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

Jiahui Xu
ETH Zurich
Zurich, Switzerland
jxu@ethz.ch

Lana Josipović
ETH Zurich
Zurich, Switzerland
ljosipovic@ethz.ch

Abstract

Dynamically scheduled high-level synthesis (HLS) automatically translates software code (e.g., C/C++) to dataflow circuits—networks of compute units that communicate via handshake signals. These signals schedule the circuit during runtime, allowing them to handle irregular control flow or unpredictable memory accesses efficiently, thus giving them performance merit over statically scheduled circuits produced by standard HLS tools.

To make HLS of dataflow circuits attractive and practical, we need various resource-optimization strategies to complement their performance advantage. A crucial technique is resource sharing: scarce and expensive resources (e.g., floating-point arithmetic units) are shared between multiple operations. However, this approach faces unique challenges in dataflow circuits, as an uncareful sharing strategy leads to performance degradation and circuit deadlock.

This work presents CRUSH, a strategy that enables efficient functional unit sharing in dynamically scheduled HLS. CRUSH systematically avoids sharing-introduced deadlocks: it decouples interactions of operations in the shared resource to break resource dependencies. CRUSH maintains the benefit of dynamism: it does not constrain circuit execution with a complex deadlock avoidance mechanism and seizes sharing opportunities enabled by out-of-order access to the shared unit. CRUSH is practical: it employs scalable and effective heuristics for sharing decisions. Compared to a prior strategy, CRUSH achieves an average reduction of 12% DSPs, 15% FFs, and 90% optimization runtime. CRUSH has been integrated into the Dynamatic HLS compiler (https://github.com/EPFL-LAP/dynamatic).

CCS Concepts: • **Hardware → Resource binding and sharing**; **Datapath optimization**; • **Computer systems organization → Data flow architectures**.

Keywords: Dataflow circuits; high-level synthesis; resource sharing

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707273

ACM Reference Format:
Jiahui Xu and Lana Josipović. 2025. CRUSH: A Credit-Based Approach for Functional Unit Sharing in Dynamically Scheduled HLS. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707273

1 Introduction

FPGAs are flexible and energy-efficient computing platforms that support fine-grained parallelism [8, 39]. Yet, they are also hard to program: expertise in low-level hardware details is necessary to achieve high design quality. *High-level synthesis* (HLS) tools promise to reduce the effort for FPGA design by automatically compiling software programming languages like C/C++ to RTL circuit descriptions [6, 19, 24, 45]. However, standard HLS strategies [6, 12, 44, 51] are limited to particular classes of applications (i.e., regular and predictable code that the HLS compiler can easily reason about) and fail in achieving the desired parallelism levels in others (i.e., code with unpredictable memory or control accesses that cannot be determined at compile time). In analogy with the processor world, standard HLS has a strong resemblance to VLIW processors [36]: to extract parallelism, it relies on complex compilation and code restructuring techniques that are successful only in certain situations. Yet, most applications today require the flexibility of out-of-order superscalar processors, where execution decisions are dynamically determined at runtime [32].

Dataflow circuits offer this benefit: they allow execution to progress dynamically as soon as data is available and critical decisions are resolved, and exploit similar mechanisms to those of out-of-order processors, such as memory reordering [28], speculation [30], and multithreaded execution [22]. Thus, recent HLS efforts generate dataflow circuits from software code [21, 29] and show impressive results. Yet, this performance advantage is not for free: dataflow circuits are also resource-expensive. To ensure that they can implement relevant large-scale applications on FPGAs, where resources are limited and, sometimes, scarce, dataflow circuits need to exploit any available resource optimization opportunity—from resource sharing [33] to redundant dataflow logic removal [48–50] and strategic buffering [41].

Resource sharing is a standard HLS optimization: the idea is to share physical units among multiple operations executing at different times [18]. In the context of FPGAs, it is typically used to share DSP blocks across multiple arithmetic operations; other sharing forms are also possible (e.g., BRAM or register sharing). Yet, resource sharing has its unique challenges 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 might create cyclic dependencies among operations, thus causing deadlock or performance degradation. Thus, producing correct and performant dataflow 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 [33]; both aspects limit the usability of sharing in complex HLS applications.

In this work, we present CRUSH, a scalable and effective sharing strategy in dataflow circuits. Our contributions are summarized as follows:

- We devise a simple and localized sharing mechanism for dataflow circuits. In contrast to a prior work [33], we seize more sharing opportunities by enabling out-of-order accesses to the shared unit. Our approach is independent of the control flow mechanism of the circuit, which makes it suitable for different dataflow HLS strategies [21, 29].
- We devise scalable and effective heuristics to decide on sharing schemes and access priorities that maintain the circuit's performance before sharing, without requiring an exhaustive and time-consuming evaluation of the effect of sharing.

We demonstrate the effectiveness of CRUSH on a set of standard HLS benchmarks and show its practicality in implementing sharing in complex HLS applications.

2 Background and Related Work

This section describes dataflow circuits and recent research studies 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 [3, 7, 14, 20, 29]. The communication between units is regulated by a handshake protocol: once the control and memory dependencies have been resolved, units exchange *tokens* which encapsulate data. There is no centralized scheduler: the units *dynamically* progress the execution as soon as all conditions to do so have been met. Many works study the generation process of dataflow circuits from high-level code [21, 29]; we focus on HLS approaches that target C code [21, 29] and the generated circuits implement single-threaded programs [21, 29, 35]; our solution is equally applicable to other dataflow HLS strategies.

The dataflow circuits that we consider are built from the following units [14, 20, 29]: (1) A *fork* has 1 input and multiple outputs; it distributes a copy of the incoming token to each successor as soon as they are ready to receive it. (2) 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. (3) A *merge* has 2 inputs and 1 output; it propagates a token to its single output from one of its inputs. (4) A *mux* is a merge with an additional control input to select the input token. (5) A *branch* has 1 data input, 1 control input, and 2 outputs; it propagates the received data token to one of its successors, depending on the value of the received condition token. (6) A *buffer* has 1 input and 1 output; it is used to store tokens, break combinational paths, and increase throughput; buffers can be arbitrarily placed on any channel without penalizing correctness [4, 29, 34].

Dataflow circuits are commonly modeled as a directed graph, where nodes are units and edges are channels. Performance optimization is commonly done on *choice-free circuits* (CFCs)—subcircuits with no conditional execution [2, 4, 34, 41]. The *initiation interval* (II) of the CFC is often the primary optimization goal, which indicates the distance between two consecutive loop iterations. A smaller II implies more loop computations can be done on average and, thus, higher circuit throughput. The *Token occupancy* denotes the number of tokens occupying a channel; it determines how many buffer slots should be allocated. The occupancy ϕ_{op} of a pipelined unit *op* is calculated as lat_{op}/II_{CFC}, where lat_{op} is *op*'s latency (number of cycles for computing the result), and II_{CFC} is the CFC's II. It is used to identify underutilized units and where it is advantageous to share them [33].

2.2 Resource Sharing in Dataflow Circuits

Resource sharing is one of the key HLS optimization techniques for generating efficient circuits [5, 6, 16, 44, 51]. Sharing is typically performed heuristically in conjunction with operation scheduling due to the computational complexity of this task [5, 18, 51].

