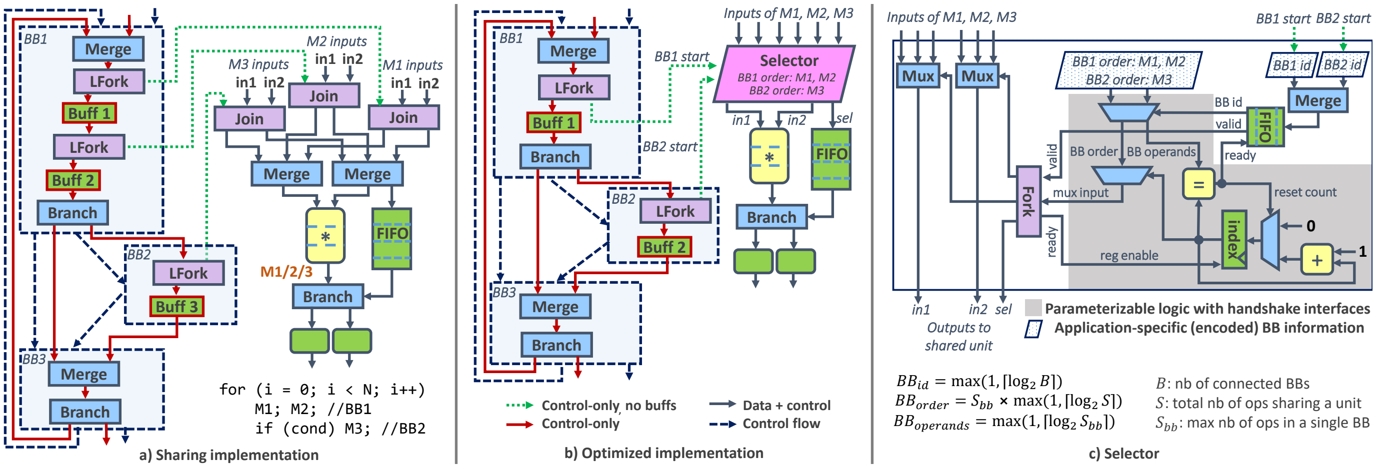

Fig. 4: Sharing implementation. A specialized in-order network and the selector unit enforce the specified ordering of operations in the shared unit based on the order of BB execution and the preencoded operation order for each BB.

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 operation within the iteration; if this value is higher than the initial II, it can constrain it.

Note that our ordering condition from Section IV-B is *sufficient* to guarantee the absence of deadlock, yet it is not always *necessary*—our generic ordering mechanism could be replaced by application-specific multiplexing and buffering. 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. 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 IV-B and IV-C. As we will later see, our strategy effectively implements sharing in realistic benchmarks.

V. Ordering Implementation

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

A. 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 dataless 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; this fork is synchronized (using a join) with a particular set of unit inputs. The fork must be lazy (see Section III-A) 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.

The loop in Figure 4a contains three multiplications: M1 and M2 in BB1, and M3 in conditionally executed BB2. The in-order network which supplies ordering information to the unit shared between M1, M2, and M3 is shown on the left of Figure 4a—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 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 once 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 3.

B. 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 in an existing HLS framework which already produces, for other purposes, an in-order network expressing the dynamic succession of executed BBs [15], [18]. Thus, we adapt our implementation to directly leverage this existing network.

Our simplified implementation is shown in Figure 4b. 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 4a 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.

Figure 4c 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 expressions for the number of bits of the encoded ordering information are shown in Figure 4c. Typically, only a few operations share a unit and these values are minor (in this case, *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 usually minor compared to the 32- or 64-bit data multiplexers (left of the selector), which are used in any sharing implementation and are not an overhead of our strategy.

VI. Putting It All Together

Algorithm 1 summarizes our resource sharing strategy. 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*, i.e., within a strongly connected component of the CFG. 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 III-C) 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 III-C 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 IV-C). 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. (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).

Algorithm 1: Sharing strategy.

