CRUSH: A Credit-Based Approach for Functional Unit Sharing in Dynamically Scheduled HLS

Jiahui Xu
ETH Zurich
Zurich, Switzerland

Lana Josipović
ETH Zurich
Zurich, Switzerland

ABSTRACT

Resource sharing in dynamic HLS-produced circuits is beneficial yet challenging—without a pre-defined schedule, out-of-order access to a shared resource creates resource dependencies between the operations, which may lead to circuit deadlock. In this work, we present CRUSH, a strategy that enables efficient functional unit sharing in dynamically scheduled HLS. CRUSH decouples the interactions of different operations in the shared resource and breaks resource dependencies, thus, deadlock freeness is guaranteed. Without the need to pessimistically order operation execution for correctness, (1) CRUSH systematically maintains performance and avoids the need for iterative and expensive performance optimization; (2) CRUSH seizes sharing opportunities enabled by out-of-order access to the shared unit; (3) CRUSH avoids complex access control mechanisms, thus, it is more resource-efficient. Compared to a prior strategy, CRUSH achieves a geomean of 10% DSP reduction, 89% reduction in the optimization runtime, and 15% FF reduction, with negligible performance degradation.

1 INTRODUCTION

Thanks to the flexibility provided by their fully distributed handshake communication, HLS-produced dataflow circuits, compared to their statically scheduled counterparts [6, 11, 32, 36], deliver high-performance for designs with irregular computation patterns [17, 20]. While enjoying the performance merit of the flexibility of the handshake logic, some common optimization techniques used in static HLS are equally applicable in the context of dynamic HLS, e.g., frequency optimization [24, 30], logic optimization [34, 35], and resource sharing [22].

Yet, resource optimization, especially resource sharing, is also challenging in dataflow circuits. When a physical resource is shared between multiple operations, during circuit execution, the pending computation of an operation might block the others that share the same physical resource; this, in conjunction with an existing data dependency, might create a dependency cycle, thus the circuit deadlocks or its performance degrades. Thus, producing correct and performant circuits with sharing requires a deadlock avoidance scheme and a systematic strategy to evaluate the impact of sharing on performance. The former typically results in complex and resource-expensive ordering mechanisms and the latter in a time-consuming evaluation of all sharing decisions [22]; both aspects limit the usability of sharing in complex HLS applications.

In this work, we present CRUSH, a sharing strategy for dataflow circuits that is scalable in optimization runtime and efficient in resource overhead. Inspired by the techniques well-known in interconnect systems [9, 15], our deadlock avoidance mechanism is simple, localized, and agnostic to the control flow mechanism of the original circuit—which makes it suitable for different HLS strategies for dataflow circuits [17, 20]. We highlight that our sharing strategy maintains the performance of the original dataflow circuit, thus, exhaustively evaluating the effect of sharing is unnecessary, which makes our solution superior in runtime over a prior solution.

2 BACKGROUND AND RELATED WORK

This section describes dataflow circuits and recent works addressing resource sharing.

2.1 Dataflow Circuits

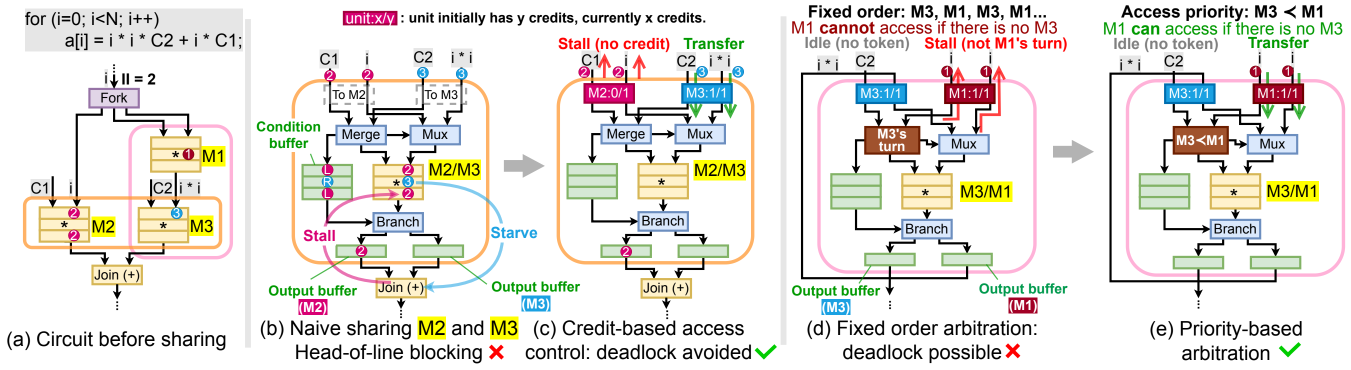
Dataflow circuits are built from *units* that communicate via *channels*. A channel consists of data and a pair of valid-ready handshake signals [13, 20]. Once the dependencies have been resolved (e.g., control and memory), units exchange *tokens* which encapsulate data; the exchange time is determined *dynamically* during runtime. Many works study the generation process of dataflow circuits from high-level code [3, 7, 16, 17, 20]. We here focus on HLS approaches that target C code [17, 20] and the generated circuits implement single-threaded programs [17, 20, 25]; our solution is equally applicable to the other dataflow HLS strategies.

The dataflow circuits that we consider are built from the following units: ■ A *fork* distributes a copy of the incoming token to each successor as soon as they are ready to receive it. ■ A *join* synchronizes multiple tokens before sending a token to its successor; it is typically used in arithmetic units to ensure the presence of all inputs before computing. ■ A *merge* propagates a token to its single output from one of its two data inputs. ■ A *mux* is a deterministic version of a merge with a control input to select the input token. ■ A *branch* propagates the received data token to one of its successors, depending on the value of the received condition token. ■ A *buffer* is used to store tokens, break combinational paths, and increase throughput; buffers can be arbitrarily placed on any channel without penalizing correctness [4, 20, 24].

Dataflow circuits are commonly modeled as a directed graph: nodes are units and edges are channels. Performance optimization is commonly done on *choice-free circuits* (CFCs)—subcircuits with no conditional execution [2, 4, 24, 30]. *Token occupancy*—the number of tokens that occupy channels—is used to determine how many buffer slots should be allocated. The occupancy ϕ_{op} of a pipelined unit op (calculated as $\frac{lat_{op}}{II_{CFC}}$, where lat_{op} is op's latency, and II_{CFC} is the CFC's *initiation interval* (II)) is used to identify underutilized units and where it is advantageous to share them [22].

