Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits

Ayatallah Elakhras
EPFL, Lausanne, Switzerland
ayatallah.elakhras@epfl.ch

Riya Sawhney
EPFL, Lausanne, Switzerland
riya.sawhney@epfl.ch

Andrea Guerrieri
EPFL, Lausanne, Switzerland
andrea.guerrieri@epfl.ch

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

Paolo Ienne
EPFL, Lausanne, Switzerland
paolo.ienne@epfl.ch

ABSTRACT

Dynamically scheduled high-level synthesis can exploit high levels of parallelism in poorly-predictable control-dominated applications. Yet, dataflow circuits are often generated by *literal* conversion of basic blocks into circuits interconnected in such a way as to mimic the program's *sequential* execution. Although correct and quite effective in many cases, this adherence to control flow still significantly limits exploitable parallelism. Recent research introduced techniques to deliver data tokens *directly* from producers to consumers and achieved tangible benefits both in circuit complexity and execution time. Unfortunately, while this successfully addressed ordinary data dependencies, the problem of potential dependencies through memory remains open: When no technique can statically disambiguate accesses, circuits must be built with load-store queues (LSQs) which, to reorder accesses safely, need memory accesses to be allocated in the queues in program order. Such in-order allocation still demands control circuitry emulating sequential execution, with its negative impact on parallelization. In this paper, we transform potential memory dependencies into virtual data dependencies and use the new direct token delivery strategy to allocate accesses sequentially into the LSQ. In other words, we exploit more parallelism by constructing control circuitry to emulate exclusively those parts of the control flow strictly necessary for in-order allocation. Our results show that we can achieve up to a 74% reduction in execution time compared to prior work, in some cases, at no area cost.

CCS CONCEPTS

• **Hardware** → **Circuit optimization**; • **Computer systems organization** → *Data flow architectures.*

KEYWORDS

high-level synthesis, dataflow, load-store queue

ACM Reference Format:
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 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3543622.3573050

FPGA '23, February 12–14, 2023, Monterey, CA, USA

ACM ISBN 978-1-4503-9417-8/23/02.
https://doi.org/10.1145/3543622.3573050

1 INTRODUCTION

Recently, there has been some interest in dynamically scheduled *high-level synthesis* (HLS) due to its flexibility in dealing with the unpredictability of program control decisions and the variable latency of some operations. For many typical software applications, this results in performance gains and demands less code refactoring at the cost of a reasonable hardware expense over classic HLS [16]. Generated circuits are referred to as *dataflow* circuits: synchronous circuits with distributed control, obtained by accompanying every piece of data with a pair of handshake signals. Dataflow circuit generation out of an imperative language such as C/C++ starts from the compiler *intermediate representation* (IR) composed of *data flow graphs* (DFGs) and a *control flow graph* (CFG). Nodes in the CFG are blocks of code, or *basic blocks* (BBs); they are connected by control flow arcs representing control decisions. Typically, circuit generation translates the DFGs of individual BBs into separate circuits and then connects the live-out channels of each BB (i.e., signals carrying data that may be used in a later BB) with the live-in channels of the adjacent BBs (i.e., signals potentially requiring data generated in earlier BBs) [4, 14]. This strategy, although correct, is very conservative: data moves along the control flow and, in many cases, arrives late at their destination. Recently, we introduced a different circuit generation strategy for *fast token delivery* that delivers data

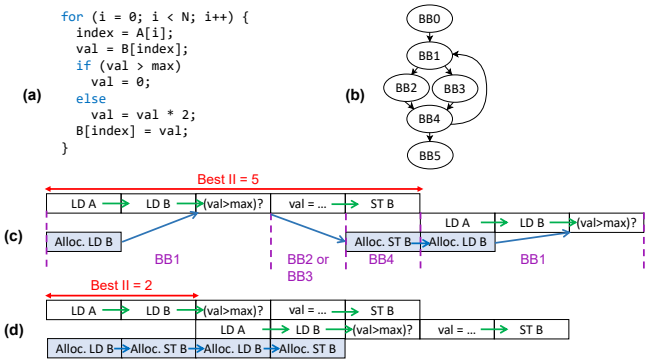

Figure 1: Motivation. (a) A code example requiring an LSQ for `B[]`. (b) Its CFG. (c) Execution schedule of strictly sequential LSQ allocations with limited parallelism due to control dependencies. (d) Our target schedule where LSQ allocations are issued as early as possible, thus improving significantly execution time.