Resource sharing in dataflow circuits has been addressed in limited scope [1, 13, 20, 26, 33, 37]. To the best of our knowledge, only Josipović et al. [33] present a solution that addresses both the correctness and performance aspects of sharing in HLS-produced dataflow circuits: (1) they point out that naive resource sharing leads to a deadlock, and (2) 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 limitations, and our new solution.

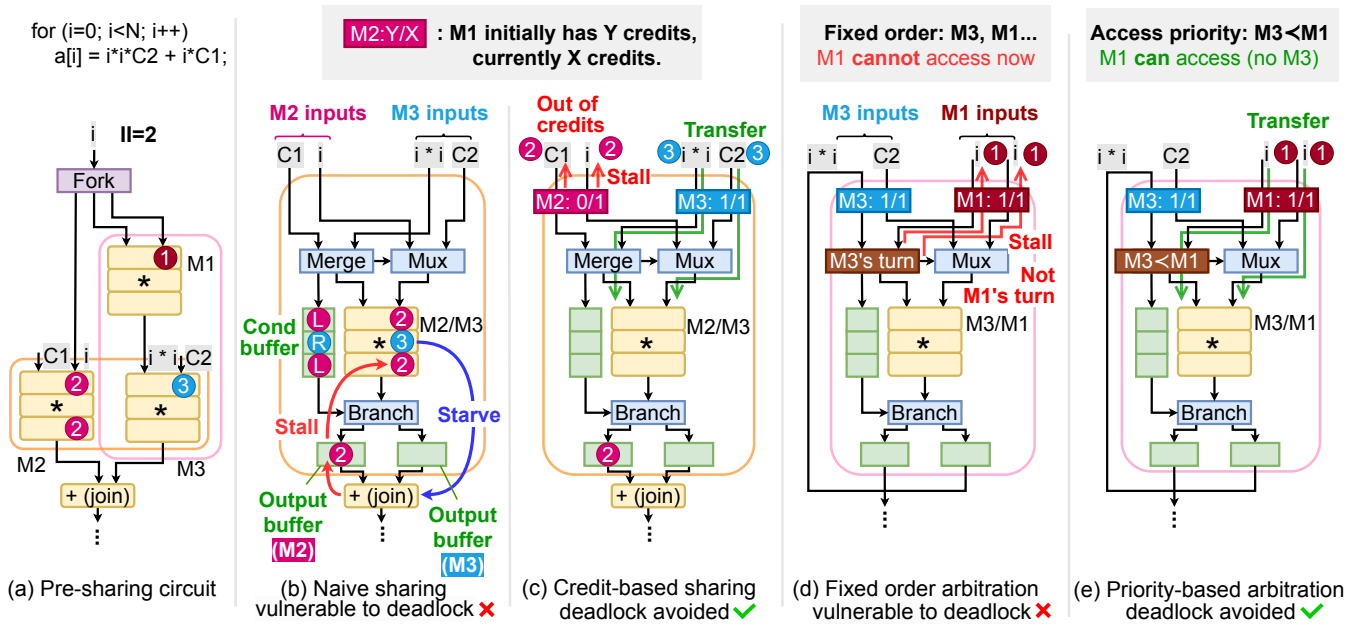

Figure 1. Naive resource sharing leads to deadlock [33]. Figure 1a describes the circuit before sharing. Figure 1b and Figure 1c describe why naively sharing cannot prevent a deadlock state caused by head-of-line blocking; we avoid it using credit-based access control (discussed in Section 4.1). Figure 1d and Figure 1e describe why a fixed order arbitration causes deadlock; we avoid it using priority-based arbitration (discussed in Section 4.2).

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 Figure 1a. The circuit implements part of the code above it. A token carrying the value of i becomes available at the circuit's input every second cycle (as II = 2); it is replicated by *Fork* and sent to $M1$ and $M2$ (both have latency = 3 and, thus, 3 pipeline stages that can compute on different tokens in parallel). $M3$ consumes $M1$'s results; a join (+) consumes tokens produced by $M2$ and $M3$. $M1$, $M2$, and $M3$ are under-utilized (i.e., not all pipeline stages contain tokens simultaneously); therefore, sharing is advantageous here, as we can potentially maintain the circuit performance while reducing the functional unit usage.

Assume that we implement operations $M2$ and $M3$ using a single unit. Figure 1b describes the shared unit (shared by $M2$ and $M3$) with a sharing wrapper: a merge and a mux select the inputs of the original operations and send the selected operands 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* [17, 33, 40]. Initially, the shared unit executes $M2$ twice before $M3$. The resulting state (in Figure 1b) has an execution dependency cycle: (1) 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). (2) The token at the head of the line blocks the token after it, i.e., the first token of $M3$. (3) 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. [33] propose a solution to this deadlock problem. Consider Figure 1b. There is no deadlock risk 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, which is 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 that $M1$ and $M3$ now share one unit with

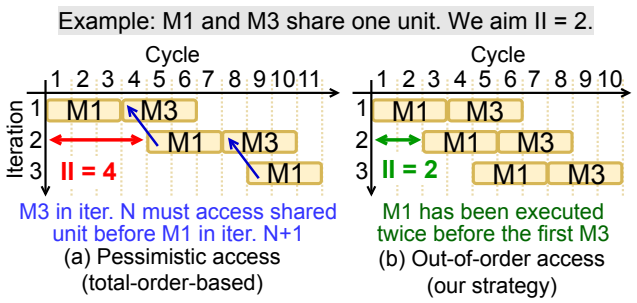

Figure 2. Schedules: $M1$ and $M3$ in Figure 1a share one unit. Figure 2a: A total token order might hurt II, thus limiting sharing. We can achieve the schedule in Figure 2b.

an order "$M1, M3, M1, M3, \ldots$". We illustrate the schedule in Figure 2a: in *cycle 1*, $M1$ starts; in *cycle 4*, $M3$ receives data from M1, and starts; in *cycle 5*, $M1$ starts; in *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 the performance limitation, the prior work relies on a time-consuming strategy to find a good ordering. Recent dataflow HLS strategies achieve high parallelism where BBs execute out of order for performance benefits—enforcing a total token order diminishes the advantage [21].

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 [9, 10, 17, 40]. (1) We seize opportunities like in Figure 2a by allowing out-of-order accesses to the shared unit, thus achieving the schedule like in Figure 2b. (2) Our deadlock avoidance strategy is agnostic to the circuit's control flow, thus, it is suitable for different dataflow HLS strategies [21, 29].

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 (i.e., dependency cycles do not exist in any possible circuit execution); this work focuses on avoiding the deadlock situations introduced by resource sharing. We classify the sharing-induced deadlocks into two types: (1) *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. (2) *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 [17, 40].

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 unit. 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 that, 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 left yet, $M2$ 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 unit 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, since the credit count limits the number of computations in the shared unit. 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 (1) the first request of $M3$ will not be issued before the first $M1$ is available, and (2) 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 unit. We realize this 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 priorities maintain the II. Section 5.3 discusses how to determine one that does.

4.3 A Sharing Mechanism for Dataflow Circuits

This subsection leverages the insights from Section 4.1 and Section 4.2 to devise a sharing circuit construction strategy. We denote the set of operations that share a unit as a *sharing group* $G := \{op_1, op_2, \ldots, op_{|G|}\}$; the group size is $|G|$. A group $G := \{op_i\}$ of one operation denotes that op_i does not share a unit with any other operations.

