

Out with LSQs: Custom Circuits for Memory Access Reordering in Dynamic HLS

Rouzbeh Pirayadi
EPFL
Lausanne, Switzerland
rouzbeh.pirayadi@epfl.ch

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

Mirjana Stojilović
EPFL
Lausanne, Switzerland
mirjana.stojilovic@epfl.ch

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

Abstract

Circuits generated by dynamically scheduled *high-level synthesis (HLS)* outperform static counterparts when execution-dependent operation reordering is possible. A key opportunity are potential memory dependencies that static analysis cannot rule out: Statically scheduled circuits must assume a worst-case scenario, enforcing conservative in-order execution even when unnecessary. In contrast, dynamically scheduled circuits use *load-store queues (LSQs)* to detect access collisions at runtime and reorder accesses whenever possible. While effective, LSQs are costly in both area and timing due to their size and critical path. We argue that LSQs in application-specific circuits are overly generic, checking all ordering relationships instead of only the necessary ones. Instead of relying on LSQs, our approach generates dataflow circuits that enforce only essential dependencies, reducing resource usage while preserving execution time. We implemented our methodology in an open-source, state-of-the-art dynamic HLS compiler and compared it against optimized LSQs. Our solution exploits the same reordering opportunities to achieve consistently Pareto-optimal solutions in terms of wall-clock time and resources. On average, we obtain similar execution times (2% better) while reducing resources by 37%, offering a superior alternative for high-performance dataflow circuit synthesis.

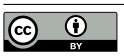
CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; **Reconfigurable logic applications**.

Keywords

high-level synthesis, dataflow circuits, load-store queue

ACM Reference Format:
Rouzbeh Pirayadi, Ayatallah Elakhras, Mirjana Stojilović, and Paolo Ienne. 2026. Out with LSQs: Custom Circuits for Memory Access Reordering in Dynamic HLS. In *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '26), February 22–24, 2026, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3748173.3779204



FPGA '26, Seaside, CA, USA

ACM ISBN 979-8-4007-2079-6/2026/02
https://doi.org/10.1145/3748173.3779204

1 Introduction

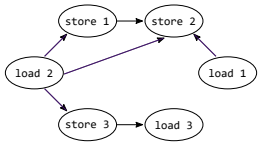
In recent years, *untagged dataflow circuits* (or *elastic circuits*, as they have sometimes been called [10, 11]) have experienced a renaissance [13, 14, 19, 23], particularly in the automatic conversion of C programs into digital circuits—a process typically referred to as *high-level synthesis (HLS)*. Their defining advantage lies in their ability to adapt execution to runtime events rather than follow a predefined schedule. A significant source of variability in many practical applications comes from memory dependencies that cannot be statically disambiguated. This issue has long been a challenge for statically scheduled processors (e.g., the `ld.a` instruction in Itanium [20]) and may have played a substantial role in their complete extinction in general-purpose systems. The problem with some memory dependencies is that their existence can only be determined at runtime; only if they do not occur can memory accesses be reordered to improve execution speed. Static approaches are forced to assume the worst-case scenario and generate a fixed, slower schedule that respects the access order specified in the original program. The inability to dynamically reorder memory accesses is, in practice, a major drawback of traditional statically-scheduled HLS tools.

The standard approach to disambiguating memory accesses in dynamically scheduled processors is to implement *load-store queues (LSQs)* [9, 30, 33, 38]. These circuit structures essentially compare all pending loads with pending stores (specifically, with those stores that precede them in program order) to determine whether a load can safely be issued to memory or if a store's future value should be forwarded to the load. A paper by Josipović et al. adapted these ubiquitous structures for spatial circuits, such as those generated by dynamically scheduled HLS tools [22]. The key contribution of that work was the introduction of a mechanism to communicate the program order of memory accesses to the LSQs, overcoming the lack of the in-order instruction fetch and decode phases typically found in processors. This challenge addressed, LSQs for spatial circuits could be designed in much the same way as those for commercial processors. In fact, an LSQ generator (written in Python and emitting VHDL) is at the core of *Dynamatic*, an open-source dynamically scheduled HLS tool [1, 24].