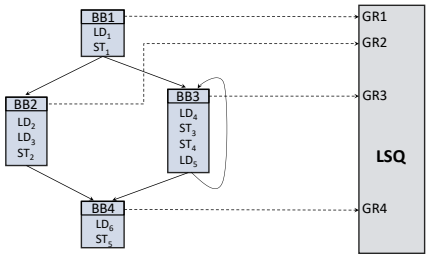

Figure 2: LSQ Allocations. Every time a BB starts, the LSQ allocates slots for the group of accesses (GR) thereby contained in the appropriate order. (Reproduced from Josipović et al. [13].)

as *directly* as possible from its producer to its consumer; each delivery path is affected only by the *necessary* control flow decisions [9]. The produced circuits do not imitate, anymore, global control flow, resulting in faster and smaller circuits.

There is a big snag, though. As in any dynamically scheduled computing system, memory accesses which cannot be statically disambiguated need an ordering decision at runtime: if the addresses happen not to collide, program order can be ignored; otherwise, the original order needs to be enforced. This job is typically delegated, in dataflow circuits as in processors, to hardware *load-store queues* (LSQs); they respond to accesses as quickly as possible but in a semantically correct order. LSQs, to enforce proper ordering, need to know the original program order of the accesses—or, equivalently, LSQs need to allocate entries in the queues as in sequential execution. Josipović et al. [13] used an existing control circuitry emulating the sequential execution to trigger allocation into the LSQs. The difficulty is that the work of *fast token delivery* [9] completely removed such in-order control circuitry and ignored programs requiring LSQs. While it is possible to reintroduce this circuitry for LSQ allocation, this is undesirable since it could partially or completely nullify the advantage of the new circuit generation strategy.

1.1 Straight to the Queue

Consider, for instance, the code in Figure 1a, inspired by a CORDIC-based hyperbolic tangent computation, and its CFG in Figure 1b. The two accesses to `B[]` cannot be disambiguated statically and need an LSQ. Conventional circuit generation [14] would enforce sequential execution in two instances: data dependencies between operations (for elementary correctness) and control dependencies between BBs (to allocate entries into the LSQ in program order); the LSQ would then enforce necessary ordering of memory accesses. This would result, qualitatively, into the execution schedule of Figure 1c, where the green arrows indicate data dependencies and the blue arrows suggest the succession of BBs. Note that BBs are essentially executed sequentially (as suggested below the timing diagram) and this affects the earliest time when the allocation of the store operation to `B[]` can happen: the best *initiation interval* (II) of the loop would be five cycles, with the assumed latencies and no memory conflicts. In substance, the blue BB transitions are a form of loop-carried dependency and determine the initiation interval. One can observe, though, that there is no need to wait for the control flow to reach BB2 or BB3 to perform the allocation of the store, since its eventual execution *does not* depend on the control decision in BB1. Yet, note also that no compiler could move the store above the `if...else` because the store is data dependent on it. What we need is to keep the allocations in order but without following literally the CFG. Figure 1d shows, again qualitatively, what we want to achieve: by representing only the minimal control dependencies across allocations, they can be performed much faster and, in particular, there is no need to hold back the rest of the circuit. The best initiation interval of the loop would now be two cycles and nothing in the circuit would embed the notion of BB. Achieving this execution schedule is the goal of this paper.

2 DATAFLOW CIRCUITS

The key characteristic of dataflow circuits is their distributed control mechanism: Execution of components is triggered by the availability of all input operands. Results are transmitted as soon as recipient components are ready to consume them. *Tokens*, transmitted through *channels*, abstract the exchanges of pieces of data between components; they can be physically implemented through two handshake control signals indicating, respectively, the validity of the data emitted by the sender and the readiness of the recipient. Conversion of imperative programs into circuits is, in principle, not hard. Data dependencies within a DFG are straightforward to implement: a variable becomes a channel and the token exchange protocol ensures that the data consumer is not executed before the data producer has made data available. Things get trickier beyond the BB: Who should get tokens corresponding to live-out variables of BBs? Who can supply tokens corresponding to BB live-in variables? Of course, it depends on the control flow path that the program takes. Josipović et al. [14] simply had tokens follow literally the control flow, connecting dynamically live-outs to live-ins through a sequence of BRANCH nodes literally representing every control flow decision as taken sequentially by the circuit. More recently, our work on *fast token delivery* [9] have developed a strategy to connect directly data producers to all of their potential consumers, irrespective of the BBs they belong to. Essentially, we place a single BRANCH component in-between every producer and potential consumer, and compute the appropriate set of control flow decisions that result in the execution of the consumer—when the path is not active, because control flowed elsewhere, the token is dropped. Therefore, while Josipović et al. explicitly modelled their circuitry after the CFG and used it to deliver tokens across BBs, our *fast token delivery* approach got rid of this circuitry and saved resources and runtime [9].

