2022 International Conference on Field-Programmable Technology (ICFPT) | 978-1-6654-5336-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICFPT56656.2022.9974425

Load-Store Queue Sizing for Efficient Dataflow Circuits

Jiantao Liu, Carmine Rizzi, and Lana Josipović
ETH Zurich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland

***Abstract*—Dataflow circuits implement dynamic scheduling and have recently been explored as an alternative to standard, statically scheduled high-level synthesis (HLS) solutions. In contrast to static HLS, dataflow circuits resolve memory dependencies during runtime by employing load-store queues (LSQs) at the memory interface. However, LSQs are extremely resource-expensive to implement in a spatial system and may cause notable frequency degradation. Therefore, there is a clear need to minimize their size and complexity, while still allowing the circuit to achieve a high computational rate. So far, designers resorted to manually tuning the LSQ depth (i.e., number of queue entries) to trade off area and performance; yet, this approach is evidently time-consuming and unfeasible for complex designs. In this work, we develop a strategy to automatically determine the most affordable LSQ depths in dataflow circuits while maintaining the best possible circuit throughput. We demonstrate our technique on benchmarks obtained from C code with different memory access patterns and show that it can effectively produce the desired Pareto-optimal design points.**

I. Introduction

In contrast to statically scheduled circuits produced by classic *high-level synthesis* (HLS) tools [31], [5], dataflow circuits [6], [11] implement high-throughput pipelines that adapt their execution dynamically, at runtime, to particular data and control outcomes. Recent HLS efforts explore this flexibility and generate dataflow circuits from high-level code [17], [3]; they achieve significant performance improvements in programs where memory dependencies and control flow cannot be determined at compile time.

One of the key features that makes dynamic scheduling superior to static is its ability to aggressively reorder memory accesses, even when they cannot be statically disambiguated: In cases where memory access patterns are not known at compile time (e.g., because the addresses are unknown or memory dependencies are imposed by statically unpredictable control flow), static HLS must make conservative scheduling assumptions and sequentialize potentially dependent accesses. In contrast, dynamic HLS uses *load-store queues* (LSQs) [16] to resolve memory access collisions at runtime, once the actual memory addresses and control flow decisions become known. This approach ensures that only the accesses that have an actual *read-after-write* (RAW), *write-after-write* (WAW), or *write-after-read* (WAR) dependency are ordered before being issued to memory; all those that are determined independent by the LSQ may execute out of order for high performance.

Although the LSQ is the key component to implementing dynamic out-of-order pipelines, it incurs significant resource costs and clock degradation when implemented on an

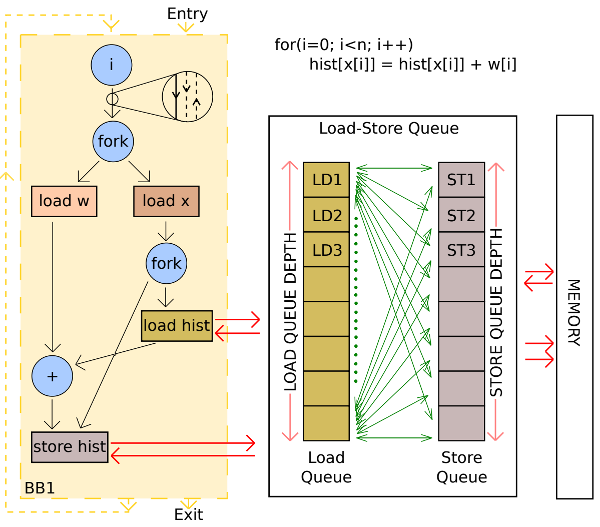


Fig. 1: A portion of a dataflow circuit and its memory interface that contains a load-store queue (LSQ). An LSQ needs a particular depth (i.e., number of queue entries) to sustain the rate of memory requests incoming from the circuit; yet, the LSQ resource requirements and critical path rapidly increase with its depth. Our goal is to systematically determine a resource-affordable LSQ depth that allows dataflow circuits to reap all performance benefits of dynamic scheduling.

FPGA [30]. These detrimental effects are directly dependent on the *depth* of the LSQ: the number of pending memory requests that the LSQ can hold impacts the storage element requirements, but also the complexity of the comparison logic used for identifying dependencies among the pending addresses. Thus, there is a clear need to make the LSQs as small and simple as possible [14], while still reaping all the performance benefits of dynamic scheduling.

In this work, we present a complete strategy to determine the minimal LSQ depth that sustains the maximal throughput of a dataflow circuit and, consequently, achieves the best performance with a minimal resource overhead. Our strategy accounts for a variety of unpredictable behaviors that can occur in dynamically scheduled systems, such as irregular control flow and memory access patterns. On a set of benchmarks obtained from C code, we show that our technique automatically produces Pareto-optimal points that previous HLS solutions could only achieve through extensive manual exploration.

II. The Need for Affordable LSQs

The code in Figure 1 represents a typical case where dynamic scheduling is known to be superior to static scheduling [17], [15]. Notice that the code has an indirect memory access to array `hist`; thus, there is a *possible, but uncertain* memory dependency between a store and any of the consecutive loads to this array. In contrast to static HLS, where each load is conservatively postponed until the previous store executes (thus resulting in low throughput), a dataflow circuit can dynamically adapt the pipeline and resolve memory dependencies at runtime, therefore achieving better performance.