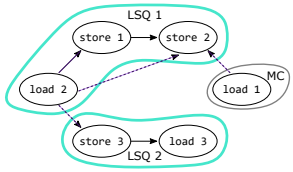
In this paper, we question whether a standard LSQ is the best choice for custom circuits generated by HLS. While LSQs are remarkably efficient and designed to handle any scenario of potential dependencies, their generality comes at a significant cost. We demonstrate that it is entirely possible to design custom dataflow circuits that achieve the same memory access disambiguation as LSQs and, in most cases, this is achieved with lower area overhead and, often, a shorter critical path. We clarify and further motivate this idea in Section 2. Section 3 provides the necessary background

```
a = 0, mid = N / 2;
b = x[0];              // load  1
for (i = 2; i < mid; i++) {
  x[i - 1] = 5;        // store 1
  a = x[i + 1];        // load  2
  x[i - 2] = a + b;} // store 2
for (j = mid; j < N - 1; j++) {
  c = x[j - 1];        // load  3
  x[j] = a + c;}       // store 3
```

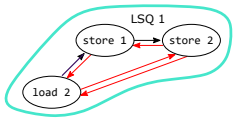
(a) Example code



(b) Initial



(c) Advanced analysis



(d) Edges considered by LSQ 1

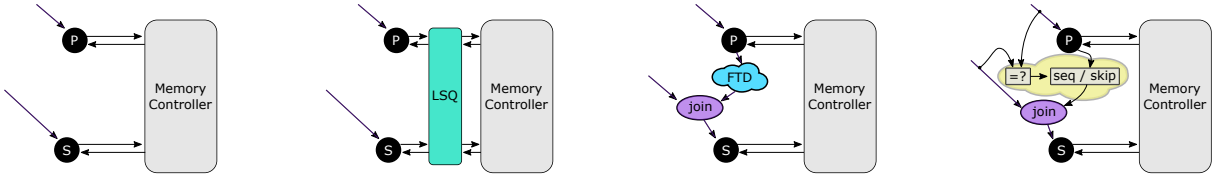
Figure 1: Memory Dependency Graphs. (a) Code example. (b) Dependency graph after standard static analysis; each edge $i \rightarrow j$ means accesses from memory operator i should execute in program order with those from operator j on collision. (c) Dashed edges are automatically respected in dataflow circuits, and can thus be removed [21]. The operations corresponding to any connected component of the remaining graph can be mapped to a separate LSQ. (d) LSQ 1 is unaware of the residual edges within its induced subgraph, so it must conservatively check every access against all pending predecessor and successor instances.

on dataflow circuits before we introduce our design approach in Section 4. Section 5 addresses a specific critical challenge inherent to dataflow circuits and Section 6 qualitatively compares our final design to an LSQ. Our experimental setup and results are presented in Section 7 and Section 8, respectively, followed by a discussion of related work in Section 9. Finally, we conclude in Section 10.

2 Respecting Memory Access Order

Reordering independent operations is crucial for efficient execution of sequential programs. Memory accesses, however, are more challenging to reorder because their independence is determined by the memory locations they access: if a store precedes a load in the code and both access the same memory location (i.e., their addresses *collide*), the load cannot be executed before the store. Compilers typically capture the precedence of memory operations in a *memory dependency graph*, where nodes represent memory operations and a directed edge from v_i to v_j indicates that v_i must execute before v_j. Compiler analyses can sometimes prove certain edges unnecessary [3, 18, 21, 35]. For example, polyhedral analysis [18, 35] can determine that two memory operations never access the same location because their addresses are affine functions of loop indices with disjoint iteration spaces, making their execution order irrelevant. As a result, Figure 1b shows only nondisambiguated pairwise dependencies for the example code shown in Figure 1a. Other analyses tailored to dataflow circuits can detect cases where memory dependencies are inherently respected due to data dependencies [21]. After these analyses, all edges that are proven unnecessary can be removed, as indicated by the dashed edges in Figure 1c. At this stage, memory operations that are no longer connected in the dependency graph may be assigned to different LSQs. However, all memory operations belonging to the same connected component must still be mapped to the same LSQ. Finally, isolated nodes may be connected directly to the memory controller (MC). An LSQ tracks all pending stores and loads (often in two separate queues) and enforces all possible dependencies: *read-after-write*, *write-after-read*, and *write-after-write*. For instance, to enforce read-after-write dependencies, it compares each load address with the addresses of pending stores previously enqueued.