2.1 Load-Store Queues for Dataflow Circuits

The discussion above covers only direct data dependencies but ignores completely dependencies through memory. For instance, when a store operation writes to a particular location and a subsequent load happens to read the same address, nothing in the circuit designed as above would prevent, in general, the load to execute before the store—and this would result in incorrect execution. In many cases, it might be possible to determine that two accesses can never collide through alias analysis; yet, there are cases where accesses are purely data dependent and thus impossible to disambiguate statically. One could generate relatively complex circuitry to check addresses and keep accesses in order only when addresses match.

(a)

```
for (i = 0; i < N; i++) {
  index = A[i];
  val = B[index];
  if (val > max)
    val = 0;
  else
    val = val * 2;
  B[index] = val;
}
```

(b)

```
done_ST_alloc = dummy;
for (i = 0; i < N; i++) {
  index = A[i];
  done_LD_alloc =
    allocate_LD(done_ST_alloc);
  val = B[index];
  if (val > max)
    val = 0;
  else
    val = val * 2;
  done_ST_alloc =
    allocate_ST(done_LD_alloc);
  B[index] = val;
}
```

Figure 3: Memory Dependencies as Data Dependencies. (a) The original code of Figure 1a. (b) The same code with pseudofunctions (i.e., LSQ allocations) and pseudovariables to guarantee correct ordering.

Josipović et al. [13] delegated these checks to the standard component used for this purpose, namely, an LSQ. Such an LSQ would be almost identical to a classic superscalar processor LSQ, except for the allocation mechanism—that is, for the process used to inform the LSQ of the original order of accesses in the program. While this is trivially implemented by the in-order instruction decode stage of a processor, dataflow circuits do not have anything like a decode phase. To address the issue, the authors first observed that the order of accesses within BBs is known statically and therefore all accesses of a BB can be allocated at once. Next, they exploited the fact that their dataflow circuits still followed closely the control flow of the program; they could then connect the LSQ to existing signals explicitly indicating that a particular BB was reached. Figure 2, from the original paper [13], illustrates the idea: every time the control flows to a BB containing relevant memory accesses, a single control token is sent to the LSQ to allocate a group of slots corresponding to all accesses in the BB (GR in Figure 2), in program order. Although this solution is perfectly correct, the in-order circuitry producing such control tokens is quite inefficient and its removal is the key performance advantage of the *fast token delivery* circuit generation strategy [9]. Our goal is now to produce the correct in-order tokens for LSQ allocations *without* the circuitry by Josipović et al.

3 KEEPING ALLOCATIONS IN ORDER

To summarize, our duty is to generate the specific allocation token for the LSQ every time execution is known to reach one of the critical memory accesses, exactly in the order sequential execution would reach them. Our goal is to generate these tokens as quickly as possible and without holding back the rest of the circuit. We observe two intuitive facts: (1) data dependent operations are always executed in order, by construction and (2) the *fast token delivery* strategy [9] honours these dependencies most efficiently. The idea we use to achieve our goal is fairly simple: Model the potential memory dependencies as ordinary data dependencies. The data exchanged are irrelevant (i.e., the tokens are dataless) but the token exchange itself is essential to maintain order. The operations receiving and producing these control tokens represent our LSQ allocations. Figure 3 illustrates the idea on the code of Figure 1a: Pseudofunctions (`allocate_LD()` and `allocate_ST()`), placed in the BBs containing the memory accesses, represent allocations—the arrival of an input argument is a proof that the memory access will certainly be reached; thus, an allocation token is sent to the LSQ. Pseudofunctions produce pseudovariables (`done_ST_alloc` and `done_ST_alloc`) which bring tokens to the next pseudofunctions and thus keep pseudofunctions executing strictly in order. Essentially, (i) we construct, through data dependencies among pseudovariables, the smallest necessary subset of the original in-order control-flow network and (ii) we use the *fast token delivery* [9] strategy to get an extremely efficient implementation of it, much superior to the direct implementation that strictly follows the CFG.