Figure 1 shows the datapath of the dataflow circuit implementing the code above, with its memory interface. The key component to implement the desired dynamic dependency resolution is the LSQ at the memory interface: it receives load and store requests to array `hist` out of order, but ensures that those that are dependent execute in the same order as in the original program. Irrespectively of the implementation details, any LSQ must contain the following: (1) storage elements to hold memory requests that have been sent from the circuit, but not yet issued to memory (indicated as load and store queue in the figure), and (2) logic to compare the load and store addresses and ensure the absence of data hazards (i.e., every green line in the figure represents a comparator for one load-store address pair). It is immediately clear that the resource requirements of the LSQ depend on its depth: in the implementation in the figure, the storage requirements increase linearly, and the comparison logic quadratically with the number of entries of the internal load and store queues [16]. Thus, there is a clear advantage in minimizing this value. On the other hand, employing a depth that is too small may compromise the pipelining capabilities of the dataflow circuit, as the LSQ would not be able to sustain the rate of incoming memory requests. Determining a depth that provides an acceptable area-performance tradeoff is nontrivial, as it depends on a variety of circuit properties (e.g., operation and memory latencies, circuit throughput) as well as its statically undeterminable behaviors (e.g., control flow, memory collisions). In the rest of this paper, we formalize this problem and present a strategy that accounts for all of these factors to automatically determine a performance- and resource-acceptable LSQ depth.

III. Background and Related Work

In this section, we describe dataflow circuits and introduce their relevant properties. We outline the role and typical operating principle of load-store queues in dataflow systems.

A. Dataflow Circuits

Dataflow circuits are built out of units that communicate using a latency-insensitive handshake protocol [6], [11]; the exchange of data (i.e., tokens) and the start of each operation are determined at runtime. Several works described how to translate imperative code into a high-throughput dataflow circuit [3], [17] and we follow one of these methodologies [17].

The circuits we consider group dataflow units into *basic blocks* (BBs), straight pieces of code with no conditionals; all control flow decisions are determined on the edges between the BBs, forming a *control-flow graph* (CFG) [23]. Our circuits respect the following properties [14]: (1) all BBs must start in program order (e.g., in Figure 1, the first execution of $\mathtt{BB}_1$ must start before the second, in this case, identical, $\mathtt{BB}_1$ execution) and (2) each individual dataflow unit processes its tokens in order (e.g., the load to `hist` of iteration 1 must complete before the same load of iteration 2). Yet, different dataflow units may execute out of order for high parallelism (e.g., load to `hist` of iteration 2 might execute before the store to `hist` of iteration 1); thus, the memory interface requires an LSQ to correctly handle potentially out-of-order memory requests.

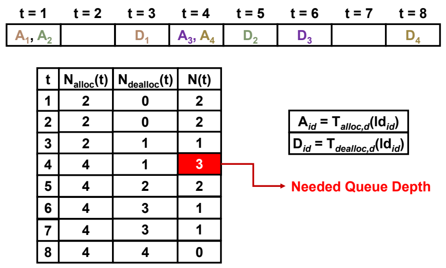
B. Load-Store Queues in Dataflow Circuits

Standard HLS relies on techniques such as modulo scheduling [33], [10], [4] to overlap loop iterations and determine the exact clock cycle in which each operation executes. Since all memory dependencies are (sometimes, conservatively) enforced by the static schedule, an LSQ is not required.

Dynamically scheduled HLS borrows the concept of LSQs from out-of-order processors [25] to dynamically resolve memory dependencies during circuit runtime. To reduce its resource expensiveness, existing approaches rely on various memory analyses to disambiguate memory accesses and optimize the memory interface, from classic alias [2] and polyhedral analysis [26], [13], to custom data dependency analysis for dataflow circuits [14]. Yet, whenever an LSQ cannot be completely removed using these techniques, all these efforts resort to manual LSQ size tuning [17] or choosing an overconservative LSQ size at a resource penalty [8]. The problem of LSQ sizing is remarkably similar to queue sizing problems in networking [22], [12] or FIFO sizing in dataflow circuits [21]; yet, its operating principle is more complex than that of a typical FIFO and its behavior cannot be fully captured using existing queueing models and sizing strategies.

C. Operating Principle of an LSQ

Regardless of its structural details and the exact operating mechanism, any LSQ for a dataflow circuit needs to implement the following functional steps [16]: (1) *Entry allocation.* As soon as a BB starts, entries corresponding to all memory accesses of this BB are allocated into the LSQ in the original program order (e.g., in Figure 1, whenever $\mathtt{BB}_1$ starts, the load and store to `hist` are allocated to the LSQ; the LSQ in the figure shows the state after three $\mathtt{BB}_1$ allocations). Since the BBs start in order and the memory accesses of each BB are statically known, this allocation mechanism conveys the complete in-order memory access sequence of the program to the LSQ and enables it to correctly resolve memory dependencies. (2) *Argument supply.* As the operations in the dataflow circuit execute (possibly, out of order), tokens corresponding to the memory addresses (and data, in case of stores) will eventually be produced and supplied to the LSQ, which adds them to the corresponding entry reserved in the previous step. (3) *Execution.* The LSQ executes the memory operation as soon as the memory port is available, all arguments are supplied, and there are no conflicts with any of the preceding (i.e., older) LSQ entries. (4) *Memory response.* The memory returns to the LSQ the load data or a store completion signal; the LSQ then sends this information to the dataflow circuit. (5) *Entry deallocation*. The LSQ deallocates the entry and makes it available for future requests.


$A_{id} = T_{alloc,d}(ld_{id})$
$D_{id} = T_{dealloc,d}(ld_{id})$
Needed Queue Depth

t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 7	t = 8
A_1, A_2		D_1	A_3, A_4	D_2	D_3		D_4

t	$N_{alloc}(t)$	$N_{dealloc}(t)$	$N(t)$
1	2	0	2
2	2	0	2
3	2	1	1
4	4	1	3
5	4	2	2
6	4	3	1
7	4	3	1
8	4	4	0

Fig. 2: Schedule of a single datapath execution, indicating the allocation and deallocation times of four loads in the datapath. The table below calculates the number of memory requests that the LSQ needs to hold at each point in time (i.e., $N(t)$, calculated based on the number of allocated and deallocated requests). The largest number of items to hold (in this case, 3) corresponds to the minimal depth that the LSQ requires to never stall incoming requests.