There is, however, an interesting point to consider. The reduction of edges in Figure 1c is leveraged to minimize both the number of ports in each LSQ and, indirectly, the number of entries in its queues. Note that while the total number of memory operations remains anyway unchanged, using multiple small LSQs instead of a single large one consistently leads to smaller and more efficient circuits. The impact is generally nonnegligible because LSQs consume a considerable fraction of resources and have a tangible influence on timing [23]. However, once connected to the memory ports, an LSQ has no knowledge of the dependency graph; it actively checks for all collisions among all accesses arriving to its ports [22], irrespective of the fact that accesses on some ports have been proven not to collide. In other words, the LSQ conservatively aims to respect the complete ordering—that is, it verifies *every* preceding access against *every* successive one—even when static analysis has determined that only a few dependencies could be violated. Figure 1d illustrates this by depicting all the dependencies the LSQ is equipped to verify: it is built to be able to compare every pending access from every memory operation with every pending access from every other. However, (i) Dataflow-tailored static analysis has proven that, due to a data dependency, every instance of load 2 is guaranteed to precede the corresponding instance of store 2 (Figure 1c); yet, the LSQ will check for possible collisions to make sure the write-after-read dependency is respected. This is neither wrong nor too conservative—it is simply unnecessary. (ii) Static analysis has also determined that a store issued by store 2 does not need to complete before load 2 can proceed, since load 2 always targets a memory location ahead of where store 2 is targeting. Consequently, this ordering never needs to be enforced (Figure 1b); yet, the LSQ has all the logic for detecting this read-after-write dependency, even if it cannot happen. Again, this is simply unnecessary logic that the LSQ possesses because it is a generic component. Admittedly, Figure 1d shows a fully connected graph and this is a worst case scenario: if there are multiple load ports, read-after-read situations never need any check, would not appear on the graph, and would not be enforced by the LSQ. Likewise, in many cases, stores can be kept in order within the LSQ simply by committing them sequentially–such edges appear on Figure 1d but may have a negligible cost, depending on the LSQ



(a) Naive approach, wrong (b) Using an LSQ, expensive (c) Sequentializing, slow (d) Our approach

Figure 2: Respecting Memory Dependencies. (a) Naive, incorrect approach: connecting dependent memory operators directly to the memory controller may execute accesses in an incorrect order. (b) Classic LSQ-based approach: routing memory accesses through an LSQ ensures correct ordering and is the traditional solution we aim to improve upon. (c) Sequentialization via completion signals: an approach is to enforce ordering by gating the successor's address with the predecessor's completion signal; however, since the number of executions of predecessors and successors may not always match, *fast token delivery (FTD)* is required to prevent deadlocks. (d) Our proposed method: Similar to (c), but selectively skips waiting when possible; to determine whether the wait can be omitted, we dynamically compare the addresses of the two memory accesses.

implementation. Nevertheless, our point is that significant circuit simplifications could be achieved by incorporating into the decision logic detailed knowledge of the exact residual dependencies that remain after static analysis. This is precisely the goal of this paper.

2.1 Out with LSQs

What are the options for enforcing the correct ordering of potentially colliding memory accesses? Consider two memory operations with a dependency edge between them in the dependency graph, $P \rightarrow S$, where P represents the predecessor operation (e.g., a store) and S the successor operation. A naive but incorrect approach would be to connect both memory operators directly to the memory controller, as shown in Fig. 2a, which fails to enforce the necessary ordering. The conventional solution, as previously discussed, is to route both operations through an LSQ that ensures correct sequencing, as illustrated in Figure 2b. As an alternative to an LSQ, a simple yet correct approach is to sequentialize memory accesses by gating the address of the successor operation with the completion signal of the predecessor. This approach leverages Join, a synchronization primitive in dataflow circuits waiting for all input values (tokens) before producing an output. By using Join, we can delay the address from reaching S until P signals its completion via a dataless token, as depicted in Figure 2c. While conceptually straightforward, this solution is problematic if implemented literally. A fundamental issue, which we will revisit in Section 5, arises from a common challenge in dataflow circuits: It is not always known in advance whether both P and S will execute. For example, if P is inside an `if` construct and does not execute, its token never arrives, causing S to stall indefinitely. To address this, the circuit labeled *fast token delivery (FTD)* in Figure 2c employs established techniques [14] to ensure progress. Although this solution now guarantees correctness, it eliminates the advantages of out-of-order execution by enforcing program order even when no memory collision occurs. Our approach, sketched in Figure 2d, avoids this limitation by only sequentializing P and S when strictly necessary. To determine whether reordering is safe, we compare the addresses of the two memory operations and, based on this comparison, we either enforce ordering or allow independent execution. This key idea will be explored in Section 4, after a brief review of dataflow circuits.