2.2 Resource Sharing in Dataflow Circuits

Resource sharing is one of the key HLS optimization techniques for generating efficient circuits [5, 6, 14, 32, 36]. In standard, static scheduling–based HLS strategies, heuristics are often applied to



for (i=0; i<N; i++)
a[i] = i * i * C2 + i * C1;
unit:x/y : unit initially has y credits, currently x credits.
(a) Circuit before sharing
(b) Naive sharing M2 and M3 Head-of-line blocking ✗
(c) Credit-based access control: deadlock avoided ✓
Fixed order: M3, M1, M3, M1...
M1 cannot access if there is no M3
(d) Fixed order arbitration: deadlock possible ✗
Access priority: M3 < M1
M1 can access if there is no M3
(e) Priority-based arbitration ✓
Condition buffer
Stall
Starve
Stall (no credit)
Transfer
Idle (no token)
Stall (not M1's turn)
Output buffer

Figure 1: Naive resource sharing leads to circuit deadlock [22]. Figure 1a describes the circuit before sharing. Figure 1b and 1c describe why naively sharing cannot prevent a deadlock state caused by head-of-line blocking; we avoid it using credit-based access control. Figure 1d and 1e describe why a fixed order arbitration causes deadlock; we avoid it using priority-based arbitration.

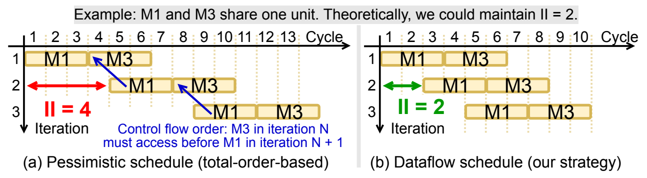


Figure 2: Schedules when sharing $M1$ and $M2$ in the circuit in Figure 1a. Figure 2a: A total token order might hurt II, thus limiting sharing opportunities. We can achieve the schedule in Figure 2b.

decide how sharing is done, as pipeline scheduling in the presence of sharing is a computationally complex task [5, 36].

Resource sharing in dataflow circuits has been addressed in limited use cases, e.g., in simple circuits without any conditionals [12, 16]. Several dataflow HLS approaches support sharing [1, 19, 22, 27]. To the best of our knowledge, only Josipović et al. [22] present a solution that addresses both the correctness and performance aspects of sharing in HLS-produced circuits: ▪ they point out that naive resource sharing leads to a deadlock, and ▪ they propose a total-token-order-based solution that prevents it. In the next section, we discuss the challenges of resource sharing in dataflow circuits, how the problem is addressed in the prior work, its inadequacy, and how we would like to address it.

3 THE CHALLENGES OF RESOURCE SHARING IN DATAFLOW CIRCUITS

This section describes the challenges of sharing in dataflow circuits, i.e., the correctness and performance aspects.

Consider the circuit in Figure 1a, which implements part of the code above it. A token carrying the value of i becomes available at the circuit's input every second cycle; it is duplicated by *Fork* and sent to $M1$ and $M2$ (both have *latency*=3). $M3$ consumes the $M1$'s results; a join (+) consumes tokens produced by $M2$ and $M3$. Figure 1b describes the shared resource (shared by $M2$ and $M3$) with a sharing wrapper: A merge and a mux select the inputs of the original units; the selected operands are sent to the shared unit. The merge informs the *condition buffer* which input has been taken; this information will be used by *Branch* to send the result to the correct successor. There is a buffer slot at each output of *Branch* to account for the temporary unavailability of the successors. This design has a deadlock risk, as described next.

Example: Naive resource sharing creates deadlock. The circuit in Figure 1b may deadlock due to *head-of-line blocking* [15, 22, 29]. Initially, the shared unit executes $M2$ twice before $M3$. The resulting state (in Figure 1b) has an execution dependency cycle: ▪ The first $M2$-produced token occupies M2's output buffer, which prevents the shared unit from sending out the token closest to its output (i.e., the head-of-line position). ▪ The token at the head of the line blocks the token after it, i.e., the first token of $M3$. ▪ The first token of $M3$ cannot reach *Join*; since *Join* needs both tokens from $M2$ and $M3$ to execute, it cannot consume the token that occupies M2's output buffer. The circuit deadlocks since no token can move.

Prior work: a total-order-based approach. Josipović et al. [22] propose a solution to this deadlock problem. Consider Figure 1b: There is no deadlock risks if we always access the shared unit in the control flow order, i.e., operations of one *basic block* (BB) must execute before the operations of another. In this example, $M2$ and $M3$ of iteration 1 must be executed before $M2$ and $M3$ in any subsequent iterations. For example, "$M2, M3, M2, M3$, etc." is a legal order, as no access runs ahead of the execution of the previous iteration. In Figure 1b, "$M2, M2, etc$." violates the total order.

In the last example, deadlock is avoided without an II penalty; the II remains 2—the best II when a resource is shared between 2 operations. Yet, the same example also shows a performance limitation of this strategy. Consider the same circuit, except we now share $M1$ and $M3$ with an order "$M1, M3$, etc.". We have the schedule in Figure 2a: ▪ cycle 1: $M1$ starts, ▪ cycle 4: $M3$ receives data from M1, and starts, ▪ cycle 5: $M1$ starts, ▪ cycle 8: $M3$ receives data from M1, etc. In this case, every $M1$ (starting from iteration 2) has to wait until the $M3$ in the previous iteration begins; this creates an execution dependency cycle of latency 4 (the entire execution of $M1$, the first stage of $M3$, and back to $M1$), which forces the achievable II to be at least 4 (greater than the theoretically achievable II).

Besides limited cases where sharing is possible, a total token order also postpones the token traversal, which requires more buffers

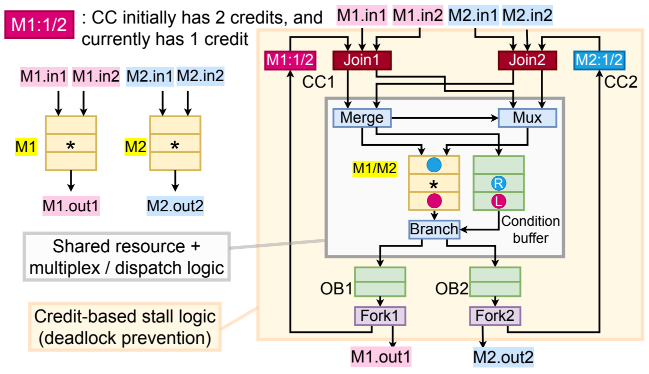

M1:1/2
: CC initially has 2 credits, and currently has 1 credit
M1.in1 M1.in2 M2.in1 M2.in2
M1
M2
*
M1.out1
M2.out2
Shared resource + multiplex / dispatch logic
Credit-based stall logic (deadlock prevention)
Join1
Join2
M2:1/2
CC1
CC2
Merge
Mux
M1/M2
R
L
Branch
Condition buffer
OB1
OB2
Fork1
Fork2

Figure 3: Credit-based sharing wrapper for $M1$ and $M2$.

than the circuit without sharing. Implementing a total token order is also challenging in some dataflow HLS strategies [17].