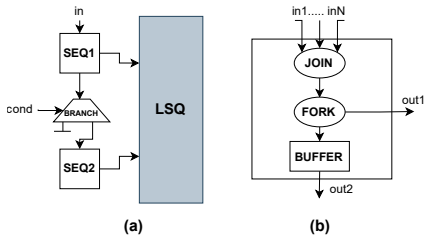

Figure 4: The Sequentializer (SEQ). (a) The typical use of SEQs. (b) Its implementation with dataflow components.

4 INTERFACING CIRCUITS WITH THE LSQ

Based on the intuition of the previous section, we detail now our methodology for interfacing dataflow circuits with LSQs.

4.1 The Sequentializer (SEQ)

To deliver control allocation tokens in the desired order, we introduce a dataflow component which we call *sequentializer* or, in short, SEQ (see Figure 4). In reference to our intuitive presentation of the solution, the SEQ instances are the implementation of the pseudofunctions `allocate_LD()` and `allocate_ST()` in Figure 3: A SEQ has an arbitrary number of input ports (but at least one) and executes only when tokens arrive on *all* ports; each input token represents the confirmation that a particular allocation preceding the current one in program order has been carried out. A SEQ has two output ports; one is used to pass an allocation token to the LSQ and the other to inform successors once the allocation takes place. We will see that SEQs are typically connected in chains or cycles, as suggested in Figure 4a—where appropriate BRANCHes deliver or remove tokens based on control flow. Note that all SEQ ports are dataless and consists only of handshake signals. In general, a SEQ is responsible for allocating a group of accesses.

To reason about the functionality of the SEQ component, consider an example of two groups of accesses $group_i$ and $group_j$ (e.g., two sets of accesses belonging to two BBs). Suppose that the control flow determines that $group_i$ should be allocated first. We would need two SEQs connected as shown in Figure 4a. Functionally, we need SEQ to respect a number of constraints: (a) SEQ must trigger only when all input tokens arrive. (b) Each SEQ must pass the control token to the next one only once the LSQ has accepted the allocation token. If the LSQ applies backpressure, SEQ cannot trigger further allocations through successive SEQs, as these may then be accepted earlier by the LSQ—and thus out of program order. (c) If the LSQ accepts immediately the allocation of both $group_i$ and $group_j$, SEQ must enforce that $group_i$ happens before $group_j$. This would not be necessarily granted, if SEQ had a combinational path from the input ports to the output to successive SEQs.

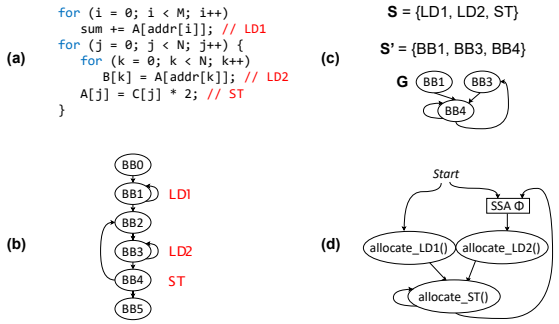

Figure 5: Steps in the Construction of the Allocation Network. (a) A sample code. (b) Its CFG. (c) The sets $\mathcal{S}$ and $\mathcal{S}'$, and the graph $\mathcal{G}$. (d) The elements added to the HLS tool IR.

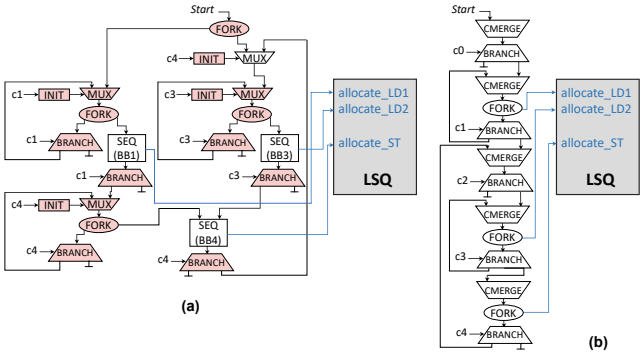

Figure 6: Circuits of the Example in Figure 5. (a) Circuit produced by our approach (shaded in red are the components added for *fast token delivery* [9]: connections between the INIT components and the *Start* signal are omitted). C_i is the condition of BB_i. (b) Circuit mimicking the in-order network of Josipović et al. [14].

We implement SEQ using common elastic components [14], as shown in Figure 4b. The three components are immediately dictated by the constraints above: (a) The first constraint determines the JOIN at the input. (b) The second constraint imposes the type of FORK needed to give the token both to the LSQ and to the successors; such FORK *must* be *lazy*, which implies that the output tokens are valid only when both successors are ready to consume them. (c) The output path towards other SEQs must be sequential or, in other words, the output must be delayed by a cycle though a nontransparent elastic buffer BUFFER. The final circuitry is not fundamentally different from prior work [14], but the use of SEQ helps us build the allocation network.