Consider Figure 3: a unit is shared between $|G| = 3$ operations using our credit-based method. Similar to the circuit in Figure 1, the sharing circuit consists of a mux, an arbiter merge (implementing the priority-based arbitration in Section 4.2), a shared unit, a condition buffer (*cond buffer*), and a branch (their usages are discussed in Section 3). We extend it as follows: for each operation op_i, a *credit counter* CC_i tracks 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). To avoid deadlock, the following must hold (described in Section 4.1):

$$N_{CC,i} \leq N_{OB,i}, \forall i. \tag{1}$$

In Figure 3, CC_1 has two initial credits and OB_1 has two slots, which honors the constraint described in Equation 1.

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.

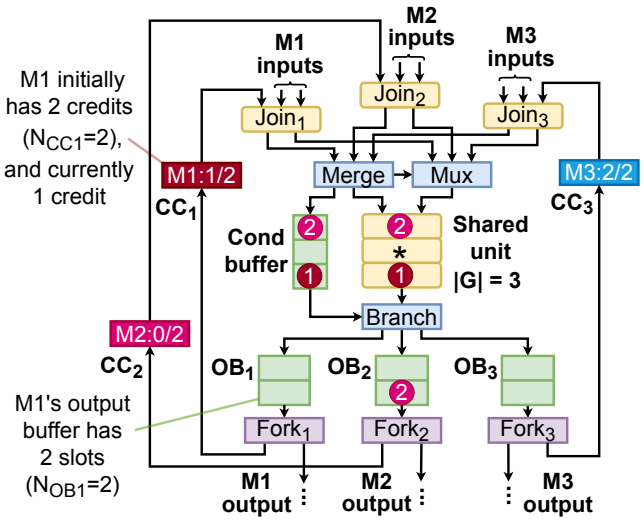

Figure 3. Credit-based sharing wrapper for a group of 3 operations, $M1$, $M2$, and $M3$.

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 [14], i.e., it propagates tokens to the successors only if both are ready, which prevents a credit from being returned before the OB slot is freed. To avoid combinational loops, in the same cycle, the credit being returned cannot be used for starting a new computation.

Our sharing wrapper introduces area and timing overheads. To estimate when sharing is beneficial (i.e., before mapping the circuit onto an FPGA, which can be very time-consuming), we devise a cost function for the shared units of operation type T and wrapper logic as the following:

$$\underbrace{C_T \cdot |groups|}_{\text{Cost of the shared units (reduces with sharing)}} + \underbrace{\sum_{G_i \in groups} C_{WP}(|G_i|)}_{\text{Cost of the sharing wrappers (increases with sharing)}}, \tag{2}$$

where C_T is one shared unit's resource cost (e.g., DSP blocks on FPGAs); $groups := \{G_1, G_2, \ldots\}$ is the set of non-empty sharing groups (in total $|groups|$ groups) of operations with type T; $C_{WP}(\cdot)$ is the resource cost of the sharing wrapper given the group size $|G_i|$. The first term models the cost of the shared units, which decreases with more units shared (i.e., fewer sharing groups) as fewer units need to be implemented. The second term models the cost of the sharing wrappers, which increases with more units shared since the selection and arbitration logic become more complex. Equation 2 can be used to model different resources and characterize different platforms (e.g. FPGAs and ASICs). It can guide the sharing heuristic (Algorithm 1 in Section 5.2) to find more desirable solutions. By plotting Equation 2 against the size of the sharing group, we can decide if sharing a particular

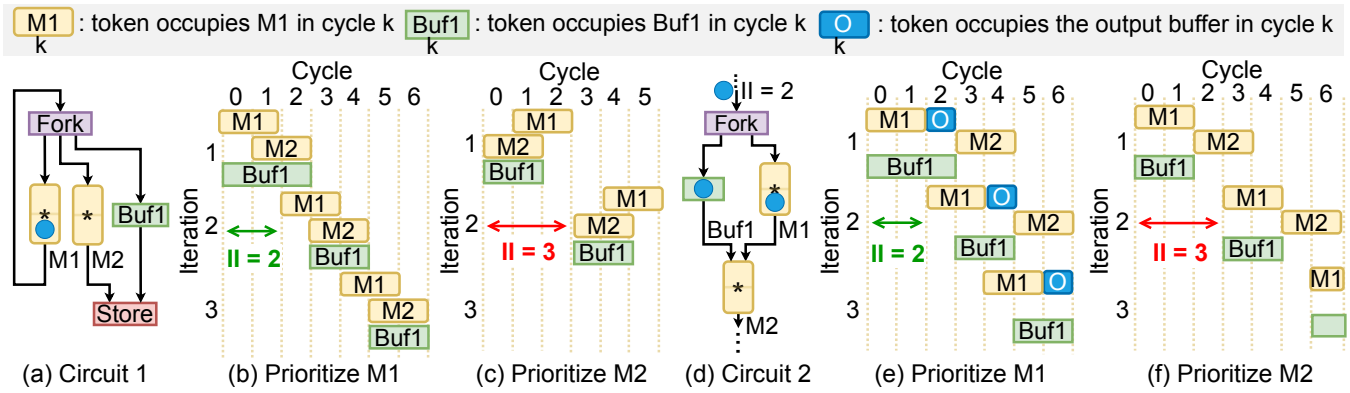

Figure 4. Access priority may penalize II when disregarding the dependency between operations. Figure 4a and Figure 4d: in both circuits, M1 and M2 share one unit and M2 depends on M1. Figure 4b and Figure 4e: respective schedules when $M1 \prec M2$; II is maintained in both cases. Figure 4c and Figure 4f: schedules when $M2 \prec M1$; II is penalized in both cases.

unit is beneficial at all (e.g., sharing integer adders is not beneficial—the cost of the wrapper quickly surpasses the cost of the removed adders). Section 6 will show that the overheads are insignificant—for practical sharing applications, say, sharing expensive floating-point units, the saving compensates the sharing overhead.

5 Sharing and Performance

This section devises a strategy that determines sharing groups; an access priority is then decided for each group. We aim 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 describe cases where sharing increases II and present heuristics that generate performance-preserving grouping and priority schemes.

5.1 Examples: Sharing Might Penalize II

This subsection discusses examples of when sharing hurts II: We discuss the cases where a good priority must be selected and the cases where none of the access priorities preserves the II, i.e., operations should not share one unit.

Example 1: access priorities. Figure 4a describes a CFC with operations $M1$ and $M2$ (they have the same type, and have 2 pipeline stages). The II before sharing is 2, which indicates that both $M1$ and $M2$ need the same pipeline stage once every two clock cycles. We can potentially implement $M1$ and $M2$ using one unit while maintaining II = 2: in each iteration, we activate $M1$ and $M2$ in two separate cycles. In this way, they can still be activated every 2 cycles after sharing and the II is maintained. Since *Fork* always makes data tokens available to $M1$ and $M2$ simultaneously, arbitration between them 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., $M1$ will access the shared unit first when both $M1$ and $M2$ can execute); 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.

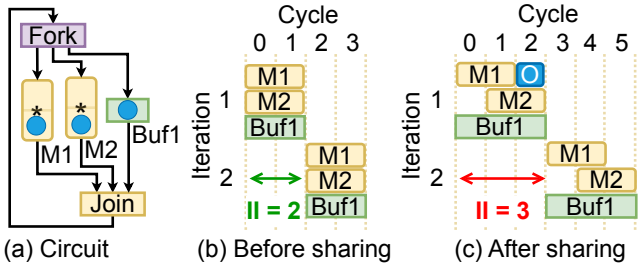

Figure 5. No access priority maintains the II. M1 and M2 are in the same SCC and are always executed simultaneously before sharing. Thus, M1 and M2 should not share one unit.