A memory access occupies an LSQ entry during the entire process outlined above (i.e., from the time it has been allocated to the LSQ, to the time when it is deallocated). The LSQ will not be a performance-limiting factor of the dataflow circuit execution only if it has sufficient entries empty and available whenever the circuit requests a new allocation; otherwise, the LSQ will create stalls that will propagate through the circuit, postpone the execution of some operations, and degrade performance. Yet, as illustrated in Section II, this entry number directly and substantially impacts the resource requirements of the LSQ. In the rest of this paper, we describe how to systematically determine an LSQ depth that appropriately balances these conflicting requirements and results in circuits that exhibit both area and performance efficiency.

IV. Determining the LSQ Depth

In the previous sections, we highlighted the need to automatically determine the minimal LSQ depth that sustains the desired circuit performance. In this section, we formalize this problem and present a mathematical formulation that allows us to systematically determine an adequate LSQ depth.

A. *LSQ Sizing in a Single Datapath*

We first consider an execution of a single datapath of operations (i.e., a straight piece of code or a single loop iteration); we will generalize our formulation to situations where datapaths overlap (i.e., loop pipelining) in Section IV-B.

To reason about the timing relations of memory accesses, we define the following metrics for a datapath d, all expressed in discrete time units (e.g., clock cycles):

- $T_{alloc,d}(m)$: allocation time of memory access m.
- $T_{dealloc,d}(m)$: deallocation time of memory access m.
- $T_{start,d}$: datapath execution start time.
- $T_{end,d}$: datapath execution end time.

A load-store queue can be characterized as follows:

- $N(t)$: number of enqueued memory requests that the LSQ holds in time t
- D: depth of the queue (i.e., number of LSQ entries).

The LSQ will never be a performance-limiting factor if its depth is such that, at any point in time and regardless of the number of requests that it already holds, it can allocate *all* incoming memory requests. Otherwise, the circuit will stall and its performance will degrade. Thus, the minimal depth of the LSQ is bound by the maximal number of memory requests that the LSQ needs to hold at any time t during the execution:

$$D = max\ N(t), \quad \forall t \in [T_{start,d}, T_{end,d}]. \tag{1}$$

Here, $N(t)$ corresponds to the number of requests that have been allocated into the LSQ before or in time t, but have not yet been deallocated:

$$N(t) = N_{alloc}(t) - N_{dealloc}(t), \tag{2}$$

where $N_{alloc}(t)$ is the number of memory requests that have been allocated into the LSQ before or during t, whereas $N_{dealloc}(t)$ is the number of memory requests that have been deallocated from the LSQ before or during t (thus, in time t, they no longer reserve an LSQ slot and we can deduct them from the current number of items in the LSQ).

The value of $N_{alloc}(t)$ can be expressed as the sum of all memory requests m of datapath d for which $T_{alloc,d}(m) \leq t$:

$$N_{alloc}(t) = \sum_{m \in d} \mathcal{F}(T_{alloc,d}(m), t). \tag{3}$$

Here, $\mathcal{F}(a, b)$ corresponds to the following function:

$$\mathcal{F}(a, b) = \begin{cases} 1, & a \leq b \\ 0, & a > b, \end{cases} \tag{4}$$

which returns 1 when $T_{alloc,d}(m) \leq t$ and 0 otherwise, thus effectively summing up all currently allocated items.

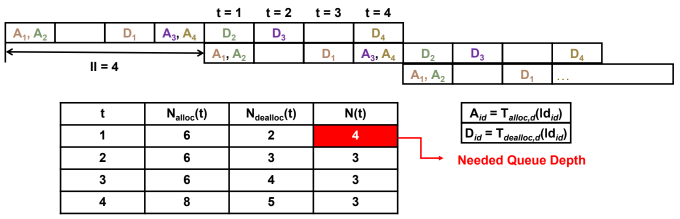
Similarly, we express the number of deallocated requests at time t (i.e., the sum of accesses m with $T_{dealloc,d}(m) \leq t$) as:

$$N_{dealloc}(t) = \sum_{m \in d} \mathcal{F}(T_{dealloc,d}(m), t). \tag{5}$$

These expressions allow us to calculate $N(t)$, as expressed by Equation 2; the largest value of $N(t)$ that is required during the entire datapath execution is the LSQ depth that guarantees maximal performance (see Equation 1).

Figure 2 shows an execution schedule of a datapath containing four loads: ld_1 and ld_2 are allocated in $t = 1$, and ld_3 and ld_4 are allocated in $t = 4$. Therefore, $N_{alloc}(t) = 2$ for $t < 4$ and $N_{alloc}(t) = 4$ for $t \geq 4$. Similarly, $N_{dealloc}(t)$ increases by 1 when each load is deallocated, i.e., in $t = 3$, $t = 5$, $t = 6$, and $t = 8$. The largest number of requests that the LSQ has to hold during the datapath execution is $N(4) = 3$ (as, in $t = 4$, the LSQ still holds ld_2 and must have space to allocate ld_3 and ld_4); thus, the LSQ requires a depth of $D = 3$ to allow all operations to execute as shown in the schedule.

An LSQ can consist of either a single, monolithic queue that holds both loads and stores, or of distinct load and store queues, as shown in Figure 1. The former requires a single depth calculation, where the set of considered memory accesses m includes both loads and stores; in the latter, the same calculation is performed separately for the load and store queue and includes only the load and store accesses, respectively. Consistently with the architecture we use, we calculate the load and store queue depths separately; we will



t	$N_{alloc}(t)$	$N_{dealloc}(t)$	N(t)
1	6	2	4
2	6	3	3
3	6	4	3
4	8	5	3

Fig. 3: Datapath of Figure 2, overlapping with an II of 4 cycles. As shown in the table, memory accesses of two iterations overlap; the LSQ now requires 4 entries to hold all pending memory requests.

demonstrate this in Section VI. Yet, our calculation is perfectly general and applies to any LSQ architecture and organization.

B. LSQ Sizing in Overlapping Datapaths

Our previous formulation describes noncyclic datapaths and nonpipelined loops (as their iterations execute sequentially and form a sequence of independent datapaths whose LSQ requirements can be calculated as above); we here generalize our approach to pipelined loops, where multiple datapaths (i.e., loop iterations) overlap in execution.