3 Dataflow Circuits

Dataflow circuits are characterized by a distributed control mechanism: Components communicate through a handshake protocol, upon the *validity* of data and the *readiness* of the receiver, without the need for a centralized controller. Data transfer is abstracted as *tokens* that can be physically implemented using two handshake signals—*valid* and *ready*.

The first step in the generation of a dataflow circuit from a software program is the translation of operations into dataflow components and the connection of components corresponding to data-dependent operations through handshake channels. Following this, the control flow of the program is implemented in the circuit using one or more *steering* components to manage the possibly conditional transfer of tokens from its sender to its recipients. These components are depicted in Figure 3. A Mux has two data inputs (T and F), a condition input (*cond*), and one data output: it sends the T-input token to the output if the condition token has value *true* and the F-input otherwise; a token on the nonselected input is left waiting. A Branch consumes a condition value and propagates a token on the data input to the T output if the condition is *true* and to the F output if the condition is *false*. A Fork has one input and multiple outputs; it propagates the input token to all of its successors. A Join, as mentioned earlier, has multiple inputs and one output; it synchronizes the tokens of all predecessors before generating the output token; the exact payload (data) of the output token depends on the context (often, only one of the input tokens

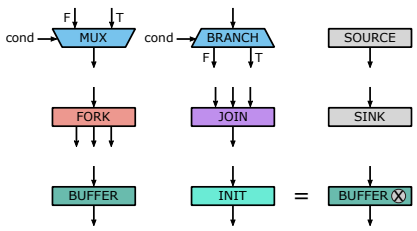


Figure 3: Dataflow Components. They *steer* data in accordance with control-flow decisions. INIT is a transparent buffer initialized with a particular token.

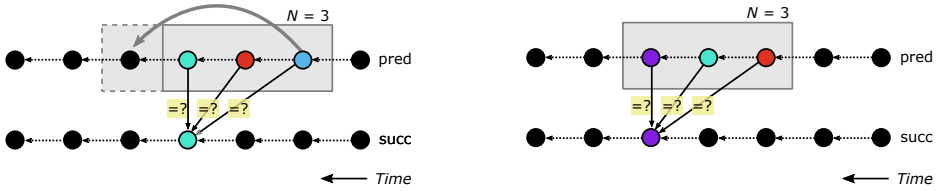


Figure 4: Predecessor Address Window. Multiple addresses of the predecessor operation are compared with the latest successor address. The window slides to the left to the next successor address only after the oldest predecessor inside it has completed.

carries data to the output). A Buffer is the only component that stores tokens without altering their value; an *opaque* version breaks combinational loops (that is, tokens must stop in it for at least a cycle, like an ordinary register). A *transparent* version, with a combinational path from input to output, is valuable to eliminate throughput bottlenecks. A Source has no inputs and constantly delivers data tokens to its successor, whereas a Sink has no outputs and is always ready to consume tokens. Additionally, we use Init: it is simply a transparent buffer initialized with a particular token before circuit operation. While this implementation of Init differs slightly and is more complex than what others have used [14], our version is more general and functional in any situation.

4 Conditionally Sequentializing

As outlined earlier, our idea is simple: We treat each edge in the memory dependency graph independently. For every edge, we gate the successor's memory access with a token that signals the completion of the predecessor. Still, we only want this gating when the two operations actually access the same location. In this section, we describe how to realize this mechanism, postponing some important details to Section 5. The procedure is as follows: (1) Compare the addresses. If they match, the predecessor's completion token triggers the successor at the appropriate time. (2) If they do not match, we (i) immediately generate an "early" token to enable the successor's execution, and (ii) later absorb the predecessor's completion token when it arrives. This behavior is implemented by the seq/skip logic shown in Figure 2d. The remaining question is this: which addresses should be compared?