Example 2: access priorities. Figure 4d depicts a subcircuit receiving a new token with an II = 2; it might be desirable to have $M1$ and $M2$ share one unit. $M1 \prec M2$ 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 $M2$, i.e., the circuit 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 operations. To avoid a performance penalty, the priority should follow the data dependency: when op_1 and op_2 share one unit, if op_2 needs the result from op_1, then op_1 is prioritized. Beyond dataflow circuits, many resource-constrained scheduling techniques employ this policy [15, 51].

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

Example 3: sharing groups. Finding a performance-preserving grouping scheme seems straightforward—we want underutilized operations to share one unit [33]. Yet, there is still one caveat. Consider Figure 5a: $M1$ and $M2$ (both have latency = 2) share one unit. Before sharing, the circuit can run in II = 2 (Figure 5b). 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 in which every node is reachable by every other node [43].

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

This subsection presents our sharing group heuristic.

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 $SCC_i \rightarrow SCC_j$ indicates an edge $n_i \rightarrow n_j \in CFC$ exists, such that $n_i \in SCC_i, n_j \in SCC_j$; this edge indicates that operations in SCC_j need the data from operations in SCC_i.

The 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 candidates* (operations that can share a unit with other operations), 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 bottleneck, having $M1$ and $M2$ share one unit 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 subsection devises a heuristic that assigns an access priority scheme to each sharing group (determined using the sharing group heuristic).

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 $Buf1$; a possible topological order is $SCC_0 \prec SCC_1 \prec SCC_2 \prec SCC_3$. When $M1$ and $M2$ share a unit, 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, 51], 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 [43]) and maximum distances in the SCCs (the SCCs are usually sparsely connected, i.e., the number of paths is small). In Section 6.3, we show that our strategy can produce efficient circuits with scalable runtime.

5.4 Buffer Sizing and Credit Allocation

This subsection describes how to decide on buffer and credit sizing to maintain circuit II.

On reconvergent paths—paths that 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, 34, 41]. If the latency change is confined to a certain range, it is unnecessary to adjust the buffer placement to counteract variation in the computation pattern. When awaiting arbitration, the operation postpones consuming tokens available at the input. In the circuit's steady state, if a new token becomes available at the input every II cycles, postponing its acceptance for less than II cycles will not propagate the stalls to the rest of the system as no token will be accumulated. Since there is no additional token accumulation, the reconvergent paths do not require any extra buffering to balance.

No additional buffer requirement. In our sharing strategy, for any operation op in a sharing group G, the maximum time that op is postponed is $t_{max} = |G| - 1$ (it has to wait for all other operations before it can get access); since $II \geq |G|$, this means $t_{max} \leq II - 1$. Thus, applying our sharing strategy does not require additional buffers.

Algorithm 1: Sharing groups

Data: Dataflow circuits, performance-critical CFCs, occupancies of operations, sharing candidates.
Result: Sharing groups *groups*.

▷ For K sharing candidates, initialize *groups* with K groups, each has 1 operation.
groups ← {{*op*}|∀*op* ∈ *sharing candidates*}
▷ Greedily merge groups until no changes can be made.
while *groups modified* **do**
  for $G_i, G_j \in groups$ **do**
    if $\neg check\_R1(G_i \cup G_j)$ **then**
      continue ▷ R1: ops must have the same type.
    if $\exists CFC \in Critical\ CFCs : \neg check\_R2(G_i \cup G_j, CFC)$ **then**
      continue ▷ R2: sum of occupancy ≤ capacity.
    if $\exists CFC \in Critical\ CFCs : \neg check\_R3(G_i \cup G_j, CFC)$ **then**
      continue ▷ R3: ops in the same *CFC* → ops always start at different time.
    if *Merging* G_i *and* G_j *reduces cost* **then**
      $G_i \leftarrow G_i \cup G_j, G_j \leftarrow \{\}$ ▷ Merge G_i and G_j.

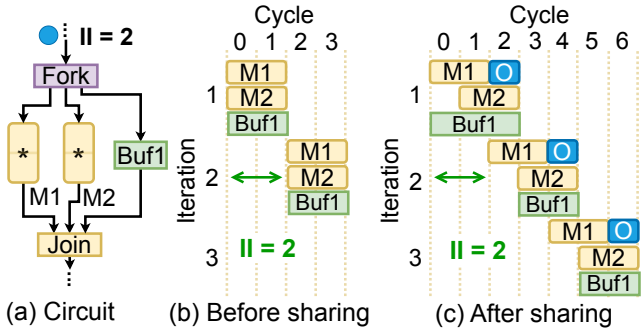

Figure 6. On reconvergent paths, operations sharing the same unit do not increase buffer usage. Before and after sharing, *Buf1* only needs 1 slot despite the latencies on other reconvergent paths having increased.

Consider Figure 6a: $M1$ and $M2$ (latency = 2) share one unit. 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 Figure 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 of the circuit after sharing (starting from iteration 2), a token stays in *Buf1* for 2 cycles, thus, no need to resize it.

Algorithm 2: Group access priority

Data: Dataflow circuit, performance-critical CFCs, sharing group G.
Result: Group G's access priority $G.prio$.

$G.prio = [op_1, op_2, \ldots, op_{|G|}]$ ▷ Priority as a list
while *G.prio modified* **do**
  for $i \in 2 \ldots |G|$ **do**
    for $CFC \in CriticalCFC$ **do**
      $SCCG_{CFC} = getSCCGraphOfCFC(CFC)$
      ▷ If ops are in different SCCs of the same CFC, we decide on the priority based on the topological order of the SCC graph ($SCCG_{CFC}$).
      if $G.prio[i-1], G.prio[i] \in CFC$ **then**
        if $\exists SCC_i, SCC_j \in SCCG_{CFC} : G.prio[i-1] \in SCC_i, G.prio[i] \in SCC_j$ **then**
          $getTopologicalOrder(SCCG_{CFC})$
          if $SCC_i.order > SCC_j.order$ **then**
            $G.prio.swap(i-1, i)$

Credit sizing requirements. There are many credit and output buffer sizing possibilities to honor the correctness constraint (see Equation 1 in Section 4.3)—yet, not all of them achieve our desired performance with reasonable cost. For operation *op*, the initial number of credits $N_{CC,op}$ is the upper bound (in all possible circuit execution) of the sum of the number of tokens carrying the intermediate results of *op* (i.e., inside the shared unit, indicating the number of simultaneous computations) and staying in the output buffers of *op* (i.e., due to temporary unavailability of the successor caused by sharing). Credits must be sufficiently allocated to avoid an II penalty, yet naively assigning many credits incurs a high output buffer cost [34].

Credit allocation rule. To maintain the circuit's II, for each operation *op* in any sharing group with occupancy Φ_{op}, we assign the number of initial credit $N_{CC,op}$ as

$$N_{CC,op} = \Phi_{op} + 1, \tag{3}$$

where Φ_{op} credits keep the shared unit fully utilized; the one extra credit hides the latency of returning the credit (see Section 4.3) and accounts for the token that occupies *op*'s output buffer.

Why do we need more credits than the average occupancy (i.e., $N_{CC,op} > \Phi_{op}$)? 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 immediately 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 (in cycle 2) since there would be no available credits.