1 // Input: units (all units of the same type)
2 // Input: sets (CFG loop nests, i.e., strongly connected components of the CFG)
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   Θ_s, *buffs = runPerformanceAnalysis (s)*
8   // Initialize groups to individual units
9   *groups (s)* = $\{u \mid u \in$ *units*, $u \in s\}$
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** $\dot{\Theta}_{g_1} + \dot{\Theta}_{g_2} \leq \mathbf{L}_u$ **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* $\in$ *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* $\in$ *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)*

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 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.

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 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

Benchmark	DSPs			LUTs			FFs			Cycle count		CP (ns)		Exec. time (μs)		
	Naive	Shared	ratio	Naive	Shared	ratio	Naive	Shared	ratio	Naive	Shared	Naive	Shared	Naive	Shared	ratio
atax	10	5	**0.50**	1970	2076	1.05	2206	1997	0.91	4140	4459	4.9	4.3	20.3	19.2	0.95
bicg	10	5	**0.50**	1627	1602	0.98	2018	1814	0.90	7909	7910	4.6	4.3	36.4	34.0	0.93
gemm	11	5	**0.45**	2339	2448	1.05	2500	2491	1.00	68827	68827	5.7	4.9	392.3	337.3	0.86
gemver	28	10	**0.36**	5580	5433	0.97	6753	5418	0.80	1817	1899	5.1	5.6	9.3	10.6	1.15
gesummv	18	5	**0.28**	2648	2666	1.01	3163	2528	0.80	7952	8391	5.0	4.9	39.8	41.1	1.03
2mm	16	5	**0.31**	3785	4200	1.11	4155	4153	1.00	16610	17325	5.5	5.6	91.4	97.0	1.06
3mm	15	5	**0.33**	3700	3653	0.99	3524	3096	0.88	24557	24621	5.2	5.5	127.7	135.4	1.06
mvt	10	5	**0.50**	2017	2029	1.01	2253	1878	0.83	15708	15740	4.9	4.9	77.0	77.1	1.00
gsum	22	5	**0.23**	2235	1989	0.89	2980	1708	0.57	2473	2473	5.6	5.8	13.8	14.3	1.04
gsumif	26	5	**0.19**	2807	2072	0.74	3865	1976	0.51	2338	2419	5.3	5.8	12.4	14.0	1.13

TABLE I: Timing and resources of dataflow circuits without sharing (i.e., *Naive*, obtained by Dynamatic [17]) and with sharing (i.e., *Shared*, this contribution). We measure the cycle count in simulation and obtain the timing and resources from Vivado, after place and route.

only in case of throughput-limiting operations on cycles; it is therefore effective in optimizing complex graphs with a large unit count. We here focus on sharing functional units and reducing the DSP count, but our strategy is applicable to other types of resources (e.g., memory blocks, LUTs) as well.

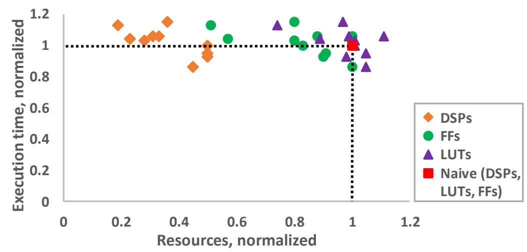

Fig. 5: Execution time and resources of dataflow circuits with sharing, normalized to the designs without sharing. Our main goal is to reduce the DSP count, which we successfully achieved.

VII. Evaluation

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

A. Methodology and Benchmarks

We evaluate a selection of floating-point kernels from the PolyBench suite [22] 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. 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 [17] that synthesizes C code into synchronous dataflow designs and implements the performance analysis from Section III-C. 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 Algorithm 1 to share every type of floating-point operation realized in DSPs. We use ModelSim to measure execution cycle count 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.

B. Results: Effectiveness of the Sharing Strategy

Table I compares 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 with 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 IV-C) 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 VI). 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 4b sometimes contribute to these variations. These discrepancies are orthogonal to our work and have been extensively discussed in the context of the timing analysis we rely on [20]. 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 Table I in Figure 5, which shows the execution time (i.e., the product of the CP