We employ the timing relations of Section IV-A to describe a single execution d of a loop iteration. As in the previous formulations, all values are expressed relative to the iteration start time $T_{start,d}$, which avoids the need to calculate absolute execution times (which may be impractical or impossible, e.g., when the loop iteration count is unknown) and allows us to effectively reason about the steady state of the loop (e.g., in Figure 3, $T_{dealloc,d}(ld_1) = 3$ for every iteration d).

The main difference with respect to the previous discussion is that, now, multiple iterations (i.e., datapaths) overlap in execution; we thus define the following metrics:

- $Iter_{max}$: number of overlapping loop iterations in the steady state of the loop (i.e., the number of in-flight loop iterations during a pipelined execution).
- II: loop initiation interval (i.e., the number of clock cycles between the starts of two consecutive iterations).

The datapath latency and loop II determine the number of overlapping executions in the steady state of the pipeline:

$$Iter_{max} = \left\lceil \frac{T_{end,d}}{II} \right\rceil . \qquad (6)$$

Figure 3 repeats the datapath of Figure 2, yet it now executes in a pipelined fashion with an initiation interval $II = 4$. Every two consecutive datapath iterations overlap in execution (i.e., $Iter_{max} = 2$), so their memory accesses might overlap as well—we must account for this effect when sizing the LSQ.

To calculate the sum of requests that the LSQ holds in time t, we need to account for the accesses of prior iterations that might not have been deallocated yet; this is the main difference with respect to our previous formulation. Considering that, at a given time, the LSQ can contain requests from at most $Iter_{max}$ iterations (as all earlier iterations and their memory accesses have already completed), we extend the calculation of allocated and deallocated items as follows:

$$N_{alloc}(t) = \sum_{i=0}^{Iter_{max}-1} \sum_{m \in d} \mathcal{F}(T_{alloc,d}(m) - i \cdot II, t), \qquad (7)$$

$$N_{dealloc}(t) = \sum_{i=0}^{Iter_{max}-1} \sum_{m \in d} \mathcal{F}(T_{dealloc,d}(m) - i \cdot II, t). \qquad (8)$$

As $T_{alloc,d}(m)$ and $T_{dealloc,d}(m)$ of each memory access are expressed relative to the start of the corresponding loop iteration, the term $i \cdot II$ offsets the times of the memory accesses from prior iterations based on the iteration distance (the smallest i corresponds to the latest iteration). This method resembles the modulo scheduling formulations [33]. We can then calculate $N(t)$, as specified by Equation 2.

Consider the second iteration in Figure 3, representing the pipeline's steady state. In its first execution cycle (i.e., $t = 1$ with respect to this iteration start), two accesses of the prior iteration ($i = 1$) remain allocated in the LSQ (e.g., $T_{alloc,d}(ld_3) - i \cdot II = 4 - 4 < 1$ and $T_{dealloc,d}(ld_3) - i \cdot II = 6 - 4 > 1$), resulting in an LSQ depth requirement of $D = 4$.

Notice that, in a regular pipeline, the execution repeats every II cycles—the behavior of the loop in $T_{start,d} + II$ is identical to $T_{start,d}$ (e.g., in Figure 3, the pipeline repeats the same behavior every $II = 4$). Therefore, the search for the maximal number of enqueued items (and, thus, the LSQ depth calculation) can be reduced to that time interval only:

$$D = max\ N(t), \quad \forall t \in [T_{start,d}, T_{start,d} + II - 1]. \qquad (9)$$

This formulation allows us to reason about the LSQ depth in any dataflow circuit: the entire execution can be decoupled into multiple overlapping and nonoverlapping datapaths analyzed independently using Equations 7 and 8. These equations naturally capture the behavior of a single, straight datapath as well: for $Iter_{max} = 1$ and $II = T_{end,d}$, they reduce to Equations 3 and 5. Different execution portions may have different LSQ requirements; defining the LSQ depth as the largest among all required depths guarantees the best performance during the entire circuit execution. One could tune the depth by omitting the depth calculation on execution portions that do not significantly contribute to performance. Similarly, one could trade off area and performance differently by modifying the desired II value in the equations above (e.g., increasing the II would lower the performance, but require a smaller LSQ, as fewer loop iterations and memory accesses would overlap).

Our formulation naturally captures situations where the loop II changes due to memory collisions (i.e., the LSQ might stall the pipeline until the collision is resolved and, temporarily, increase the loop II [19]): In Equations 7 and 8, the II can be adapted to every pair of neighboring iterations (instead of considering a constant value, as specified by the equations above). Similarly, our formulation naturally captures situations where not all loop iterations are identical (e.g., due to control flow statements inside the loop): instead of considering $Iter_{max}$ identical datapaths, one can simply account for $Iter_{max}$ different datapaths with different sets of memory accesses. Of course, the control flow and the exact datapath

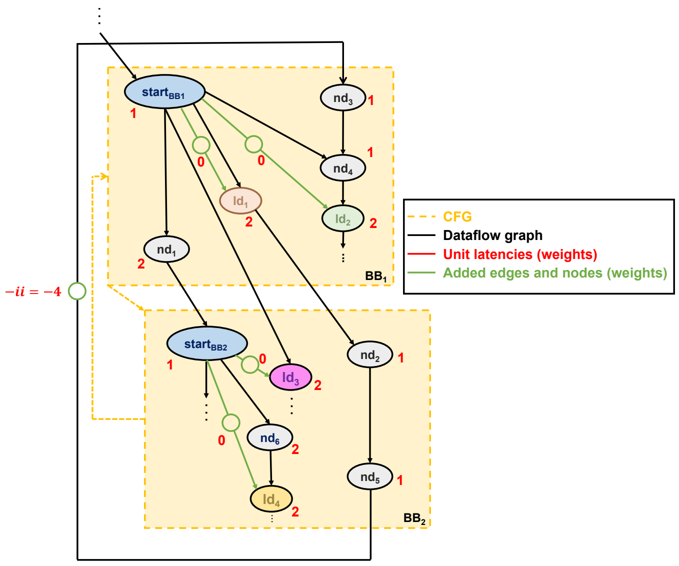


Fig. 4: Dataflow graph with nodes (corresponding to dataflow operators), annotated with their latencies. We use the timing relations between the executions of particular dataflow nodes to calculate memory access allocation and deallocation times.

sequence might not always be known as *a priori*; in such cases, one could assume an average case or rely on application profiling [21], [9]. We will examine these effects in Section VI.