Why is exactly 1 more credit sufficient (Equation 3)? In a steady state, there is *at most* one token staying in the output

Technique	LUTs	FFs	DSPs
No sharing	76k/101k (75%)	115k/202k (57%)	**790/600 (132%)**
CRUSH	46k/101k (45%)	45k/202k (22%)	**60/600 (10%)**

Table 1. By unrolling (a common HLS optimization strategy to increase parallelism [46]) the loops in *gesummv* by 75, the resource cost easily exceeds the capacity of our target Kintex-7 FPGA device. After using our method, the kernel can easily fit on the FPGA.

buffer, as a token remains in the buffers for at most $|G|$ - 1 = II - 1 cycles due to arbitration. Thus, in the worst case, *CC* gets a new credit from the output buffer every II cycles, allowing the shared unit to initiate a new computation every II cycles for *op*. Therefore, Φ_{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 CRUSH—our resource sharing strategy for dataflow circuits. The artifact evaluation instructions are available in Appendix A.

The costs of an FPGA design are the number of used LUTs (implement combinational logic) and FFs (registers). An FPGA synthesis tool further offloads complex arithmetic functions onto specialized DSP blocks. DSP blocks are scarce resources: recent FPGAs typically have 300k–800k LUTs/FFs and, in contrast, only 1k–2k DSP blocks [27].

To demonstrate the practical importance of sharing, we aim to share the floating-point arithmetic units as much as possible. These units are often a desirable sharing candidates for FPGA designs since they demand multiple DSP blocks. Our strategy is also applicable to other units.

6.1 Methodology

This subsection describes the experiment setup.

We evaluate our approach on standard HLS benchmarks that exhibit different computation patterns and loop properties, including a subset of PolyBench [38] (*atax*, *bicg*, *2mm*, *3mm*, *symm*, *gemm*, *gesummv*, *mvt*, and *syr2k*) and *gsum* and *gsumif*, which have irregular computation patterns that are often used to showcase the benefit of dynamic scheduling [11]. All the kernels have an II > 1 due to the long-latency loop-carried dependencies between floating-point operations, therefore, many units are underutilized and can be shared without a performance penalty.

We employ *Dynamatic* [24, 31], an open-source HLS compiler for translating C code to a dataflow circuit. Dynamatic places buffers to optimize the circuit throughput and frequency [34]. After obtaining dataflow circuits, we apply the heuristics described in Section 5.2 and Section 5.3 to determine the sharing groups and the access priority within each group. We then apply the sharing strategy from Section 4.3 to share floating-point arithmetic units in the circuit. We determine the number of credits and output buffer sizes according to the occupancy in the performance analysis result reported by Dynamatic (see Section 5.4).

We use ModelSim [42] to obtain the execution latency in the clock cycle count and verify the functionality. We confirm that the circuit produces the same result as the C code and the circuit does not deadlock. We use Vivado (2019.1) [45] to synthesize the RTL design and obtain the post-place-and-route area and maximum frequency, targeting a Kintex-7 FPGA (part number: xc7k160tfbg484-1) with a clock period (CP) of 6 ns. We use the same MILP formulation as Dynamatic for the performance optimization, with the same solver (Gurobi [25] version 11.0.3) and a timeout of 2 min.

We show that our sharing strategy is equally effective on two different dataflow HLS strategies: (1) a circuit generation strategy that organizes units into BBs [29, 31] and (2) a circuit generation strategy that organizes units into producer-consumer pairs [21].

We contrast our method with a prior resource sharing strategy, which restricts access to the shared unit to BB ordering [33] (see Section 3). For fairness, both sharing strategies aim to share the functional units as much as possible without hurting the performance of the *inner-most loops*. We show that, while both approaches maintain the circuit performance, CRUSH can seize more sharing opportunities (e.g., Figure 2) while spending substantially lesser runtime.

6.2 Discussion: Importance of Resource Sharing on FPGAs

To demonstrate the feasibility of our approach on larger benchmarks and the importance of resource sharing on FPGAs, we unroll the inner loop of the kernel *gesummv* by 75 (typically done in HLS and on FPGAs for parallelism [46]). Table 1 reports the synthesis result: without sharing, even a single kernel does not fit on our target Kintex-7 FPGA (i.e., requires 32% more than the available DSP units), whereas it fits after applying CRUSH to share resources. This demonstrates the relevance of sharing: even a modest workload requires it to fit onto an FPGA. Moreover, resource sharing frees up available resources, which can be used to perform other computations.

6.3 Discussion: Effectiveness of Our Strategy

Table 2 reports the utilization, performance, and optimization runtime of different approaches. The column **Technique** categorizes the used sharing strategy: **Naive** indicates no resource sharing [34], **In-order** indicates total-order-based sharing [33], and **CRUSH** indicates our credit-based sharing strategy. The columns **Functional units** and **DSPs** respectively report the functional unit count in the dataflow circuit and the number of DSPs in use—they indicate the effectiveness of the sharing strategies.

CRUSH vs Naive. CRUSH greatly reduces the functional unit usage—all units with identical types can be implemented

Benchmark	Technique	Functional units	DSPs	Slices	LUTs	FFs	CP (ns)	Cycles	Exec. time (us)	Opt. time (s)
atax	Naive	2 fadd 2 fmul	10	616	1581	1921	5.2	4604	23.9	0.7
	In-order	1 fadd 1 fmul	5	577	1540	1600	5.3	4604	24.4	8.5
	CRUSH	1 fadd 1 fmul	5	580	1669	1446	5.5	4604	25.3	1.2
bicg	Naive	2 fadd 2 fmul	10	592	1434	1801	5.1	8842	45.1	0.5
	In-order	1 fadd 1 fmul	5	551	1475	1531	5.2	8786	45.7	6.2
	CRUSH	1 fadd 1 fmul	5	524	1437	1310	5.3	8758	46.4	0.7
gsum	Naive	5 fadd 4 fmul	22	757	2135	2833	6	3642	21.9	0.5
	In-order	5 fadd 4 fmul	22	757	2135	2833	6	3642	21.9	33.6
	CRUSH	1 fadd 1 fmul	5	531	1521	1386	5.8	3642	21.1	1
gsumif	Naive	7 fadd 4 fmul	26	1065	2730	3694	6.3	3556	22.4	1
	In-order	3 fadd 2 fmul	12	829	2361	2568	6.3	3624	22.8	61.4
	CRUSH	1 fadd 1 fmul	5	616	1794	1731	6.6	3556	23.5	1.2
2mm	Naive	2 fadd 4 fmul	16	1364	3198	3798	5.7	18627	106.2	1.6
	In-order	1 fadd 1 fmul	5	1228	3110	3211	6	18242	109.5	27.7
	CRUSH	1 fadd 1 fmul	5	1168	3144	3004	5.5	18223	100.2	2
3mm	Naive	3 fadd 3 fmul	15	1224	2933	3246	5.2	28195	146.6	2
	In-order	1 fadd 1 fmul	5	1003	2718	2436	5.6	27330	153	11.7
	CRUSH	1 fadd 1 fmul	5	1046	2986	2232	5.5	27310	150.2	3.1
symm	Naive	4 fadd 7 fmul	29	1816	4337	4925	6.3	39922	251.5	1.1
	In-order	1 fadd 1 fmul	5	1464	3865	3376	6.5	47113	306.2	60.3
	CRUSH	1 fadd 1 fmul	5	1432	3885	3056	6.3	39372	248	1.7
gemm	Naive	1 fadd 3 fmul	11	760	1939	2363	5.5	76851	422.7	0.8
	In-order	1 fadd 1 fmul	5	729	1975	2162	5.6	76433	428	12.7
	CRUSH	1 fadd 1 fmul	5	701	1955	2076	5.6	76433	428	1.1
gesummv	Naive	3 fadd 4 fmul	18	885	2246	2988	5.5	8856	48.7	0.7
	In-order	1 fadd 1 fmul	5	746	2048	2099	5.7	9267	52.8	16.4
	CRUSH	1 fadd 1 fmul	5	720	2052	1892	5.5	8773	48.3	1
mvt	Naive	2 fadd 2 fmul	10	649	1637	1989	5.1	17646	90	0.8
	In-order	1 fadd 1 fmul	5	625	1625	1680	5.6	17487	97.9	4.4
	CRUSH	1 fadd 1 fmul	5	610	1699	1498	5.2	17477	90.9	1.1
syr2k	Naive	2 fadd 5 fmul	19	1307	2970	3610	5.5	17472	96.1	2
	In-order	1 fadd 1 fmul	5	1133	3014	2983	5.6	18262	102.3	37.8
	CRUSH	1 fadd 1 fmul	5	1087	2992	2628	5.6	17372	97.3	2.7
Average improvement of CRUSH vs Naive. Slices: -17% LUTs: -6% FFs: -32% DSPs: -66% Opt. time (s): +47% Exec. time (us): +1%										
Average improvement of CRUSH vs In-order. Slices: -7% LUTs: -3% FFs: -15% DSPs: -12% Opt. time (s): -90% Exec. time (us): -4%										