Benchmark	DSPs			LUTs			FFs			Cycle count		CP (ns)		Exec. time (μs)		
	Vivado	Shared	ratio	Vivado	Shared	ratio	Vivado	Shared	ratio	Vivado	Shared	Vivado	Shared	Vivado	Shared	ratio
atax	5	5	**1.00**	388	2076	5.35	762	1997	2.62	5041	4459	3.3	4.3	16.6	19.2	1.15
bicg	5	5	**1.00**	425	1602	3.77	824	1814	2.20	9421	7910	3.3	4.3	31.1	34.0	1.09
gemm	5	5	**1.00**	458	2448	5.34	837	2491	2.98	91201	68827	3.2	4.9	291.8	337.3	1.16
gemver	10	10	**1.00**	1032	5433	5.26	1631	5418	3.32	2534	1899	3.4	5.6	8.6	10.6	1.23
gesummv	5	5	**1.00**	553	2666	4.82	944	2528	2.68	9029	8391	3.4	4.9	30.7	41.1	1.34
2mm	5	5	**1.00**	598	4200	7.02	963	4153	4.31	24402	17325	3.3	5.6	80.5	97.0	1.20
3mm	5	5	**1.00**	666	3653	5.48	1104	3096	2.80	34803	24621	3.3	5.5	114.8	135.4	1.18
mvt	5	5	**1.00**	481	2029	4.22	802	1878	2.34	18782	15740	3.3	4.9	62.0	77.1	1.24
gsum	5	5	**1.00**	558	1989	3.56	1023	1708	1.67	10067	2473	3.4	5.8	34.2	14.3	0.42
gsumif	5	5	**1.00**	542	2072	3.82	963	1976	2.05	10047	2419	3.5	5.8	35.2	14.0	0.40

TABLE II: Timing and resources of Vivado HLS circuits (i.e., *Vivado*) and our dataflow circuits with sharing (i.e., *Shared*, repeated from Table I). The matching DSP counts indicate that our approach successfully identified all sharing opportunities. The LUT, FF, and CP overheads of dataflow circuits are expected and orthogonal to our sharing contribution. 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.

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 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.

C. 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], [16]. 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 VII-A with Vivado HLS [29]; 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 VI.

Table II compares the results obtained by Vivado HLS with dataflow circuits with sharing (i.e., *Shared* results from Table I). 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 IV-D), 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 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 [28] 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 [20] 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.

VIII. Conclusions

Resource sharing is one of the key optimizations in high-level synthesis; dataflow circuits could be competitive in this context only if they could exploit this optimization as well. 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 achieving different area-performance tradeoffs as well as to making dataflow designs 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 R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–18, Mar. 1990.

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

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

[4] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar. A general model for performance optimization of sequential systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 362–69, San Jose, Calif., Nov. 2007.

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

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

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems*, 13(2):24:1–24:27, Sept. 2013.

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

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

[10] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky. Elastic systems. In *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 149–58, July 2010.

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

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

[13] S. A. Edwards, R. Townsend, and M. A. Kim. Compositional dataflow circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 175–84, Vienna, Sept. 2017.

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

[15] L. Josipović, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems*, 16(5s):125:1–125:19, Sept. 2017.

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

[17] L. Josipović, A. Guerrieri, and P. Ienne. Dynamatic: From C/C++ to dynamically scheduled circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 1–10, Seaside, Calif., Feb. 2020.

[18] L. Josipović, A. Guerrieri, and P. Ienne. From C/C++ code to high-performance dataflow circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Aug. 2021. To appear.

[19] L. Josipović, A. Guerrieri, and P. Ienne. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine*, 21(2):97–118, May 2021.

[20] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella. Buffer placement and sizing for high-performance dataflow circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 186–96, Seaside, Calif., Feb. 2020.

[21] S. F. Nielsen, J. Sparsø, and J. Madsen. Behavioral synthesis of asynchronous circuits using syntax directed translation as backend. *IEEE Transactions on Very Large Scale Integration Systems*, 17(2):248–61, Feb. 2009.

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

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

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

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

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

[27] M. Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign*, pages 171–80, Cambridge, MA, July 2009.

[28] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*, 2018.

[29] Xilinx Inc. *Vivado High-Level Synthesis*, 2018.

[30] Z. Zhang and B. Liu. SDC-based modulo scheduling for pipeline synthesis. In *Proceedings of the 32nd International Conference on Computer-Aided Design*, pages 211–18, San Jose, Calif., Nov. 2013.