V. Determining the Timing of Memory Accesses

Our LSQ depth calculation relies on the information about when each memory request is allocated and deallocated from the LSQ; we here discuss how we systematically obtain these values for any memory access and dataflow circuit topology.

As discussed in Section III-A, we consider a graph of dataflow nodes organized into BBs; each node is characterized by its sequential delay (i.e., latency). A possible dataflow graph that exhibits the behavior of Figure 3 is shown in Figure 4. The dataflow nodes are grouped into two BBs; $\mathtt{BB}_1$ contains ld_1 and ld_2, and $\mathtt{BB}_2$ contains ld_3 and ld_4. Although not shown in the figure, all loads are connected to the LSQ: they receive an address from the predecessor node, store it in the LSQ, and issue the data to the successor node when the LSQ provides it. Nodes $start_{\mathtt{BB}_1}$ and $start_{\mathtt{BB}_2}$ represent the first (i.e., earliest) executed operation of the corresponding BB and the start of BB execution. Nodes nd_1 to nd_6 represent other dataflow units; their functionality is not relevant for this discussion, but their latencies may impact the memory access timing.

We define path $\mathcal{P}(src, dst)$ as the longest weighted path of nonrepeating nodes from src to dst, where node weights correspond to their operation latencies. The latency of this path, $Lat_{\mathcal{P}}(src, dst)$, is the sum of the weights of all nodes in the path; it dictates the time between the start of src (i.e., src is triggered by its predecessors) and the execution completion of dst (i.e., dst produces an output for its successors). In Figure 4, $Lat_{\mathcal{P}}(nd_3, ld_2) = 4$.

As discussed in Section III-B, an access m is allocated into the LSQ when its BB, $\mathtt{BB}_m$, starts. We can thus calculate the allocation time of m with respect to the datapath start as:

$$T_{alloc,d}(m) = Lat_{\mathcal{P}}(start_{\mathtt{BB}_{\mathtt{init}}}, start_{\mathtt{BB}_m}), \tag{10}$$

where $\mathtt{BB}_{\mathtt{init}}$ is the initial BB of the datapath (in a cyclic CFG, it corresponds to the loop header BB).

Consider the example in Figure 4: ld_1 and ld_2 are allocated when $start_{\mathtt{BB}_1}$ triggers; this is, coincidentally, also the datapath start (i.e., $start_{\mathtt{BB}_{\mathtt{init}}}$), thus $T_{alloc,d}(ld_1) = T_{alloc,d}(ld_2) = Lat_{\mathcal{P}}(start_{\mathtt{BB}_{\mathtt{init}}}, start_{\mathtt{BB}_1}) = 1$. As ld_3 and ld_4 are in $\mathtt{BB}_2$, $T_{alloc,d}(ld_3) = T_{alloc,d}(ld_4) = Lat_{\mathcal{P}}(start_{\mathtt{BB}_{\mathtt{init}}}, start_{\mathtt{BB}_2}) = 4$.

An access is deallocated from the LSQ after all its arguments arrive, the LSQ executes the access, and the memory subsystem processes it (see operating steps (2) to (4) from Section III-B). Thus, $T_{dealloc,d}(m)$ can be expressed as:

$$T_{dealloc,d}(m) = T_{arg}(m) + Lat_{LSQ} + Lat_M, \tag{11}$$

where $T_{arg}(m)$ is the arrival time of the latest argument of m, Lat_{LSQ} is the latency of the LSQ execution, and Lat_M is the memory latency. Lat_{LSQ} and Lat_M are known values that depend on the memory architecture; their sum corresponds to the latency of each m node in the dataflow graph (e.g., in Figure 4, we assume Lat_{LSQ} = 1 and Lat_M = 1, so the latency of all loads is 2, as shown in the figure).

We can therefore calculate the deallocation time of m as:

$$T_{dealloc,d}(m) = Lat_{\mathcal{P}}(start_{\mathtt{BB}_{\mathtt{init}}}, m). \tag{12}$$

In Figure 4, $T_{dealloc,d}(ld_1) = Lat_{\mathcal{P}}(start_{\mathtt{BB}_{\mathtt{init}}}, ld_1) = 3$; in other words, the load returns the data and is deallocated from the LSQ 3 cycles after the datapath starts.

The dataflow circuit construction strategy we use defines the $start_{\mathtt{BB}}$ nodes and ensures that there is always a directed path between them (as this network is built exactly for allocating memory accesses into the LSQ [20]). Yet, there are two key pieces of information that the dataflow graph does not necessarily contain and that we need to supply ourselves:

(1) *Allocation precedes memory access execution.* There may be no dataflow edge between m and $start_{\mathtt{BB}_m}$. Yet, m can be executed by the LSQ only after $start_{\mathtt{BB}_m}$, when an LSQ slot for m has been allocated. To represent this ordering, we connect all memory nodes with the corresponding $start_{\mathtt{BB}}$ nodes using an additional edge (see green edges in Figure 4). In this figure, there is no dataflow edge connecting $start_{\mathtt{BB}_2}$ and ld_3. Although $start_{\mathtt{BB}_1}$ directly provides the argument (i.e., address) to the load, it can be accepted and the load executed only after $\mathtt{BB}_2$ starts. The added edge between $start_{\mathtt{BB}_2}$ and ld_3 conveys this order and creates a new path that accurately reflects the deallocation time of the load as $T_{dealloc,d}(ld_3) = Lat_{\mathcal{P}}(start_{\mathtt{BB}_{\mathtt{init}}}, ld_3) = 6$.