Our work: a credit-based approach. To overcome these limitations, we devise a *credit-based* sharing strategy inspired by techniques well-known in interconnect systems [8, 9, 15, 29]. (1) Our strategy seizes opportunities like in Figure 2a by allowing out-of-order access to the shared unit, thus achieving the schedule like in Figure 2b. (2) Our strategy postpones the operation for a shorter duration, which makes our buffer requirement smaller. (3) Our deadlock avoidance strategy is agnostic to the circuit's control flow; thus, it is suitable for different dataflow HLS strategies [17, 20].

The rest of the paper is organized as follows: Section 4 describes our sharing strategy and how it ensures deadlock-free resource-sharing. Section 5 discusses the performance aspect of sharing and presents efficient heuristics for reducing resource usage and maintaining the II. Section 6 presents a complete workflow and evaluates our approach. Section 7 concludes the paper.

4 DEADLOCK-FREE RESOURCE SHARING: A CREDIT-BASED APPROACH

This section presents our credit-based sharing approach for preventing deadlocks.

We assume that the circuit before sharing is deadlock-free per se; this work focuses on avoiding the deadlock situations introduced by resource sharing. We classify the sharing-induced deadlocks into two types: ■ *Head-of-line blocking*: The interaction between the sharing wrapper and its successors might create a deadlock (elaborated in Section 3). We avoid it with a credit-based stall mechanism. ■ *Fixed access order policy*: The interaction between the sharing wrapper and its predecessors might create a deadlock. We avoid it with an adaptive arbitration scheme with priority.

4.1 Avoiding Deadlock Caused by Head-of-Line Blocking

This section describes how we eliminate head-of-line blocking using a credit-based approach [15, 29].

We extend the naive sharing wrapper (Figure 1b) with a credit-based control mechanism, as shown in Figure 1c. At the inputs of the sharing wrapper (i.e., before the merge and mux), each operation that shares the unit maintains the number of available *credits*—the number of computations it can issue to the shared resource. Initially, the number of credits must be no greater than the number of output buffer slots. A computation is issued by consuming 1 credit; whenever an operation has all its input data but no credit, the wrapper prevents access by stalling the operation's predecessor(s). Consider Figure 1c: both $M2$ and $M3$ initially have 1 credit (the same as the output buffer slots); 0/1 means initially the operation has 1 credit, but currently it has 0 credits. When the sharing wrapper has issued an $M2$ (left) and its result has not yet left, $M2$ will be out of credits, which stops the sharing wrapper from issuing any $M2$; meanwhile, the wrapper can issue an $M3$ (right) as there is 1 available credit. Whenever a token leaves the output buffer, a credit is returned to the input; this indicates that an output buffer slot becomes free. In this way, at any point in time, each token in the shared resource can always find a free slot at its destination output buffer, and the token at the head of the line can never be stalled, thus, the deadlock risk due to head-of-line blocking is eliminated.

Irrelevant to correctness, sufficient credits are necessary for maintaining the II. Consider Figure 1c: The shared unit will be underutilized when $M2$ and $M3$ each have only 1 credit (at most 2 out of 3 pipeline stages can be used). In Section 5.4, we will discuss how to allocate sufficient credits to maintain the desired II.

4.2 Avoiding Deadlock Caused by a Fixed Access Order

The sharing wrapper must have an access policy to handle situations where multiple operations can be executed. One might attempt to use a round-robin style arbiter to control the access, i.e., an arbiter that strictly follows a predefined order to execute the operations. However, this might create a deadlock if we do not know the dependencies between the input operations.

Example: a fixed access order causes deadlock. Consider Figure 1d; we share operations $M1$ and $M3$, where $M3$ needs the result of $M1$ to execute. Assuming that the arbiter follows a fixed order "$M3$, $M1$, $M3$, $M1$", the circuit deadlocks as ■ the first request of $M3$ will not be issued before the first $M1$ is available, and ■ the first $M1$ cannot be executed as it has to wait for the first $M3$.

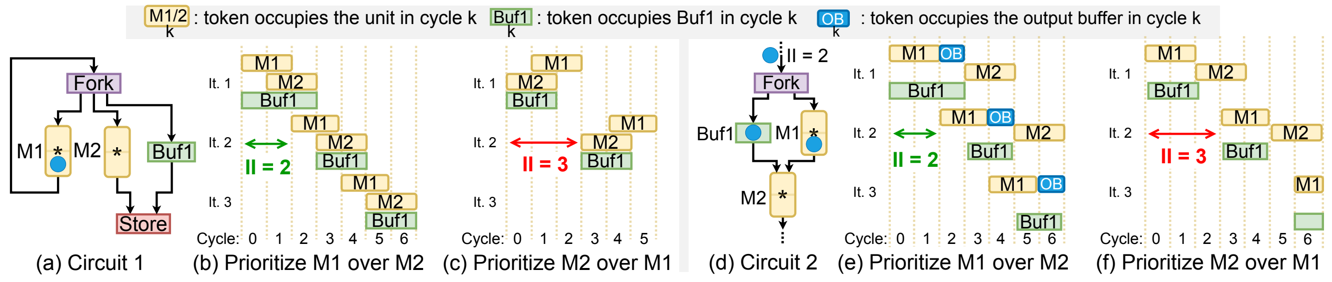
We do not allow an absent request keeping any other request out of the shared resource. This can be realized through an arbiter with a priority. Consider Figure 1e. In this case, unlike in Figure 1d where a fixed access order is used, an arbiter decides which operation can run based on a predefined priority (e.g., in this case, $M3$ is prioritized over $M1$); the key difference compared with fixed order is that $M1$ can execute in the absence of $M3$.

In this way, sharing will not create any dependency between the predecessors of different operations that share the unit, thus avoiding deadlock caused by the interaction between the sharing wrapper and its predecessors.

Irrelevant to correctness, not all access priorities maintain the II. Section 5.3 discusses how to determine one that does.

4.3 A Sharing Mechanism for Dataflow Circuits

This section leverages the insights from Section 4.1 and 4.2 to devise a strategy to construct circuits with sharing. We denote



M1/2 k : token occupies the unit in cycle k
Buf1 k : token occupies Buf1 in cycle k
OB k : token occupies the output buffer in cycle k
Fork
M1
M2
Buf1
Store
It. 1
It. 2
It. 3
II = 2
II = 3
Cycle: 0 1 2 3 4 5 6
(a) Circuit 1
(b) Prioritize M1 over M2
(c) Prioritize M2 over M1
(d) Circuit 2
(e) Prioritize M1 over M2
(f) Prioritize M2 over M1

Figure 4: Two examples show that access priority may penalize circuit II. Both circuits in Figure 4a and 4d: M1 and M2 are shared; M2 depends on M1. Figure 4b and 4e: respective schedules when $M1 \prec M2$; II is maintained. Figure 4c and 4f: schedules when $M2 \prec M1$; II is penalized.