4.2 Construction of the Allocation Network

Figure 5 shows all steps in the construction of the allocation circuitry for the accesses to `A[]` in the code of Figure 5a. Figure 6 contrasts our circuit to the circuit mimicking the sequential control flow. We will refer to this example throughout the section.

Any classic memory dependence analysis must provide us with two elements: (1) The set $\mathcal{S}$ of memory operations whose access patterns cannot be disambiguated and thus require an LSQ to prevent potential data hazards. (2) The potential dependencies between the individual memory operations in the set $\mathcal{S}$. We impose no particular requirements on this memory analysis and it can be implemented in any way (e.g., polyhedral analysis); simply, all potentially colliding accesses *must* be included in $\mathcal{S}$. The amount of parallelism the circuit will be able to exploit will depend on how conservative (that is, how unable to disambiguate accesses) this memory analysis is. Note that, in general, there will be, for a single program, multiple sets of mutually potentially dependent access—that is, multiple sets $\mathcal{S}$, each associated with a different LSQ. Without loss of generality, we will refer in the sequel to a single set $\mathcal{S}$ and a single LSQ; everything described here must be repeated, independently, for each $\mathcal{S}$. In our example, simply $\mathcal{S} = \{LD1, LD2, ST\}$. From this information, we progressively build our minimal allocation network in four steps.

Clustering. We cluster the different elements of the set $\mathcal{S}$ into groups where all operations within one group will be allocated to the LSQ at once. Like Josipović et al. [13], we consider as a group all operations of $\mathcal{S}$ belonging to the same BB because (1) the order of accesses within one BB can be statically determined and (2) the conditions of execution of all operations within one BB are the same by definition (i.e., every time one operation executes, all of the other operations are guaranteed to execute as well). This results in a set $\mathcal{S}'$ whose elements are essentially those BBs of the program that contain any of the accesses in $\mathcal{S}$. In our simple example, $\mathcal{S}' = \{BB1, BB3, BB4\}$.

Building a dependence graph. We build a directed graph $\mathcal{G}$ whose nodes are the elements of $\mathcal{S}'$. There is an arc between two nodes u and v of $\mathcal{G}$ if the memory analysis reports a possible dependency from any of the $\mathcal{S}$ nodes corresponding to u to any of the $\mathcal{S}$ nodes corresponding to v. Figure 5c shows $\mathcal{G}$ in our example.

Implementing the dependence graph. In the compiler's intermediate representation (IR) of the input program, we add a pseudofunction (representing a SEQ) in each DFG corresponding to the nodes of graph $\mathcal{G}$. For each arc of graph $\mathcal{G}$, we add a pseudovariable (in the format used in the IR); this variable is assigned the value returned by the pseudofunction at the source of the arc and becomes one of the arguments of the pseudofunction at the target of the arc. Obviously, every pseudofunction will have as many arguments as predecessors of the corresponding node in $\mathcal{G}$. If a node in $\mathcal{G}$ has multiple successors, a FORK needs to be inserted, as it is normally the case in dataflow circuits.

Adding initial triggers and final sinks. Some nodes in $\mathcal{G}$ may have no predecessors (i.e., the corresponding accesses are not potentially dependent on any preceding access); since we need a token to ever trigger the corresponding SEQ component, we initialize a pseudovariable at the beginning of the program and we make it an argument of the corresponding pseudofunction (which would have no arguments otherwise). This is equivalent, in circuit terms, to connecting the *Start* signal (a signal that is triggered once at beginning of the circuit's execution) to SEQ through an appropriately conditioned path. Similarly, nodes in $\mathcal{G}$ may not have successors if no further access depends on the corresponding accesses: we simply assign the value returned from the corresponding pseudofunction to a new pseudovariable which we make a live-out variable of the program. In circuit terms, we connect this port to a SINK (and SEQ–SINK pairs will be replaced with simple JOINs). Finally, there are often cases of cyclic dependencies in $\mathcal{G}$: in this case, every node has a predecessor but the corresponding circuit will never

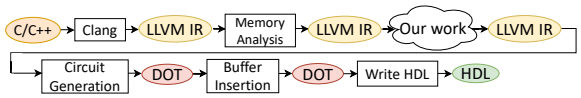

Figure 7: Our methodology inside *Dynamatic* [15]. Nothing is changed but the generation of the allocation circuitry ("Our work").