A Window over the Predecessors. Consider a case where a potential dependency (e.g., a read-after-write) appears inside a loop. Our objective is to allow the load to execute as early as possible; this is feasible only if there is no collision not only with the store of the same iteration but also with all stores of previous iterations that have not yet completed. Therefore, if we want to proceed aggressively, we need to maintain a history of past instances of the store against which a new load address can be compared. In practice, we can only compare against a limited number of preceding stores and we denote this number with N; this is a design-time parameter similar to sizing the queues of an LSQ. Thus, we create a window over the store sequence, as suggested in Figure 4, and compare a new load with the last N preceding stores. The figure, generalized to arbitrary predecessor and successor operations, visualizes the succession of address tokens flowing from left to right; tokens on the right are older and dashed arrows represent the sequence in each channel. Besides implementing comparisons with the last N predecessors, we must also ensure that the $(N+1)$-th predecessor has completed, because its address is outside the window and cannot be checked. Note that since all accesses of an operation happen sequentially and tokens are never reordered in untagged dataflow circuits, the completion of the $(N+1)$-th predecessor is sufficient to ascertain the completion of all earlier ones. This additional dependency is represented by the thick gray arrow in the figure: for the execution window to advance, the predecessor access corresponding to the blue token must be completed first. Once this occurs, the purple token can be safely compared against the N tokens of the predecessor in the window, and the blue one can be excluded.

Overall Strategy. In summary, this is what we need:

(1) Compare the current successor address against the N previous predecessor addresses to determine if it is safe to execute the successor before the completion of these accesses.
(2) Let the successor proceed only if those N previous predecessors are either collision-free with it or already complete.
(3) Never execute the successor before the $(N+1)$-th previous predecessor completes, because the corresponding address has left the tracking window.

To meet these requirements, we introduce the circuitry of Figure 5 (shown for $N = 3$, with all Forks implicit for readability) between the two memory operators (Pred Mem and Succ Mem) to establish an explicit and controlled pseudodependency.

The basic idea, as suggested, is to allow the token carrying the address to reach the successor memory operator only when it is safe to execute the access. This is implemented by the bottom Join in the figure. For now, we assume that the program's control flow is such that every time the predecessor executes, the successor also executes exactly once, and vice versa; we will relax this important restriction in Section 5. We next examine each part of Figure 5 in turn to explain its functionality.

Predecessor Address Window. The circuitry at the top left and top right of Figure 5 consists of two Delay Generators, one for addresses and one for control tokens (i.e., tokens without any payload). Assuming $N = 3$, the behavior of a Delay Generator is illustrated in Figure 6a. Gray tokens represent dummy invalid tokens present at the beginning of execution in the Init components. For every token received at its input, a Delay Generator outputs once, in parallel, the last N tokens received. When connected to the address channel of the predecessor, this Delay Generator implements the address window of Figure 4 and produces the N tokens corresponding to the N predecessor addresses that the successor address must be compared with. The Init components initialize the queue as *empty* by supplying invalid elements.

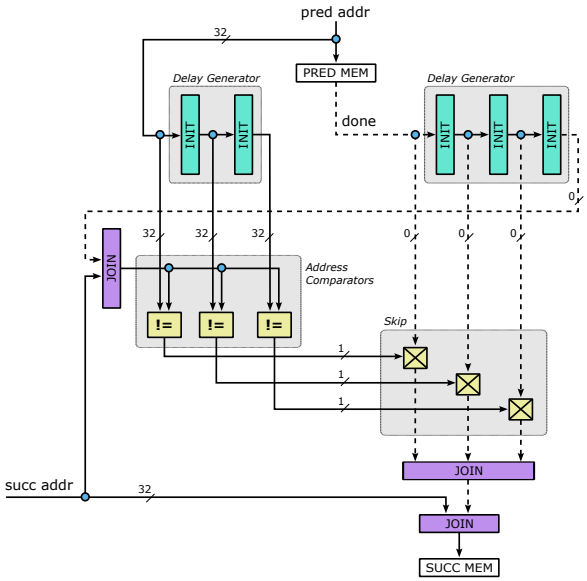


Figure 5: Circuit to Enforce a Potential Dependency. Example shown for two memory operations in the same basic block with $N = 3$. For each edge of the memory dependency graph, this circuit is inserted between the predecessor memory operation Pred Mem and the successor Succ Mem. The three subcomponents inside the Skip block are shown in Fig. 7.