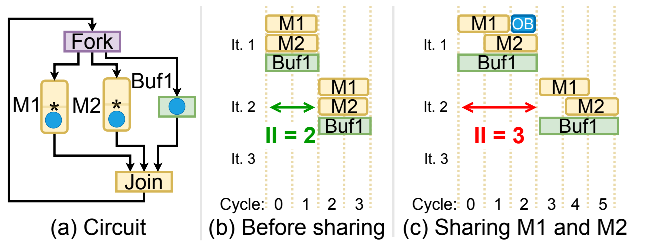


Figure 5: M1 and M2 are in the same SCC and are always executed simultaneously before sharing. No access priority maintains the II.

the set of operations that share one unit as a *sharing group* $G := \{op_1, op_2, \ldots, op_{|G|}\}$; the group size is denoted as $|G|$.

Consider Figure 3: a unit is shared between $|G| = 2$ operations using our credit-based method. The inner part of the sharing wrapper with the caption "*shared resource+...*" is the same as in Figure 1b (discussed in Section 2.2). The wrapper consists of a mux, an arbiter merge, a shared unit, a condition buffer, and a branch. We extend it as follows: for each operation op_i, a *credit counter* CC_i is used to track the number of computations that can be issued to the shared unit. CC_i resets with $N_{CC,i}$ credits (implemented as dataless tokens); here, $N_{CC,i} = 2$. After each output channel i of the branch, there is a buffer (*output buffer* (OB_i)) that has $N_{OB,i}$ slots (here $N_{OB,i} = 2$)—it holds tokens dispatched from the shared unit, but cannot be taken yet by the output (e.g., see Figure 1). As described in Section 4.1, to avoid deadlock, $N_{CC,i} \leq N_{OB,i}, \forall i$. Here CC_1 has two initial credits, therefore, OB_1 must have at least two slots.

Before the merge and muxes, a join Join_i synchronizes op_i's credit and operands from op_i's predecessors and CC_i. When op_i has an available credit and all the operands, Join_i, the merge, and muxes will consume a credit and start a computation in the shared unit; whenever the operands are incomplete or have no credits, Join_i prevents op_i access by stalling the predecessors and CC_i.

When a token produced by op_i leaves the sharing wrapper, a fork Fork_i returns a credit to CC_i. The fork must be lazy [13], i.e., it propagates tokens to the successors only if both are ready, which prevents a credit from being returned before the slot is freed. To avoid combinational loops, it takes 1 clock cycle to return the credit, i.e., in the same cycle, the credit being returned cannot be used for starting a new computation.

Our solution has area and timing overheads, e.g., buffers, credit counters, and joins. Yet, it is a general solution to avoid deadlock introduced by sharing. Section 6 will show that the overheads are insignificant.

5 SHARING AND PERFORMANCE

This section devises a strategy that determines sharing groups; an access priority is then decided for each group. Our goal is to use as few groups as possible while maintaining the II of the *performance-critical CFCs* (e.g., the innermost loop of each loop nest). We will (1) describe cases where sharing increases II and (2) present heuristics that generate performance-preserving grouping and priority schemes. Our sharing strategy differentiates from total-order-based sharing since it is based on priority [22].

5.1 Examples: Sharing Might Penalize II

This section discusses representative examples of when sharing hurts II: We discuss (1) cases where a good priority must be selected, then discuss (2) cases where none of the access priorities preserves the II, i.e., some operations should not be shared.

Example 1: access priorities. Figure 4a describes a CFC; operations $M1$ and $M2$ are shared. Before sharing, the II is 2; the average occupancies Φ_{M1}, Φ_{M2} of both $M1$ and $M2$ are 1 (defined in Section 2.1), therefore, sharing might be desirable. As they are always enabled simultaneously, arbitration is needed. We denote op_a *precedes* op_b with $op_a \prec op_b$. Figure 4b depicts the schedule if we consider $M1 \prec M2$ (i.e., $M1$ is prioritized); this is desirable as $II = 2$. Instead, if $M2 \prec M1$ (Figure 4c), a token has to wait for $M2$ for 1 cycle, and stays in $M1$ for 2 cycles; since a new iteration can start only once $M1$ produces a value, the II becomes 3.

Example 2: access priorities. Figure 4d depicts a subcircuit receiving a new token with an $II = 2$; sharing $M1$ and $M2$ might be desirable. $M1 \prec M2$ is maintains the II (Figure 4e). On the other hand, if $M2 \prec M1$ (Figure 4f), since $M1$ and $M2$ can execute simultaneously every 2 cycles, postponing $M1$ also delays the subsequent executions of $M2$, i.e., $M2$ can no longer run with an $II = 2$.

In both examples, $M2$ needs the result from $M1$. In Figure 4a, $M2$ needs the result from $M1$ in the previous iteration. In Figure 4d, $M2$ depends on $M1$ in the same iteration. The priority $M2 \prec M1$ in both examples ignores the dependency between the shared operations. To avoid a performance penalty, the priority should follow the data dependency: when sharing op_1 and op_2, if op_2 needs the result from op_1, then op_1 should be prioritized.

Algorithm 1: Determining sharing groups

```
Data: Dataflow circuits, performance-critical CFCs, occupancies of
      operations, sharing targets.
Result: Sharing groups groups.
1  /* For K sharing targets, initialize groups with K groups,
      each has 1 operation.                                  */
2  $groups \leftarrow \{\{op\} | \forall op \in sharing\ targets\}$
3  /* Greadily merge groups until no changes can be made.    */
4  while groups modified do
5      for $G_i, G_j \in groups$ do
6          if $\neg check\_R1(G_i \cup G_j)$ then
7              break        /* R1: ops must have the same type. */
8          if $\exists CFC \in Critical\ CFCs : \neg check\_R2(G_i \cup G_j, CFC)$ then
9              break      /* R2: sum of occupancy ≤ capacity. */
10         if $\exists CFC \in Critical\ CFCs : \neg check\_R3(G_i \cup G_j, CFC)$ then
11             break    /* R3: ops in the same CFC → ops always
                  start at different time.  */
12         $G_i \leftarrow G_i \cup G_j, G_j \leftarrow \{\}$          /* Merge $G_i$ and $G_j$. */
```

Algorithm 2: Determine the group access priority (based on bubble sort)

```
Data: Dataflow circuit, performance-critical CFCs, sharing group G.
Result: Group G's access priority G.prio.
1  $G.prio = [op_1, op_2, \dots, op_{|G|}]$               /* Priority as a list */
2  while G.prio modified do
3      for $i \in 2 \dots |G|$ do
4          for $CFC \in CriticalCFC$ do
5              $SCCG_{CFC} = getSCCGraphOfCFC(CFC)$
6              /* If ops are in different SCCs of the same CFC, we
                  decide on the priority based on the topological
                  order of the SCC graph ($SCCG_{CFC}$).          */
7              if $G.prio[i-1], G.prio[i] \in CFC$ then
8                  if $\exists SCC_i, SCC_j \in SCCG_{CFC} : G.prio[i-1] \in$
                    $SCC_i, G.prio[i] \in SCC_j$ then
9                      $getTopologicalOrder(SCCG_{CFC})$
10                     if $SCC_i.order > SCC_j.order$ then
11                         $G.prio.swap(i-1, i)$
```

The next example describes a case where the II is penalized no matter which access priority we choose—this indicates that operations should not be shared.