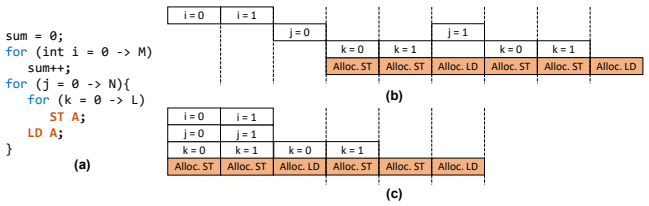

Figure 8: Explanation of Our Performance Gains. (a) A sample code. (b) Schedule using the in-order network [14] that serializes iterations of independent loops. (c) Schedule using our approach for allocating as early as possible by overlapping different loop iterations.

trigger because the cycle does not contain any token. Such a token should be introduced in the cycle by triggering the SEQ which is reached first in the cycle. A straightforward way to implement this is to initialize all pseudovariables at the beginning of the program and let a standard *static single-assignment* (SSA) [7] transformation pass take care of this: essentially, many of the initial assignments will be considered dead code but one or more will carry the initial value to the corresponding pseudofunctions through SSA ϕ's (and, thus, bring an initial token into the cycle). Figure 5d shows the implementation of the dependence graph $\mathcal{G}$ in the compiler IR, including the connections to *Start*, both directly and via an SSA ϕ.

Once we have added all this to the compiler IR, the pseudovariables and pseudofunctions appear as any other element of the original code and can be converted into a circuit with the very same strategy adopted for the rest of the original code elements. *Fast token delivery* [9] is particularly suitable to implement an efficient circuit and can be used. The only differences, compared to the rest of the circuit, are (i) that the implementation of the pseudofunctions will be SEQ components and (ii) that the datapath should be zero bits wide (i.e., control only). Of course, the SEQ nodes will also need to be connected to the appropriate ports of the LSQ. Figure 6a is the resulting circuit (slightly simplified by omitting the self-loop on BB4 for simplicity). One can still recognize the topology of $\mathcal{G}$ but now the components added for *fast token delivery* (shaded in red) are responsible for steering tokens as the execution flow demands. Finally, Figure 6b is the circuit that the method of Josipović et al. [14] would produce that serializes the allocations of LD1 and LD2, while they can happen in parallel with our scheme.

5 EXPERIMENTAL SETUP

We implement our methodology inside *Dynamatic* [15], an open-source C-to-dataflow circuits HLS tool based on LLVM [18], as a new LLVM pass. We use the memory analysis of *Dynamatic* [12] that is based on polyhedral analysis to select the minimum set of memory operations that needs to be interfaced with an LSQ. *Dynamatic* implements both the original circuit generation by Josipović et al. [14] as well the more recent *fast token delivery* strategy [9]; the latter is used to implement the circuits described in Section 4.2. We also use the unmodified backend to insert buffers in the resulting dataflow circuit and generate the RTL description. We interface with the standard LSQs used in *Dynamatic*. Yet, this LSQ design supports only a single allocation per cycle whereas now multiple allocations can be requested at once (for instance, if two groups contain only load operations); therefore, we add round-robin arbiters to pass a single allocation token at a time to the LSQs. As already observed by Cheng et al. [6], such an arbiter could induce deadlock in some peculiar situations; we size the LSQs sufficiently large to avert the problem. Figure 7 shows our work inside *Dynamatic*. We synthesize the generated VHDL netlists with Vivado 2019.2.1 [23] with a clock period constraint of 4 ns, targeting a Kintex-7 Xilinx FPGA. We simulate the designs with ModelSim 10.5c [19] for verification. We measure (1) the cycle count obtained from simulation, (2) the clock period (CP) from the postrouting timing analysis, and (3) resource usage from Vivado after placement and routing. We report the number of DSPs but they are unaffected by our approach.

We collect results for eight kernels with different control structures and memory access patterns, mostly adapted from the PolyBench suite [21]. We benchmark our methodology against two references: (1) The original circuit generation by *Dynamatic*, with its in-order control network that mimics the global control flow. (2) The *fast token delivery* circuit generation [9] that, alone, cannot handle circuits needing LSQs; therefore, we add ourselves the same in-order control network used by *Dynamatic* for LSQ interfacing.