Table 2. Comparison between no sharing (Naive) [34], total-order-based sharing (In-order) [33], 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.

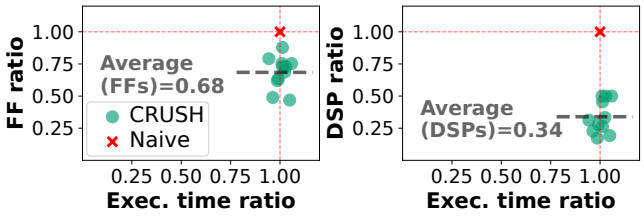

Figure 7. The trade-off between FF, DSP, and latency (Exec. time) of CRUSH (this work) vs Naive [29] in Table 2. All metrics are normalized to the value of Naive. The dashed lines indicate the average values.

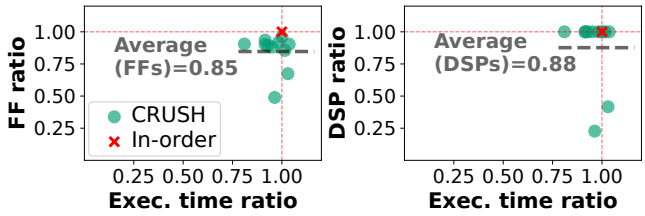

Figure 8. The trade-off between FF, DSP, and latency (Exec. time) of CRUSH (this work) vs In-order [33]. All metrics are normalized to the value of In-order.

using one shared unit. This is achieved at a negligible performance degradation, as reported in **Exec. time (us)** (calculated as CP × Cycles). Compared with **Naive**, **CRUSH** has 1% performance loss on average (CRUSH has a CP overhead but has a lower cycle count, as discussed later). Figure 7 depicts the trade-off between FF, DSP, and latency from Table 2: **CRUSH**'s solutions are Pareto optimal or dominate the solutions of **Naive**. This shows that our heuristics successfully maintain the performance (Section 5.2 and Section 5.3).

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. The overhead is negligible as it is typically within 1 s.

Compared with **Naive**, **CRUSH** occasionally has a lower clock cycle count. This effect is accidental—**Naive** and **CRUSH** are expected to have approximately the same clock cycle count. They are different because Dynamatic's units

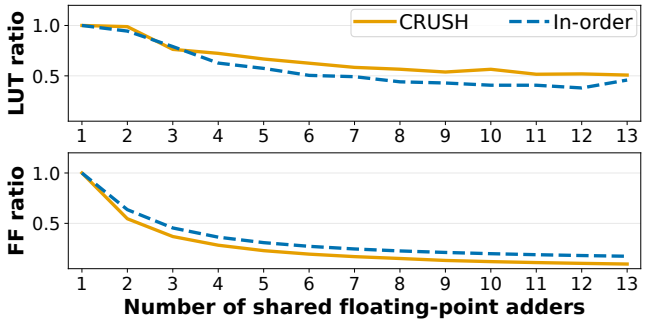

Figure 9. Resource cost (Equation 2) ratio between sharing and not sharing floating-point adders; we report the shared unit's and the sharing wrappers' utilizations (see Figure 3) when synthesized in isolation. The solid curves denote **CRUSH**'s ratio; the dashed curves denote **In-order**'s ratio.

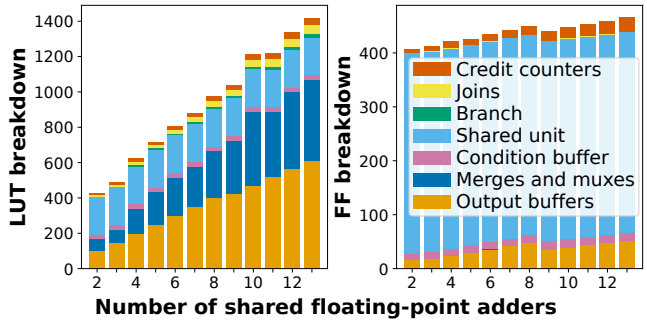

Figure 10. Resource breakdown of **CRUSH**'s sharing wrapper (i.e., the resource cost of each dataflow unit in Figure 3) for different group sizes $|G|$. The number of initial credits and buffer sizes is calculated using Equation 3, where unit occupancy Φ_{op} is the maximum achievable occupancy when $|G|$ operations share the unit ($lat_{op}/|G|$).

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; **CRUSH** eliminates head-of-line blocking, which accidentally improves the performance.

CRUSH vs In-order. 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 accesses to shared units (see Section 3). Figure 8 reports the trade-off between FF, DSP, and latency in Table 2: except for *gsumif* (where **CRUSH** has higher latency but lower DSP usage), *atax* and *bicg* (where **CRUSH** is Pareto optimal), **CRUSH** Pareto-dominates **In-order** for all these metrics. **CRUSH** has significantly better runtime (an average reduction of 90%) than **In-order**, since **In-order** requires repetitively solving the MILP formulation to evaluate the effect of sharing.

6.4 Discussion: Resource Efficiency of Our Strategy

Columns **Slices**, **LUTs**, **FFs** in Table 2 indicate the FPGA resource utilization (a slice contains several LUTs and FFs). Outside the DSP units, the floating-point units use pipeline registers to buffer the intermediate results and logic to control the DSP units. Thus, compared with **Naive**, **CRUSH** reduces LUT usage when many floating-point units are shared (*gsum* and *gsumif*) and reduces the FF usage consistently.