Example 3: sharing groups. Finding a performance-preserving grouping scheme seems straightforward—we want underutilized operations to share one unit [22]. Yet, there is still one caveat. Consider Figure 5a: $M1$ and $M2$ are shared (both have latency=2). Picking any of them to start first delays the join's execution and the II becomes 3 instead of 2 (schedule in Figure 5c). Here, both $M1$ and $M2$ need each other's results and they always become available to execute simultaneously; in other words, $M1$ and $M2$ are in the same *strongly connected component* (SCC)—a subgraph of nodes in which every node is reachable by every other node [31].

The three examples above show that reasoning about dependencies is critical to making decisions about groups and priorities. We next devise heuristics for generating the groups and priority. Both heuristics rely on analyzing the SCCs in the CFCs.

5.2 Sharing Group Heuristic

We here devise a heuristic for sharing group generation. We aim to preserve the II of each performance-critical CFC.

Each CFC has a set of SCCs. For each CFC, we can build an *SCC graph*—a directed graph that describes the dependencies between the SCCs, where the nodes are SCCs, and an edge $SCC_i \longrightarrow SCC_j$ represents that there exists an edge $n_i \longrightarrow n_j \in CFC$ such that $n_i \in SCC_i, n_j \in SCC_j$; this edge also implies that operations in SCC_j need the data from operations in SCC_i.

The sharing heuristic is depicted in Algorithm 1—the goal is to prevent sharing operations that are ■ in the same SCC and ■ always execute simultaneously. For K possible sharing targets, we initialize K groups (each has 1 operation). Given 2 groups G_i, G_j we test the following rules on their union $G = G_i \cup G_j$; G_i, G_j are merged if no rule has failed: ■ R1: Operations in G must have the same type. ■ R2: For each performance-critical CFC CFC, the sum of token occupancies in all operations in $G \cap CFC$ must be lower than the unit capacity. ■ R3: For each performance critical CFC CFC and each $SCC \in CFC$, if $\exists op_i, op_j \in G \cap SCC$, then $\forall u \in SCC \setminus \{op_i, op_j\}$, u must have different maximum distances to op_i and op_j. The heuristic merges groups until no change can be made.

R3 avoids arbitration between operations in the same SCC, thus avoiding II penalty. Consider Figure 5a: $M1$ and $M2$ are in the same SCC; the maximum distance from any other unit to $M1$ and $M2$ is always identical, e.g., the distances between Buf1 to $M1$ and $M2$ are both 0; by R3, $M1$ and $M2$ should not be in the same group. R3 is a heuristic: if the depicted SCC is not the CFC's bottleneck, sharing $M1$ and $M2$ does not penalize the II; we opt for this rule since it avoids complex analysis of the actual change of the II.

5.3 Access Priority Heuristic

This section devises an access priority heuristic; for each sharing group (determined using the sharing group heuristic), our heuristic assigns an access priority scheme to it.

The heuristic is depicted in Algorithm 2; it is based on applying bubble sort to a list that encodes group G's access priority. The following details how the heuristic compares the given 2 list elements, i.e., a pair of operations op_i, op_j in G. For each CFC CFC and the SCC graph of CFC, we determine a topological order of the SCCs. For any pair of SCCs $SCC_i, SCC_j \in CFC$, and $op_i, op_j \in G$ such that $op_i \in SCC_i$ and $op_j \in SCC_j$, the access priority between op_i and op_j must follow the topological order of SCC_i and SCC_j in the SCC graph. If op_i, op_j are in the same SCC, then any priority between op_i, op_j is accepted.

Example: priority based on SCC graph. Consider Figure 4a: the CFC has 4 SCCs, i.e., $SCC_0 = \{Fork, M1\}, SCC_1 = \{M2\}, SCC_2 = \{Buf1\}, SCC_3 = \{Store\}$. As $M2$ and $Buf1$ depend on the result of $Fork$ and $M1$, and $Store$ needs the result from both $M2$ and Buf; a possible topological order is $SCC_0 \prec SCC_1 \prec SCC_2 \prec SCC_3$. When sharing $M1$ and $M2$, since in the topological order $SCC_0 \prec SCC_1$, we order $M1 \prec M2$ to avoid performance loss. Note that this ordering scheme is not necessary to prevent performance loss. Consider Figure 4a: if there was a unit before M2 that introduces one sequential delay, then order $M2 \prec M1$ does not hurt the II, as $M1$ and $M2$ never execute at the same time.

In line with practical HLS strategies [5, 36], our grouping and priority strategies are based on heuristics. Our heuristics rely on standard, scalable graph analysis techniques, i.e., determining SCCs in a CFC (scales linearly to the CFC size [31]) and maximum distances in the SCCs (the SCCs are usually sparsely connected, i.e., the

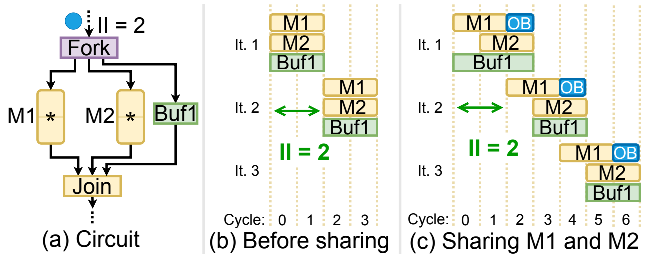


Figure 6: Sharing operations on reconvergent paths does not increase buffer usage. Before and after sharing, *Buf1* only needs 1 slot despite the latencies on other reconvergent paths having increased.

number of paths is small). In Section 6.2, we show that our strategy can produce efficient circuits with scalable runtime.

5.4 Buffer Sizing and Credit Allocation

We here describe how to decide on buffer and credit sizing.