(2) *Back edge orders operations of different loop iterations.* Whenever a path between src and dst includes the back edge of the loop, it does not impose a delay between src and dst of the same datapath execution (as all other paths do); in contrast, it introduces a delay between the execution of src of one iteration and the execution of dst from the succeeding iteration. To account for this effect, we annotate the back edge of the loop with a latency of $-II$ (this strategy qualitatively corresponds to the time offsets in Equations 7

and 8). Consider ld_2 in Figure 4. The dataflow nodes on the cyclic path from $start_{\mathtt{BB}_1}$ to ld_2 have a weighted latency sum of 9; however, this path represents the delay between the execution of $start_{\mathtt{BB}_{\mathtt{init}}}$ from the previous iteration and the deallocation of ld_2 from the current iteration. To obtain the correct distance within the current iteration, we need to deduct from this delay the distance between the two consecutive executions of $start_{\mathtt{BB}_{\mathtt{init}}}$ (corresponding to the II of 4), which is exactly what our negative weight on the back edge achieves (i.e., $T_{dealloc,d}(ld_2) = Lat_{\mathcal{P}}(start_{\mathtt{BB}_{\mathtt{init}}}, ld_2) = 5$).

These two insights allow us to accurately reason about the allocation and deallocation times of all memory accesses (the times obtained for the loads of Figure 4 are as illustrated in Figure 3). We can directly exploit this information to calculate the LSQ depth, as described in Section IV.

The timing calculation assumes that all latencies are known and fixed. In a dynamically scheduled system, this is not always the case: Some units may take a variable time to compute a result [24], [29]. Similarly, a memory interface may take a variable number of cycles to complete a memory request (e.g., in a multi-level memory hierarchy, Lat_M may be variable). These effects can impact the values of $T_{alloc,d}(m)$ and $T_{dealloc,d}(m)$ and, consequently, the LSQ depth. Consistently with our optimization objective and existing dataflow performance optimization approaches [19], whenever a latency is variable, we assume the latency value that guarantees the best possible throughput and, thus, performance. Other timing estimates can be easily included into our strategy (e.g., by assuming an average or most frequent latency value).

VI. Evaluation

In this section, we evaluate the effectiveness of our LSQ sizing strategy as well as its impact on area and timing.

A. Methodology and Benchmarks

We incorporated our sizing strategy in Dynamatic [18], an open-source HLS tool that produces dataflow circuits from C/C++. Our work is an open-source plugin for Dynamatic (the benchmarks are included) [1]. Dynamatic employs various memory analyses to simplify the memory interface [14]; we apply our LSQ sizing strategy on those LSQs that the tool determines necessary for correctness. We use the output dataflow graph of Dynamatic, already annotated with unit latencies, to calculate the memory access timing, as discussed in Section V. We then calculate the LSQ depth as presented in Section IV-B; consistently with the LSQ architecture that we employ [16], we calculate the depth of the load and the store queue separately. We incorporate the LSQ of the appropriate depths into the RTL description of Dynamatic's dataflow circuit.

We evaluate kernels from standard HLS suites [28], [27] and recent works [17], [7], with different memory access patterns and control flow: (1) *Bicg* and *Gaussian* have statically determinable memory dependencies in the innermost loop of their loop nests. (2) *Histogram* and *Matrix Power* exhibit irregular memory access patterns; their memory dependencies are determined by the input data (see example in Figure 1). (3) *GetTanh*

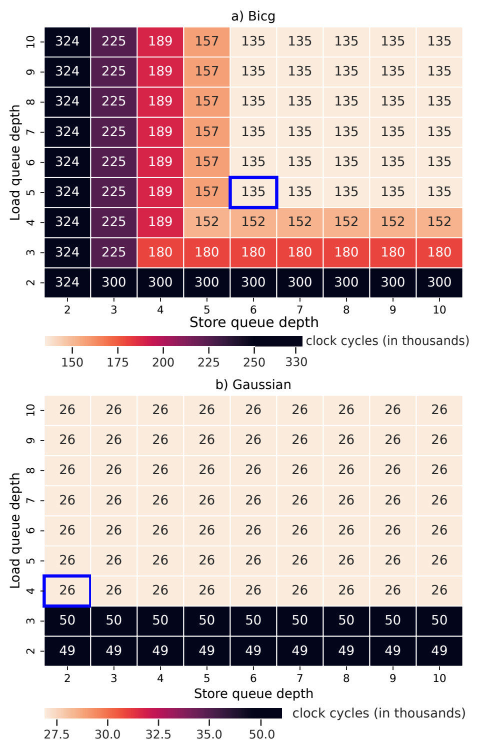


Fig. 5: LSQ depth exploration in regular benchmarks. In both benchmarks, our LSQ sizing strategy successfully identifies the minimal load and store queue depths that achieve the best possible clock cycle count (i.e., the points with blue frames).

has variable control flow (i.e., an if-else statement inside a loop) and possible, input-dependent, memory dependencies.

We investigate the benchmarks with data-dependent behavior under the following conditions: (1) there are no dependencies among any of the accesses (*No Collisions*), (2) there is a RAW dependency among the accesses of all odd loop iterations (*Half Collisions*), and (3) there is a RAW dependency among the accesses of all consecutive loop iterations (*All Collisions*). When the circuit behavior is perfectly predictable (i.e., *Bicg* and *Gaussian*), the performance analyzer of Dynamatic provides the exact loop II; in cases with irregular dependencies, it calculates the best achievable II (corresponding to the *No Collisions* case). We calculate the II of the other two cases based on the best-case II and memory dependency distance. When all iterations have a dependency, there is a single II that we incorporate into Equations 7 and 8; when half of the iterations are dependent, these equations use two II values interchangeably, as discussed in Section IV-B.

We use ModelSim for functional verification and for measuring the clock cycle count of the circuit execution. We use Vivado [32] (targetting a Kintex-7 Xilinx FPGA) to determine the post-place-and-route clock period and resources (i.e., LUTs, FFs) of the final dataflow circuits with the LSQs.