Advancing the Window. As discussed above (Figure 4), the window cannot advance until the $(N + 1)$-th predecessor has completed execution. This is ensured by the Join at the successor address input to the comparators: a new set of comparisons is enabled only if the *Done* signal from the $N + 1$ predecessor (output of the Delay Generator) is received.

Comparing Addresses. The Address Comparators in Figure 5 compare each successor address with the last N addresses received by the predecessor. For now we assume that predecessor and successor execute the same number of occurrences, so the comparators receive the expected number of tokens. The outputs are binary (one payload bit plus the handshake signals), where *true* indicates that no dependency exists with the corresponding address.

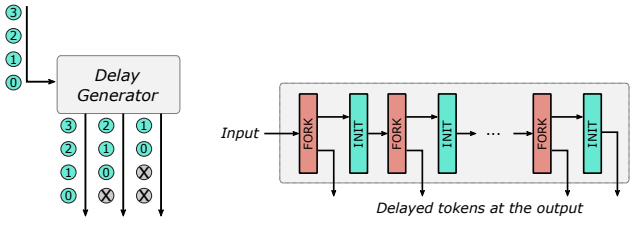
Completion Signals. The rightmost Delay Generator emits a set of control tokens (no payload, only handshake) each time Pred Mem produces a *Done* signal. The N vertical channels on the right correspond directly to the N addresses sent on the vertical channels on the left: each token indicates that the predecessor's access for the corresponding address has completed. Naturally, tokens on the rightmost channels of the set represent older predecessor accesses and can advance earlier to the Skip section, thanks to the Inits, their initial tokens, and the tokens that will later refill them. By contrast, the token on the leftmost vertical control channel can only arrive once the most recent instance of the predecessor has executed. If the execution of Pred Mem falls behind new executions of Succ Mem, the Init buffers will eventually drain and Skip will starve.

Skipping Nonexistent Dependencies. The last part of the circuit is the Skip section. If the vertical control channels from the rightmost Delay Generator were connected directly to the Join at the bottom, correctness would be guaranteed: the N joined completion signals would indicate that the last N instances of the predecessor have completed; since the Join at the top ensures that no more than N predecessor instances can still be in flight, all predecessors would be complete and it would be safe to execute the successor. However, we do not want to wait unless it is strictly necessary. This optimization is implemented by the circuit in Figure 7, where the tokens from the Address Comparators, when *true* and thus indicating lack of dependence, cause fresh tokens to be sent immediately to the Join, bypassing the completion of the corresponding noncolliding predecessor access. The eventual completion token still arrives and must be disposed of properly in a Sink through the Branch. Conversely, if token from the comparisons is *false*, no fresh token can be inserted; in this case, the actual *Done* token must be propagated to the Join and waited for, thereby truly sequentializing the accesses.

Multiple Dependency Edges. As noted at the beginning of this section, the circuit of Figure 5 enforces a single edge of the memory dependency graph and must be instantiated independently for each edge. If a memory operation has more than one incoming edge, its address must be gated with the outputs of all corresponding Joins. Conversely, if a memory operation has more than one outgoing edge, the Delay Generators of Figure 5 are shared among the circuits, since they implement the same windows over past accesses.

5 Matching Token Counts

It is worth noting that the functionality described above critically depends on a defining property of dataflow circuits: no operation is executed unless all its operands are available. The complete circuit advances in lockstep and its correct behavior may not be immediately apparent if one reasons in terms of classic digital circuits. For instance, one might think that any new successor address is compared with the predecessor addresses in the window (the top-left Delay Generator) as soon as the successor address arrives. This is not the case: if a second successor address arrives after a single predecessor address, the comparators will stall until a new set of addresses is produced by the Delay Generator, which in turn only happens when a new predecessor address arrives. This behavior is in fact correct and expected: since we have assumed that predecessor and successor execute under the same conditions (e.g., within the same loop), if a second successor address arrives, the leftmost comparator must also wait for the immediate predecessor address.