CRUSH's sharing wrapper has a similar cost as **In-order**'s wrapper. We individually synthesized each building block in the sharing wrappers (i.e., each unit in Figure 3). Figure 9 reports the aggregated resource usage of a shared floating-point adder with a sharing wrapper: the orange solid lines denote **CRUSH**; the blue dashed lines denote **In-order**. The figure shows only a minor difference between the resource usage of CRUSH and In-order, which can also be confirmed in Table 2: when CRUSH and In-order share the same number of units, CRUSH uses more LUTs while In-order uses more FFs. Compared with **In-order**, **CRUSH** achieves an average reduction of 7% Slices, 3% LUTs, and 15% FFs because more functional units have been shared.

Figure 10 breaks down the resource usage of each dataflow unit in a sharing circuit (e.g., the one in Figure 3). The sharing circuit is not FF-demanding—it consumes far fewer FFs than the shared floating-point adder. As the size of the sharing group grows, the sharing circuit uses more LUTs. This trend is expected: as more operations share the same unit, the selection and distribute logic (i.e., the merge, muxes, and the branch in Figure 3) becomes more complex, and more output buffers have to be placed. The output buffers constitute the majority of the sharing overhead (approximately 50% LUTs when sharing 7 operations); this is due to the bypass and FIFO logic. Recall that these buffers are used to prevent the shared unit from being stalled. However, if we can prove (e.g., using model checking [50]) that the output is always ready to take tokens computed by the shared unit, then the output buffer is redundant and can be removed to save resources.

The column **CP (ns)** in Table 2 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 orthogonal to our goal, i.e., reducing resource usage; it can be mitigated by enhancing Dynamatic's timing model [41] to support sharing.

6.5 Discussion: Generality of Our Strategy

In the previous sections, our 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 [21]; prior work (**In-order**) re-introduces the BB order, which

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	789	2114	2374	5.5	3594	19.8	4.9
	CRUSH	1 fadd 1 fmul	5	760	2202	1893	5.6	3518	19.7	6.7
bicg	**Fast token**	2 fadd 2 fmul	10	675	1794	2048	5.9	8090	47.7	0.8
	CRUSH	1 fadd 1 fmul	5	626	1808	1557	5.7	8120	46.3	1.1
gsum	**Fast token**	5 fadd 4 fmul	22	878	2213	2886	6	3633	21.8	0.4
	CRUSH	1 fadd 1 fmul	5	527	1559	1389	5.8	3633	21.1	0.7
gsumif	**Fast token**	7 fadd 4 fmul	26	1026	2797	3467	6.4	2317	14.8	0.4
	CRUSH	1 fadd 1 fmul	5	656	1853	1453	6.9	2317	16	0.6
2mm	**Fast token**	2 fadd 4 fmul	16	1678	3785	4175	5.6	8623	48.3	16.8
	CRUSH	1 fadd 1 fmul	5	1439	3820	3381	5.6	8234	46.1	17.1
3mm	**Fast token**	3 fadd 3 fmul	15	2004	4712	4847	5.9	8624	50.9	11.2
	CRUSH	1 fadd 1 fmul	5	1822	4619	3833	6	8232	49.4	11.9
symm	**Fast token**	4 fadd 7 fmul	29	1886	5042	5064	5.8	35580	206.4	30.4
	CRUSH	1 fadd 1 fmul	5	1534	4578	3196	5.8	34580	200.6	27.6
gemm	**Fast token**	1 fadd 3 fmul	11	881	2461	2581	5.6	69234	387.7	24.4
	CRUSH	1 fadd 1 fmul	5	911	2479	2294	5.8	68833	399.2	25.8
gesummv	**Fast token**	3 fadd 4 fmul	18	873	2433	2929	5.4	8017	43.3	1.1
	CRUSH	1 fadd 1 fmul	5	745	2180	1833	5.8	7931	46	1.4
mvt	**Fast token**	2 fadd 2 fmul	10	845	2342	2587	5.7	8004	45.6	1.1
	CRUSH	1 fadd 1 fmul	5	852	2435	2096	5.6	7890	44.2	1.2
syr2k	**Fast token**	2 fadd 5 fmul	19	1590	3989	4072	5.7	16445	93.7	69.2
	CRUSH	1 fadd 1 fmul	5	1404	3981	3088	5.7	16345	93.2	64.8
Average improvement of CRUSH vs Fast token. Slices: -14% LUTs: -7% FFs: -29% DSPs: -66% Opt. time (s): +21% Exec. time (us): -0%										

Table 3. Comparison between Fast token (a more recent dataflow HLS strategy [21]) 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 [33] does not apply here; this shows that our strategy is general. The results show that CRUSH is effective on different dataflow HLS approaches.

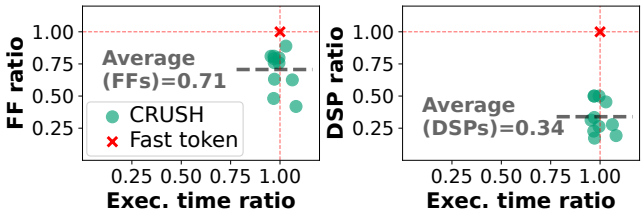

Figure 11. The trade-off between FF, DSP, and latency when applying CRUSH to a more recent dataflow HLS strategy [21] (results in Table 3). All metrics are normalized to the value of the pre-sharing circuit (**Fast token**).

requires non-trivial code analysis [23], may diminish the benefit of omitting BB organization, and does not make up for the limited sharing capability. In contrast, **CRUSH** can be effortlessly ported to different dataflow HLS approaches; we demonstrate this by integrating unmodified **CRUSH** into a recent fast token delivery approach [21].

Table 3 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 that 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 an average reduction of 66% DSPs, 29% FFs, and 14% slices; this is similar to the improvement in the other HLS strategy (Table 2) using **CRUSH**. The trade-off between resources and performance is depicted in Figure 11, our solution is systematically Pareto-optimal or Pareto dominates **Fast-token**.

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. All these benefits make HLS of dataflow circuits more attractive and practical. CRUSH has been integrated into the Dynamatic HLS compiler (https://github.com/EPFL-LAP/dynamatic).

Acknowledgments

We thank the anonymous reviewers and our shepherd Aaron Lee Smith for their valuable feedback. This work has been supported by the Swiss National Science Foundation (grant number 215747) and the ETH Future Computing Laboratory (donation from Huawei Technologies).

A Artifact Appendix

A.1 Abstract

This artifact contains all the source codes and benchmarks of CRUSH. It utilizes a Dockerfile to set up the environment and scripts to replicate our experiments in Section 6 and generate data for Tables 1–3 and Figures 7–11.

A.2 Artifact check-list (meta-information)

- **Program:** The source code for CRUSH is available in `pyhash` (in the Zenodo archive).
- **Compilation:** Use the Bash script `build_project.sh` to build the project.
- **Run-time environment:** The experiments run in an Ubuntu 22.04 docker (Dockerfile provided). The installation tarballs and executables of Vivado 2019.1, ModelSim 20.1, and the Gurobi optimizer have to be downloaded by the evaluator (see Appendix A.3.3).
- **Hardware:** AMD Ryzen 7 PRO 5850U (or any similar processors) with ≥ 32 GB memory.
- **Output:** Generate simulation reports, synthesis reports, and log files (for optimization runtime). Tables 2–3 and Figures 7–11 are automatically generated.
- **Experiments:** Run all experiments using the Bash script `run_all_experiments.sh`. `README.md` documents the individual experiments.
- **Disk space required:** The public archive needs 170 MB after unzipping. The docker image and the software installation packages add up to ≥ 180 GB.
- **Time needed to prepare workflow:** Approximately 2.5 hours to setup the workflow.
- **Time needed to complete experiments:** Around 4 hours to complete all experiments.
- **Publicly available:** Yes.
- **Code licenses:** MIT license.
- **Workflow automation framework used:** Docker.
- **Archived:** Archived on Zenodo with DOI: 10.5281/zenodo.14017399.