B. LSQ Depth Exploration

In this section, we investigate the effectiveness of our LSQ sizing strategy in identifying the minimal LSQ depths that achieve the best performance (i.e., clock cycle counts).

We explore the number of clock cycles that our benchmarks require to execute while varying the LSQ depths. We represent our exploration results as heatmaps in Figures 5 and 7, where the x- and y-axis show the store and load queue depths, respectively, and the value in each heatmap cell represents the clock cycle count; lighter cell colors represent a lower clock cycle count. The purpose of our exploration is to investigate whether the point identified by our strategy is the lowest and leftmost (i.e., smallest queue depths) among the lightest points in the heatmap (i.e., lowest clock cycle count).

Figure 5 shows the exploration results for *Gaussian* and *Bicg*. We highlight the points found by our strategy with a blue frame. In both cases, our approach successfully identified the minimal depths that achieve the lowest cycle count (e.g., in *Bicg*, the load and store queue of depth 5 and 6 are the minimal depths that allow the circuit to execute in 135k clock cycles; all smaller queues cannot sustain the incoming rate of memory requests and, thus, increase the total clock cycle count). This indicates the successfulness of our sizing strategy: it automatically identifies the desired Pareto points and, most importantly, removes the need for extensive manual exploration of all design points shown in the figures.

Figure 7 explores the benchmarks with irregular behaviors in the same manner and for different data distributions. The RAW dependencies modify the II, which directly impacts the LSQ requirement: The cases with no dependencies have the best possible II and the largest number of iterations overlap; consequently, the LSQ requires the largest depth to hold their memory requests. As the number of collisions increases, the II increases as well (as evident from the increasing clock cycle counts), therefore reducing the LSQ depth requirements. Our sizing strategy accurately follows these trends and identifies the minimal LSQ depths that achieve the best cycle count for each particular execution scenario.

Although we are here able to fully benefit from our sizing strategy for all data distributions, this might not always be the case: the distribution might be unknown or more irregular than we assume, which is simply the nature of the unpredictability of dynamic scheduling. The fact that our depth calculation can correctly identify the appropriate depth range and accurately supports different data estimates indicates its broad usability and easy adaptation to various area-performance targets.

C. Area and Timing Analysis

In the previous section, we demonstrated the ability of our approach to identify the minimal LSQ depths that achieve the best clock cycle counts. We now investigate the impact of these findings on the circuit's resources and total execution time.

Figure 6 plots the resources (i.e., LUTs, FFs) and execution time (i.e., the product of the clock cycle count and achieved clock period) of three design points of each benchmark: the point with the minimal LSQ depths (*Min Depth*, corresponding to the leftmost and lowest point of the heatmaps from the previous section), our LSQ depths (*Opt Depth*, the highlighted blue points in the heatmaps), and the largest explored LSQ depths (*Max Depth*, the top- and rightmost points). The clock period (CP) of each design point is annotated next to it.

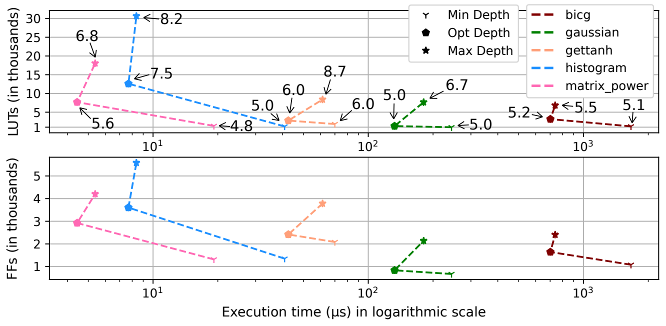


Fig. 6: Resources and execution time of our benchmarks, incorporating LSQs with various depths (i.e., minimal depth, maximal depth, and the depth our strategy identifies). Our points are always Pareto-optimal and achieve the best performance with affordable resources.

The resource impact of the LSQ is immediately evident: *Max Depth* designs require up to 25.7× more LUTs and 4.1× more FFs than the corresponding *Min Depth* designs (these overheads worsen for even larger LSQs that we did not explore here). The *Min Depth* designs suffer in total performance as the LSQ is not able to sustain the rate of incoming memory requests; we already observed this effect in Section VI-B; our points increase the LSQ depth sufficiently to achieve the minimal clock cycle count and, therefore, improve the performance. Interestingly, the execution time of the designs with the largest explored LSQs increases with respect to our points: although the *Opt Depth* and *Max Depth* designs achieve an identical clock cycle count, the critical path of the LSQ deteriorates with the depth, causing an increase in the circuit's clock period and, consequently, the total execution time. Therefore, our designs Pareto-dominate the *Max Depth* designs. In addition to the significant resource variabilities of different LSQ solutions, this CP degradation is another motivator to minimize the LSQ depth; both aspects indicate the importance of our strategy in making dataflow circuits obtained from C code area- and timing-affordable.

VII. Conclusions

Load-store queues are a fundamental component of a dataflow system, as they allow memory accesses disambiguation at runtime and enable dynamic schedule adaptation to the presence or absence of memory dependencies. Yet, LSQs are extremely expensive to implement in a spatial context. The main factor that impacts their resource cost and clock degradation is the queue depth, i.e., the number of pending memory requests that the LSQ can hold; this parameter determines the queue storage requirements, complexity of the address comparison logic, and its critical path. In this work, we present a methodology to calculate the LSQ depth that minimizes the resource requirements of the LSQ while maintaining all performance benefits of dynamic scheduling; our technique removes the need for manual LSQ tuning and enables us to automatically identify Pareto-optimal design points. We believe that this contribution makes an important step in building memory interfaces suitable for dataflow designs obtained from high-level code, thus making this HLS paradigm more attractive and viable in a variety of practical cases.