No additional buffer requirement. On reconvergent paths—paths start and end at the same fork and join—buffers are inserted on the short-latency path to avoid the propagation of stalls [2, 4, 24, 30]. Arbitration may increase the latency on one or more paths, which generally requires larger buffers on the other paths. Using a fixed token order, this latency might be arbitrarily large depending on the circuit topology. But this is not the case when the latency is introduced by priority-based arbitration: in the steady-state, for any operation op in a sharing group G of size $|G|$, the maximum time that a op is postponed is $|G| - 1$, i.e., it has to wait for all other operations before it can get access. Since $II \geq |G|$, a unit has to be postponed at most $II - 1$ cycles. This allows the tokens to stay in the buffers for the same number of cycles, thus the II is maintained without resizing the buffers. Consider a circuit after sharing: for every pair of reconvergent paths, the eager fork at the beginning of the path observes a valid token every II cycles; this means, from the cycle that a new token is available, postponing the consumption of that token for $II - 1$ cycle does not block the next token from coming after II cycles. Thus, no additional buffer is required as the backward stall does not prevent new tokens from arriving.

Consider Figure 6a: we share $M1$ and $M2$, both have latency = 2. To balance token occupancy on the three paths, a 1-slot buffer (*Buf1*) is placed on the right path; this prevents stalling the Fork. Figure 6b and 6c describe the schedule without and with sharing ($M1 \prec M2$). Compared with the unshared circuit, the execution of $M2$ is always delayed for one cycle due to arbitration. The cycles where *Buf1* takes tokens are also delayed by 1 since the time it can dispatch the token out to the *Join* is delayed, therefore the time it can receive a token is also delayed. In the steady state (starting from iteration 2), a token stays in *Buf1* for 2 cycles, thus, no need to resize it.

Credit sizing requirements. Credits must be sufficiently allocated to avoid an II penalty, yet naively assigning many credits incurs a high output buffer cost [24]. For operation op, the initial number of credits $N_{CC,op}$ is the maximum of the sum of the number of tokens ▪ carrying the intermediate results of op (i.e., inside the shared unit), and ▪ staying in the output buffers of op (i.e., due to temporary unavailability of the successor caused by sharing); insufficient credits limit the number of simultaneous computations, thus the desired II cannot be achieved. For correctness, we must have $N_{OB,op} \geq N_{CC,op}$, where $N_{OB,op}$ is the number of op's output buffer slots (see Section 4.3).

Credit allocation rule. For each operation op in any sharing group with occupancy Φ_{op}, we assign $N_{CC,op} = \Phi_{op} + 1$ credits to sustain the II; Φ_{op} credits keep the shared unit fully utilized; 1 more credit (1) hides the latency of returning the credit (see Section 4.3), and (2) accounts for the token that occupies op's output buffer.

The interaction between operations might cause the output buffers to be occupied. Consider Figure 6c: After processed by $M1$ (see cycle 2), the token has to stay in the output buffer (see OB in Figure 3) while waiting for $M2$ to get access to the shared resource. In this way, the credit corresponding to that token is not returned to the credit counter for $M1$; if there were initially only 1 credit (the same as average occupancy Φ_{op}), $M1$ could not start again since there would be no available credits. In a steady state, there is *at most* one token staying in the output buffer, as a token remains in the buffers for at most $|G| - 1$ cycles due to arbitration, i.e., the token will already leave the output buffer when the next token comes; this justifies that $\Phi_{op} + 1$ credits are sufficient.

This concludes the description of our sharing strategy—we will evaluate its effectiveness in the following section.

6 EVALUATION

This section evaluates the effectiveness of our strategy in sharing functional units in dataflow circuits. Our research artifact (code and benchmarks) will be publicly available.

6.1 Methodology

To show the effectiveness of our strategy, we aim to minimize DSP usage without penalizing performance. To achieve this, we target sharing the floating-point arithmetic units realized using DSPs. Our strategy is also applicable to other functional units.

We employ *Dynamatic* [18, 21] to convert C code to a dataflow circuit and place buffers to optimize the circuit throughput and frequency. After obtaining dataflow circuits, we apply the heuristics described in Section 5.2 and 5.3 to determine the sharing groups and the access priority within each group. We then apply the sharing strategy in Section 4.3. We determine the number of credits and output buffer sizes according to the occupancy in the performance analysis result (see Section 5.4).

We use ModelSim [26] to (1) obtain the execution latency in clock cycle count and (2) verify the functionality, i.e., to confirm that the circuit produces the same result as the C code and the circuit never deadlocks. We use Vivado (2019.1) [33] to obtain the post-place-and-route area and maximum frequency, targeting a Kintex-7 FPGA with a clock period (CP) of 6 ns.

We evaluate our method on different dataflow HLS strategies [17, 20]. ▪ For a circuit generation strategy that organizes units into BBs (i.e., the standard Dynamatic flow [20, 21]), we contrast our method with a total-order-based sharing strategy [22]. The results are reported in Table 1. ▪ For a circuit generation strategy that organizes units into producer-consumer pairs [17], we show that our sharing strategy is equally effective; we note that in-order-based sharing [22] does not apply here. The results are reported in Table 2. For fairness, we use the same MILP formulation for the performance