A.3 Description

The detailed descriptions are all documented in `README.md` in the Zenodo archive [47]. In this subsection, we describe some notable hardware/software requirements (e.g., proprietary software dependencies).

A.3.1 How to access.

The artifact is publicly available at https://doi.org/10.5281/zenodo.14017399. In this Zenodo archive, `asplos25summer-crush-main.zip` contains all the source codes, benchmarks, scripts, and environment setup files (i.e., Dockerfile). `generated-files.zip` contains all the generated files, i.e., log files, synthesis/simulation reports, figures, and tables.

A.3.2 Hardware dependencies.

A Ubuntu 22.04-LTS Linux machine with at least 180 GB of free disk space and 32 GB memory.

A.3.3 Software dependencies.

The experiments depend on proprietary software. Vivado (version 2019.1) [45] and ModelSim (version 20.1) [42] have free versions. Gurobi optimizer (version 11.0.3) [25] can be downloaded for free and offers a free academic license. `README.md` provides the instructions for downloading the software and obtaining the Gurobi license.

All the remaining dependencies are automatically configured by `Dockerfile`.

A.4 Installation

This subsection describes the steps for generating the run-time Docker environment for our experiments.

- Install Docker (https://www.docker.com/).
- Download the Zenodo archive (https://doi.org/10.5281/zenodo.14017399) and follow the instructions in the `README.md` to download the proprietary software dependencies (see Appendix A.3.3), obtain the Gurobi license, and build the docker image.

A.5 Experiment workflow

Follow the instructions in `README.md`. Launch the container, build Dynamatic using `build_project.sh`, and run all experiments using `run_all_experiments.sh`.

A.6 Evaluation and expected results

Tables 1–3 and Figures 7–11 will be generated with identical or nearly identical numbers as reported in Tables 1–3 (location of the files are documented in `README.md`). Any discrepancy in the synthesis results (i.e., Slices, LUTs, FFs) is due to the non-deterministic behavior of MILP solver on different machines. The optimization runtime is not expected to be identical but the reduction ratios are expected to be similar.

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. https://doi.org/10.1109/ICCAD.2004.1382681

[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. https://doi.org/10.1109/ASYNC.2006.26

[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. https://doi.org/10.1109/ISPASS.2005.1430572

[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. https://doi.org/10.1109/ICCAD.2007.4397291

[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, Germany, 1–8. https://doi.org/10.1109/FPL.2014.6927490

[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. https://doi.org/10.1145/2514740

[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. https://doi.org/10.1109/43.945302

[8] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th International Symposium on Microarchitecture*. Taipei, Taiwan, 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[9] 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. https://doi.org/10.1007/s10703-011-0134-0

[10] 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. https://doi.org/10.1109/MDT.2012.2205998

[11] 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. https://doi.org/10.1145/3373087.3375297

[12] 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. https://doi.org/10.1145/1146909.1147025

[13] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. 2010. Elastic Systems. In *Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign*. Grenoble, France, 149–58. https://doi.org/10.1109/MEMCOD.2010.5558639

[14] 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. https://doi.org/10.1145/1146909.1147077

[15] Neal C. Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W. Keckler. 2024. WASP: Exploiting GPU Pipeline Parallelism with Hardware-Accelerated Automatic Warp Specialization. In *Proceedings of 2024 IEEE International Symposium on High-Performance Computer Architecture* (Edinburgh, United Kingdom). 1–16. https://doi.org/10.1109/HPCA57654.2024.00086

[16] 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. https://doi.org/10.1145/3174243.3174268

[17] Willian James Dally and Brian Patrick Towles. 2004. *Principles and Practices of Interconnection Networks*. Elsevier, Amsterdam, Netherlands.

[18] Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York.

[19] Siemens EDA. 2024. Catapult High-Level Synthesis and Verification. Retrieved from https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[20] 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. https://doi.org/10.1145/3127041.3127055

[21] 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. https://doi.org/10.1109/FPL57034.2022.00046

[22] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. 2024. Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits. In *Proceedings of the 32nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 44–54. https://doi.org/10.1145/3626202.3637556

[23] Ayatallah Elakhras, Riya Sawhney, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2023. Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits. In *Proceedings of the 31st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 39–45. https://doi.org/10.1145/3543622.3573050

[24] EPFL-LAP. 2023. Dynamatic. Retrieved from https://github.com/EPFL-LAP/dynamatic.

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

[26] John Hansen and Montek Singh. 2012. Multi-Token Resource Sharing for Pipelined Asynchronous Systems. In *Proceedings of the 2012 Design, Automation and Test in Europe Conference and Exhibition*. Dresden, 1191–96. https://doi.org/10.1109/DATE.2012.6176674

[27] Advanced Micro Devices Inc. 2021. 7 Series Product Tables and Product Selection Guide (XMP101). Retrieved from https://docs.amd.com/v/u/en-US/7-series-product-selection-guide.

[28] Lana Josipović, Philip Brisk, and Paolo Ienne. 2017. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Transactions on Embedded Computing Systems* 16, 5s (Sept. 2017), 1–19. https://doi.org/10.1145/3126525

[29] 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. https://doi.org/10.1145/3174243.3174264

[30] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative Dataflow Circuits. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, 162–71. https://doi.org/10.1145/3289602.3293914

[31] 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. https://doi.org/10.1145/3373087.3375391

[32] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2021. Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits. *IEEE Circuits and Systems Magazine* 21, 1 (May 2021), 97–118. https://doi.org/10.1109/MCAS.2021.3071631

[33] 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. https://doi.org/10.1109/FCCM53951.2022.9786084

[34] 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. https://doi.org/10.1145/3373087.3375314

[35] 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

[36] Monica S. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation*. Atlanta, GA, 318–28. https://doi.org/10.1145/53990.54022

[37] 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. https://doi.org/10.1109/TVLSI.2008.2005285

[38] Louis-Noël Pouchet. 2012. *Polybench: The Polyhedral Benchmark Suite*. Retrieved from https://web.cs.ucla.edu/~pouchet/software/polybench/.

[39] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of the 41st International Symposium on Computer Architecture*. Minneapolis, MN, 13–24. https://doi.org/10.1145/2678373.2665678

[40] 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. https://doi.org/10.1109/DATE.2012.6176626

[41] 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. https://doi.org/10.1109/FPL57034.2022.00063

[42] Siemens EDA. 2024. ModelSim. Retrieved from https://eda.sw.siemens.com/en-US/ic/modelsim/.

[43] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (June 1972), 146–160. https://doi.org/10.1137/0201010

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

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

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

[47] Jiahui Xu. 2024. Research Artifact of CRUSH: A Credit-Based Approach for Functional Unit Sharing in Dynamically Scheduled HLS. https://doi.org/10.5281/zenodo.14017399

[48] Jiahui Xu and Lana Josipović. 2023. Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification. In *Proceedings of the 42nd International Conference on Computer-Aided Design*. San Francisco, CA, 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323796

[49] 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. https://doi.org/10.1145/3626202.3637570

[50] 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. https://doi.org/10.1145/3543622.3573196

[51] 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. https://doi.org/10.1109/ICCAD.2013.6691121