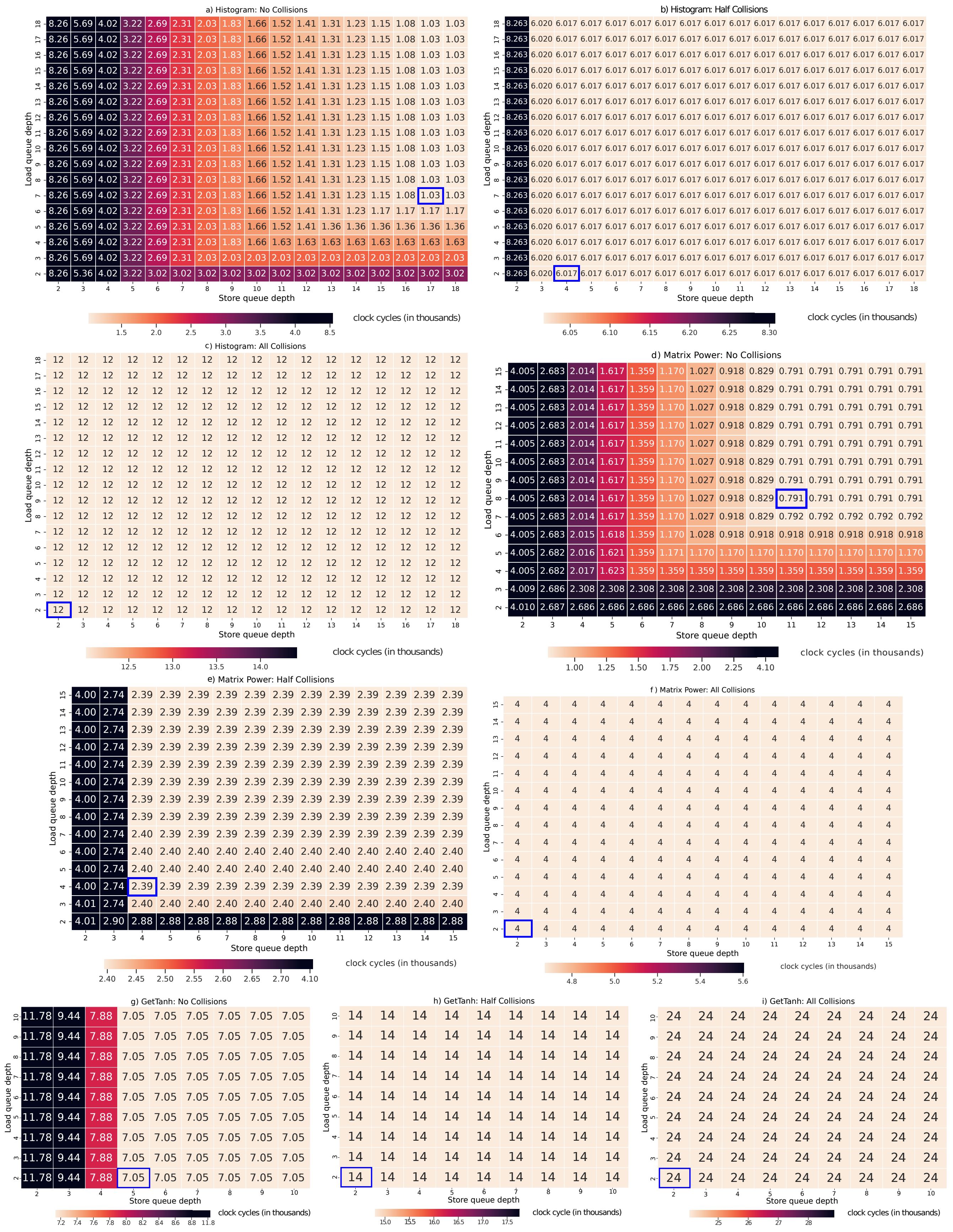


Fig. 7: LSQ depth exploration in irregular benchmarks. In all benchmarks, our LSQ sizing strategy successfully identifies the minimal load and store queue depths that achieve the best possible clock cycle count (i.e., the points with blue frames).

References

[1] LSQ sizing plug-in , https://doi.org/10.5281/zenodo.7232825, Oct. 2022.

[2] J. R. Appel and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, first edition, 2001.

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

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

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

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

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

[8] J. Cheng, L. Josipović, J. Wickerson, and G. A. Constantinides. Dynamic inter-block scheduling for HLS. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022. To appear.

[9] J. Cheng, J. Wickerson, and G. A. Constantinides. Probabilistic scheduling in high-level synthesis. In *Proceedings of the 29th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 195–203, Orlando, FL, May 2021.

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

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

[12] W. J. Gordon and G. F. Newell. Closed queuing systems with exponential servers. *Operations Research*, 15(2):254–265, Mar. 1967.

[13] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly - polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*, pages 1–6, Chamonix, Apr. 2011.

[14] 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.

[15] L. Josipović, P. Brisk, and P. Ienne. From C to elastic circuits. In *Proceedings of the 51st Annual Asilomar Conference on Signals, Systems, and Computers*, pages 121–25, Pacific Grove, CA, Nov. 2017.

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

[16] 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.

[17] 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, CA, Feb. 2018.

[18] 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, CA, Feb. 2020.

[19] L. Josipović, A. Guerrieri, and P. Ienne. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine*, 21(1):97–118, May 2021.

[20] L. Josipović, A. Guerrieri, and P. Ienne. From C/C++ code to high-performance dataflow circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):2142–55, July 2022.

[22] D. G. Kendall. Stochastic process occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, Sept. 1953.

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

[24] M. Olivieri. Design of synchronous and asynchronous variable-latency pipelined multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(2):365–376, Apr. 2001.

[25] I. Park, C. L. Ooi, and T. Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 411–22, San Diego, CA, Dec. 2003.

[26] L. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 29–38, Monterey, CA, Feb. 2013.

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

[28] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.

[29] A. K. Verma, P. Brisk, and P. Ienne. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1250–55, Munich, Mar. 2008.

[30] H. Wong, V. Betz, and J. Rose. Efficient methods for out-of-order load/-store execution for high-performance soft processors. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 442–45, Kyoto, Dec. 2013.

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

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

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