6 EXPERIMENTAL RESULTS

Our implementation is publicly available [8]. Table 1 reports our measurements and Figure 9 contrasts graphically our results and those of the second baseline [9] normalized to the original *Dynamatic* [15]. In the majority of cases, our results Pareto-dominate both baselines. In general, our gains are more on the performance side reflecting the fact that the in-order network that we replace is, in many cases, very detrimental to the speed of execution (in terms of cycles) and is relatively simple (in terms of hardware resources)—especially when compared to the size of the LSQs; thus, the area is unaffected in most cases. Wall time is reduced by more than 70% in one case and the geomean of reductions is around 20%, over both baselines. Figure 8 explains our performance gains by contrasting the allocation time of the strictly in-order memory accesses (Figure 8b) to our approach (Figure 8c) in loop-dominated kernels, which model the structure of our benchmarks, as discussed next.

getTanh is adapted from a CORDIC-based hyperbolic tangent computation. It inspired the motivational example of Section 1.1 and our methodology is essential to achieve a lower loop initiation interval, which in turn shortens significantly wall time execution.

2mm, **3mm** and **covariance** perform multiplication and addition of matrices in complicated nested loop structures. **atax** and **jacobi_1d** have simpler control structures with a single outer loop enclosing two separate inner loops. The in-order control network limits the pipelining of loop-nests and forces the sequential execution of separate loops, as shown in Figure 8b. In removing it, we gain by overlapping the outer and inner loops iterations, as well as parallelizing the separate consecutive loops. This enables faster LSQ allocations, where possible, as shown in Figure 8c.

Table 1: Our circuits contrasted to those produced by the circuit generation methodologies of *Dynamatic* [15] and *fast token delivery* [9], both interfacing LSQs using the in-order control network mimicking the global control flow.

Bench-mark	Cycle count			CP (ns)			Execution time (μs)					LUT			FF			DSP
	[15]	[9]	Ours	[15]	[9]	Ours	[15]	[9]	Ours	vs. [15]	vs. [9]	[15]	[9]	Ours	[15]	[9]	Ours	
getTanh	7,053	6,006	2,009	9.11	8.82	8.44	64.25	52.99	16.95	**-74%**	**-68%**	19,008	19,460	18,994	4,379	4,087	4,058	9
2mm	3,231	2,711	2,498	7.80	8.10	7.77	25.20	21.96	19.41	**-23%**	**-12%**	23,444	22,505	22,190	7,696	6,770	6,715	12
3mm	4,382	4,213	2,498	8.29	8.29	7.87	36.34	34.93	19.66	**-46%**	**-44%**	41,691	40,337	39,742	11,481	10,882	10,667	9
covariance	37,499	36,335	36,307	8.07	7.68	7.08	302.77	279.09	257.13	**-15%**	**-8%**	21,468	21,333	21,345	6,236	5,671	5,694	3
jacobi_1d	1,474	1,320	1,173	6.86	7.22	7.24	10.11	9.54	8.49	**-16%**	**-11%**	20,054	19,120	18,911	4,756	4,335	4,338	3
atax	1,149	991	840	6.80	7.39	6.76	7.81	7.32	5.68	**-27%**	**-22%**	20,559	20,154	20,256	5,424	4,924	4,903	6
triangular	9,895	9,892	9,892	9.18	7.51	7.36	90.79	74.25	72.82	**-20%**	**-2%**	20,581	19,689	20,046	5,082	4,592	4,573	3
histogram	1,015	1,016	1,016	6.92	6.88	6.45	7.02	6.99	6.55	**-7%**	**-6%**	18,916	18,912	19,437	4,236	4,214	4,198	0
geomean										**-27%**	**-20%**							

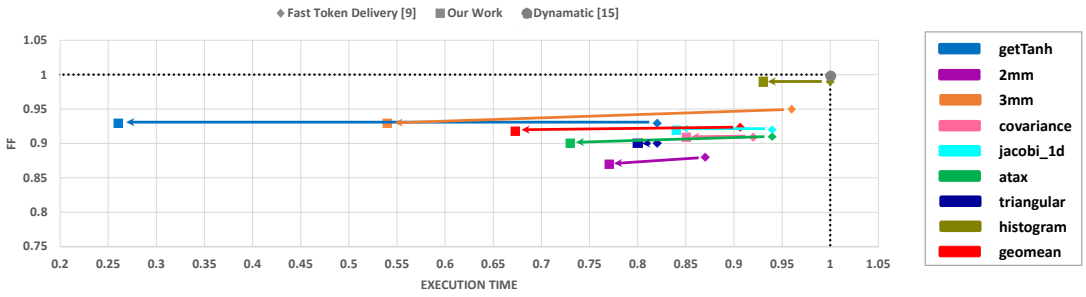

Figure 9: Execution time and FFs in circuits generated by the *fast token delivery* strategy [9] using the strictly in-order network for LSQ allocations once and using our work another time. The results are normalized to those of *Dynamatic* and Pareto-dominate them [15].

histogram computes the histogram of an array and **triangular** computes matrix multiplication of a triangular matrix. Both kernels result in smaller performance gains in comparison to other kernels mainly because they have simpler control structures. We use them to show that in simple control structures our control network will be the same as the classic in-order control network.