Benchmark	Technique	Functional units	DSPs	Slices	LUTs	FFs	CP (ns)	Cycles	Exec. time (us)	Opt. time (s)
atax	**Naive**	2 fadd 2 fmul	10	675	1695	2028	5.0	4546	22.7	11.1
	In-order	1 fadd 1 fmul	5	700	1936	1806	5.4	4544	24.5	50.6
	CRUSH	1 fadd 1 fmul	5	644	1809	1619	5.1	4546	23.2	14.7
bicg	**Naive**	2 fadd 2 fmul	10	603	1604	1869	5.0	8815	44.1	0.8
	In-order	1 fadd 1 fmul	5	682	1750	1737	5.3	8787	46.6	23.7
	CRUSH	1 fadd 1 fmul	5	619	1734	1444	5.3	8758	46.4	1.6
gsum	**Naive**	5 fadd 4 fmul	22	831	2245	2910	5.9	3242	19.1	0.6
	In-order	3 fadd 2 fmul	12	814	2426	2557	5.8	3727	21.6	10.8
	CRUSH	1 fadd 1 fmul	5	593	1714	1529	5.9	3242	19.1	1.3
gsumif	**Naive**	7 fadd 4 fmul	26	1120	2965	3848	5.9	3093	18.2	2.1
	In-order	2 fadd 1 fmul	7	836	2691	2503	6.1	3573	21.8	120.4
	CRUSH	1 fadd 1 fmul	5	731	2272	1945	6.6	3093	20.4	3.1
2mm	**Naive**	2 fadd 4 fmul	16	1595	3742	4119	5.2	18205	94.7	147.5
	In-order	1 fadd 1 fmul	5	1495	3614	3614	5.6	18233	102.1	740.9
	CRUSH	1 fadd 1 fmul	5	1503	3690	3391	5.6	18204	101.9	154.2
3mm	**Naive**	3 fadd 3 fmul	15	1455	3423	3413	5.4	27883	150.6	40.3
	In-order	1 fadd 1 fmul	5	1180	3181	2753	5.4	27328	147.6	411.4
	CRUSH	1 fadd 1 fmul	5	1108	3189	2465	5.3	27289	144.6	50.6
symm	**Naive**	4 fadd 7 fmul	29	1911	4880	5223	6.0	38707	232.2	120.9
	In-order	1 fadd 1 fmul	5	1658	4476	3691	7.1	45313	321.7	1373.4
	CRUSH	1 fadd 1 fmul	5	1738	4618	3419	6.0	38337	230.0	121.7
gemm	**Naive**	1 fadd 3 fmul	11	846	2226	2539	5.5	76434	420.4	120.6
	In-order	1 fadd 1 fmul	5	857	2355	2391	5.4	76433	412.7	488.4
	CRUSH	1 fadd 1 fmul	5	839	2298	2285	5.4	76433	412.7	121.3
gesummv	**Naive**	3 fadd 4 fmul	18	970	2653	3098	5.4	8800	47.5	2.1
	In-order	1 fadd 1 fmul	5	928	2419	2548	5.3	9267	49.1	146.4
	CRUSH	1 fadd 1 fmul	5	889	2515	2068	5.7	8773	50.0	2.6
mvt	**Naive**	2 fadd 2 fmul	10	767	1889	2122	5.0	17524	87.6	1.8
	In-order	1 fadd 1 fmul	5	809	2156	1921	5.3	17487	92.7	7.7
	CRUSH	1 fadd 1 fmul	5	758	2079	1697	5.2	17467	90.8	2.6
syr2k	**Naive**	2 fadd 5 fmul	19	1474	3362	3769	5.2	17649	91.8	121.3
	In-order	1 fadd 1 fmul	5	1253	3388	3163	5.6	18245	102.2	625.6
	CRUSH	1 fadd 1 fmul	5	1278	3590	2853	5.8	17352	100.6	122.0
Geomean improvement of CRUSH vs Naive. Slices: -12% LUTs: -4% FFs: -29% DSPs: -68% Opt. time (s): +31% Exec. time (us): +3%										
Geomean improvement of CRUSH vs In-order. Slices: -7% LUTs: -4% FFs: -15% DSPs: -10% Opt. time (s): -89% Exec. time (us): -6%										

Table 1: Comparison between no sharing (Naive) [24], total-order-based sharing (In-order) [22], and our work (CRUSH). Compared with In-order, CRUSH seizes more sharing opportunities and has significant runtime savings; this is achieved at negligible performance degradation.

Benchmark	Technique	Functional units	DSPs	Slices	LUTs	FFs	CP (ns)	Cycles	Exec. time (us)	Opt. time (s)
atax	**Fast token**	2 fadd 2 fmul	10	887	2347	2573	5.7	3595	20.5	3.9
	CRUSH	1 fadd 1 fmul	5	936	2488	2158	5.7	3520	20.1	4.7
bicg	**Fast token**	2 fadd 2 fmul	10	723	1951	2183	5.6	8003	44.8	1.2
	CRUSH	1 fadd 1 fmul	5	723	2044	1758	5.8	8309	48.2	1.7
gsum	**Fast token**	5 fadd 4 fmul	22	918	2293	2956	6.0	3233	19.4	0.5
	CRUSH	1 fadd 1 fmul	5	582	1728	1525	5.9	3233	19.1	1.1
gsumif	**Fast token**	7 fadd 4 fmul	26	1152	2954	3609	6.5	2312	15.0	0.7
	CRUSH	1 fadd 1 fmul	5	756	2323	1657	7.1	2312	16.4	1.4
2mm	**Fast token**	2 fadd 4 fmul	16	1962	4372	4803	5.6	8622	48.3	26.9
	CRUSH	1 fadd 1 fmul	5	1822	4481	4075	5.7	8235	46.9	28.0
3mm	**Fast token**	3 fadd 3 fmul	15	2234	5352	5487	5.8	8625	50.0	20.8
	CRUSH	1 fadd 1 fmul	5	2176	5491	4499	5.8	8235	47.8	23.2
symm	**Fast token**	4 fadd 7 fmul	29	2132	5493	5501	5.9	35580	209.9	120.3
	CRUSH	1 fadd 1 fmul	5	1792	5142	3692	6.0	34580	207.5	121.0
gemm	**Fast token**	1 fadd 3 fmul	11	1023	2800	2944	5.6	69233	387.7	63.9
	CRUSH	1 fadd 1 fmul	5	1040	2861	2690	5.8	68834	399.2	54.3
gesummv	**Fast token**	3 fadd 4 fmul	18	969	2623	3162	5.4	8016	43.3	1.6
	CRUSH	1 fadd 1 fmul	5	847	2465	2132	5.6	7931	44.4	2.0
mvt	**Fast token**	2 fadd 2 fmul	10	988	2648	2756	5.6	8005	44.8	1.1
	CRUSH	1 fadd 1 fmul	5	990	2831	2331	5.5	7890	43.4	2.0
syr2k	**Fast token**	2 fadd 5 fmul	19	1769	4390	4499	5.7	16444	93.7	121.8
	CRUSH	1 fadd 1 fmul	5	1637	4530	3581	5.6	16346	91.5	122.5
Geomean improvement of CRUSH vs Fast token. Slices: -11% LUTs: -3% FFs: -27% DSPs: -68% Opt. time (s): +29% Exec. time (us): +0%										

Table 2: Comparison between Fast token (a more recent dataflow HLS strategy [17]) and the same fast-token circuit optimized using CRUSH. Since the circuit does not have the notion of BBs, the total-order-based sharing solution [22] does not apply here; this shows that our strategy is general. The results show that CRUSH is effective on different dataflow HLS approaches.

analysis and optimization, with the same solver (Gurobi 10.0.3) and a timeout of 2 min.

We consider benchmarks that exhibit different computation patterns and loop properties, which have (1) a selection of kernels in PolyBench [28] (*atax*, *bicg*, *2mm*, *3mm*, *symm*, *gemm*, *gesummv*, *mvt*, and *syr2k*); (2) *gsum* and *gsumif*, which have irregular computation patterns that are often used to showcase the benefit of dynamic scheduling [10]. All the kernels have an $II > 1$ due to the long-latency loop-carried dependency floating-point operations, i.e., many units are underutilized and can be shared without a performance penalty.

6.2 Discussion: Effectiveness of Our Strategy

Table 1 reports the utilization, performance, and optimization runtime of different approaches. The column **Technique** categorizes the used sharing strategy: ■ **Naive**: no resource sharing [24], ■ **In-order**: optimized using total-order-based sharing [22], ■ **CRUSH**: optimized using our credit-based sharing strategy. The columns **Functional units** and **DSPs** respectively report the functional unit count in the dataflow circuit and the required number of DSPs by the synthesized kernel—they indicate the effectiveness of the sharing strategies.