7 RELATED WORK

Several authors explored the generation of dataflow circuits from imperative code [4, 10, 14, 17], but the only work interfacing with LSQs is *Dynamatic* [13], our baseline. Cheng et al. [6] proposed a solution to *Dynamatic*'s sequential LSQ allocation through a static analysis which identifies independent loops that can execute in parallel and built a custom, fast allocation scheme for that particular case. Our strategy is not customized to a particular situation and guarantees fast allocation for *any* control structure. Other works either serialize conservatively all memory accesses [10, 11, 17], or only those whose dependencies cannot be resolved statically [5].

There is a huge resemblance between our dataflow circuits and the dataflow architectures explored in the past decades [1, 20]; they, similarly, had no mechanism to ensure the in-order execution of memory operations. To guarantee correct execution, they chose to support only memory operations with constrained semantics and provided special memory structures for them. One example is the write-once I-structures [2] which are initialized to empty and block any read requests when empty; thus, they eliminate the read-after-write data hazards. Another example is the mutable M-structures [3] that become empty after a read and can be rewritten only when empty. Our approach is better suited for the more generic memory semantics of imperative languages. Swanson et al. [22] provided *WaveScalar*, an instruction set architecture for dataflow machines that encodes information about memory dependencies within instructions. Their instruction encodings are similar to our virtual data dependencies; however, to resolve potential ambiguities that arise in complicated control structures, they introduce MEMORY-NOP instructions which add overhead and unnecessarily consume the memory bandwidth, whereas we benefit from the standard SSA algorithm and the generic *fast token delivery* [9] to resolve such ambiguities without imposing overheads.

8 CONCLUSIONS

Dynamically scheduled HLS has some qualitative advantages when it comes to compiling arbitrary, software-oriented applications. This work represents the latest realization showing that even modest improvements in dataflow circuit generation can have a tangible effect on the quality of the results. With little implementation effort, we have shown a novel strategy to allocate slots in the LSQs of dataflow circuits whose speedup gains are considerable (up to 3.8× in one case) at no cost in resources.

ACKNOWLEDGMENTS

This research is partially supported by Huawei.

REFERENCES

[1] Arvind and R. S. Nikhil. Executing a program on the MIT Tagged-Token dataflow architecture. *IEEE Trans. Computers*, 39:300–318, 1990.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, Oct. 1989.

[3] P. S. Barth and R. S. Nikhil. M-structures: Extending a parallel, non-strict, functional language with state. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 538–568, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

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

[5] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[6] J. Cheng, L. Josipović, G. A. Constantinides, and J. Wickerson. Dynamic interblock scheduling for HLS. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022.

[7] R. G. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Symposium on the Principles of Programming Languages*, pages 25–35, Austin, TX, Jan. 1989.

[8] A. Elakhras. Straight LSQ interface. https://doi.org/10.5281/zenodo.7406581, 2022.

[9] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne. Unleashing parallelism in elastic circuits with faster token delivery. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*, pages 253–61, Belfast, UK, Aug. 2022.

[10] N. Gädke-Lütjens. *Dynamic Scheduling in High-Level Compilation for Adaptive Computers*. Ph.D. thesis, Technischen Universität Braunschweig, Braunschweig, Germany, Apr. 2011.

[11] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu. Elastic CGRAs. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 171–80, Monterey, Calif., Feb. 2013.

[12] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne. Shrink it or shed it! Minimize the use of LSQs in dataflow designs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 197–205, Tianjin, Dec. 2019.

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

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

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

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

[17] N. Kasprzyk. *COMRADE—Ein Hochsprachen-Compiler für Adaptive Computersysteme*. Ph.D. thesis, Technischen Universität Braunschweig, Braunschweig, Germany, 2005.

[18] The LLVM Compiler Infrastructure. *http://www.llvm.org*, 2018.

[19] Mentor Graphics. ModelSim, 2016.

[20] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1998.

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

[22] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The WaveScalar architecture. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[23] Xilinx Inc. *Vivado Design Suite*, 2019.