CRUSH effectively reduces the functional unit usage—all the functional units with identical types can be shared with one unit. This is achieved at a negligible performance degradation, as reported in **Exec. time (us)** (calculated as CP × Cycles). Compared with **Naive**, **CRUSH** has a geomean of 3% performance loss (mainly due to the CP overhead). This shows that our heuristics successfully maintain the performance (see Section 5.2 and 5.3).

Compared with **Naive**, occasionally **CRUSH** has a lower clock cycle count. This effect is accidental—**Naive** and **CRUSH** are expected to have approximately the same clock cycle count. The difference occurs due to the units we use: our units have a single enable signal for the entire pipelined unit. The unit is stalled if the token at the head-of-the-line position cannot move out; our sharing strategy eliminates head-of-line blocking, which accidentally improves the performance.

CRUSH is more effective in sharing functional units compared with **In-order**—in *gsum* and *gsumif*, **CRUSH** seizes sharing opportunities because it permits out-of-order access to shared resource (see Section 3). The column **Opt. time (s)** reports the total optimization time (MILP + sharing); **CRUSH** has a runtime overhead compared with **Naive**, as **Naive** does no resource analysis; more importantly, **CRUSH** has significantly better runtime than **In-order**, which requires repetitively solving the MILP formulation to evaluate the effect of sharing.

6.3 Discussion: Resource Efficiency of Our Strategy

Our main goal is to reduce the DSP usage—we achieve this with a smaller overhead than the prior approach.

Columns **Slices**, **LUTs**, **FFs** indicate the FPGA resource utilization. When comparing **CRUSH** against **In-order**, we achieved a geomean reduction of 7% Slices, 4% LUTs, and 15% FFs; this shows that **CRUSH**'s sharing wrapper is more resource efficient compared with **In-order**. Sharing reduces the LUT/FF usage because the shared floating-point units contain ■ pipeline registers (outside the DSPs) and ■ additional logic implemented in LUTs [23]. Compared with **In-order**, **CRUSH** requires more output buffers, yet they can be efficiently implemented using LUTRAM [33]. Moreover, this overhead is compensated by the savings from two other factors: ■ **CRUSH** does not rely on the global control-flow ordering to avoid deadlocks, thus the expensive logic for tracking the control-flow ordering is not needed; ■ **In-order** postpones the operations more than **CRUSH**, thus **In-order** produced circuits might need larger buffers to preserve the II; since **CRUSH** postpones taking tokens for at most $II - 1$ cycles, the original buffers are sufficient (see Section 5.4).

The column **CP (ns)** indicates the achieved CP. Overall, sharing increases the critical path, since the sharing wrapper adds combinational logic (e.g., the input multiplexers); when many operations are sharing the same unit, as expected, the CP overhead becomes large (e.g., *gsumif*). This aspect is in parallel with our goal, i.e., reducing resource usage; it can be mitigated by enhancing Dynamatic's timing model to support sharing [30].

6.4 Discussion: Generality of Our Strategy

In the previous sections, our comparison baseline (**In-order**) is specialized for circuits whose units are organized into BBs, where sharing can be determined and regulated by BB ordering. Yet, this is not always the case: more recent dataflow HLS strategies omit BB organization for performance merits [17]—making total-order-based sharing challenging to implement. In contrast, our sharing strategy can be effortlessly ported to different dataflow HLS approaches; we demonstrate this by integrating **CRUSH** without any modifications into a recent fast token delivery approach [17].

Table 2 reports statistics of circuits produced by the fast token delivery approach without and with our resource-sharing method. Each table entry with **Fast token** denotes the circuit is generated using the fast token delivery approach, and **CRUSH** indicates that the circuit is additionally resource-optimized using our resource-sharing strategy. The result shows that we have achieved a geomean reduction of 68% DSPs, 27% FFs, and 11% slices; this is similar to the improvement in the other HLS strategy (Table 1) using **CRUSH** which, again, shows the generality of our approach.

7 CONCLUSION

Resource sharing has always been a challenging topic for dataflow-based systems due to the risk of deadlock. Existing solutions are inefficient in both the optimization runtime and the produced circuits; they are also limited to particular HLS strategies for dataflow circuits. We have presented CRUSH, an efficient resource sharing strategy for dataflow circuits. Our deadlock avoidance mechanism is modular, localized, and independent of the circuit's control mechanism, which makes it available to different HLS strategies for dataflow circuits. Experimental results show a geomean reduction of 15% FFs, 10% DSP usage, and 89% optimization runtime. All these benefits make HLS of dataflow circuits more attractive and practical.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] 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, CA, 775–82.

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

[3] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. 2005. Dataflow: A Complement to Superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. Austin, TX, 177–86.

[4] Dmitry Bufistov, Jordi Cortadella, Mike Kishinevsky, and Sachin Sapatnekar. 2007. A general model for performance optimization of sequential systems. In *Proceedings of the International Conference on Computer-Aided Design*. San Jose, CA, 362–69.

[5] Andrew Canis, 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.

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems* 13, 2 (Sept. 2013), 24:1–24:27.

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

[8] Satrajit Chatterjee and Michael Kishinevsky. 2012. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. *Formal Methods in System Design* 40 (2012), 147–69.

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

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

[11] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 43rd Design Automation Conference*. San Francisco, CA, 433–38.

[12] 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*. 149–58.

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

[14] Steve Dai, Gai Liu, and Zhiru Zhang. 2018. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 137–146.

[15] Willian James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks*. Elsevier.

[16] Stephen A. Edwards, Richard Townsend, and Martha A. Kim. 2017. Compositional Dataflow Circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. Vienna, 175–84.

[17] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2022. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 253–261.

[18] EPFL-LAP. 2023. Dynamatic. https://github.com/EPFL-LAP/dynamatic

[19] 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.

[20] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 127–36.

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

[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, Axel Marmet, Andrea Guerrieri, and Paolo Ienne. 2023. Resource Sharing in Dataflow Circuits. *ACM Transactions on Reconfigurable Technology and Systems* 16, 4 (Sept. 2023), 1–27.

[24] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, 186–96.

[25] 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

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

[27] 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.

[28] Louis-Noël Pouchet. 2012. *Polybench: The polyhedral benchmark suite*. http://www.cs.ucla.edu/pouchet/software/polybench

[29] Sayak Ray and Robert K. Brayton. 2012. Scalable progress verification in credit-based flow-control systems. In *Proceedings of the 2012 Design, Automation and Test in Europe Conference and Exhibition*. Dresden, Germany, 905–910.

[30] Carmine Rizzi, Andrea Guerrieri, Paolo Ienne, and Lana Josipović. 2022. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 375–383.

[31] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (June 1972), 146–160.

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

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

[34] Jiahui Xu and Lana Josipović. 2024. Suppressing Spurious Dynamism of Dataflow Circuits Via Latency and Occupancy Balancing. In *Proceedings of the 32nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 188–98.

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

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