(a) Behavior **(b) Implementation**

Figure 6: Delay Generator. This circuit stores the last N tokens received and, upon each new input token, outputs the updated set of N most recent tokens.

While this reasoning assures us that the design is correct and sound when predecessor and successor share identical control-flow conditions (essentially, when they belong to the same basic block), we must still determine how to handle cases when this assumption does not hold. This section addresses the general situation.

Execution Mismatches. Consider the following situation, where predecessor and successor executions mismatch:

```
int new_val = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    array[st_index[i*n+j]] = new_val;  // store
  }
  new_val = array[ld_index[i]];        // load
}
```

Here, each execution of the load operation is preceded by multiple store accesses. In the circuit of Figure 5, this means the predecessor circuitry at the top produces one set of tokens for every iteration of the inner loop: N address tokens and N completion tokens. By contrast, the load operation at the bottom of Figure 5 consumes only one set of tokens. Dataflow circuits cannot tolerate excess tokens, nor, in a complementary example, could they consume tokens that were never produced.

The Tokens that Matter. It is easy to see what the correct solution must be in this case: all sets produced by the predecessor circuitry are irrelevant except for the last one, which is the only set the successor circuitry must consume. Note that this last set contains information about the most recent N executions of the predecessor and is emitted only after the $(N+1)$-th predecessor has completed. Thus, this final set therefore provides the complete information required to advance the load.

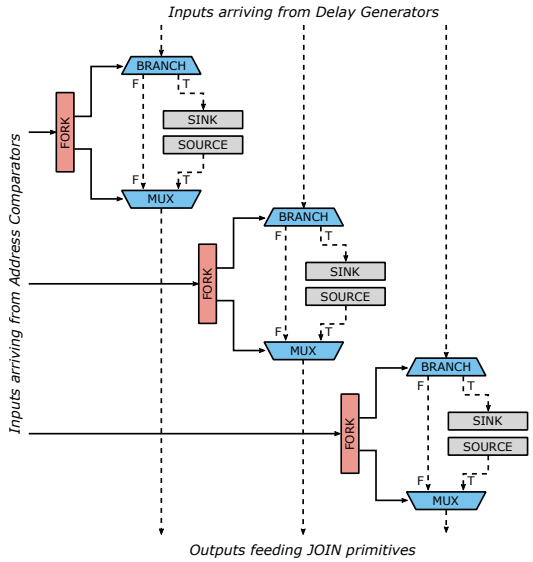


Figure 7: Skip. This circuit anticipates incoming control tokens (without payload) by emitting fresh tokens when the condition is *true*; in such cases, the corresponding tokens arriving from the top are later discarded.

A General Problem. The most general view of the problem can be illustrated by the following modification of the example:

```
int new_val = 0;
fence_state = {};
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    array[st_index[i*n+j]] = new_val;  // store
    // shift_in inserts the newest {index, done} pair and
    // drops the oldest, keeping size N
    fence_state = shift_in({ st_index[i*n+j], done });
  }
  // check load index against last N stores
  wait(fence_state, ld_index[i]);
  new_val = array[ld_index[i]];        // load
}
```

Although this pseudocode is only illustrative, it conveys the idea that each execution of the store produces a new value for `fence_state`, but only the last one is consumed by the `wait()` that gates the load. Our token-matching problem is thus no different from the general problem of propagating tokens from a producer (the top half of Figure 5) to a consumer (the bottom half of Figure 5).

A General Solution. This is a key issue in the generation of dataflow circuits from high-level code. Several approaches exist to address it, but the most efficient one we know is *fast token delivery* [14]. We therefore modify our circuit as shown in Figure 8: we mark all components above the FTD black box as belonging to the same basic block as Pred Mem (as if they assigned a value to `fence_state`) and all those below as belonging to the basic block of Succ Mem (as if they consumed the value of `fence_state` as `wait()` does), and we let a standard implementation of fast token delivery manage the connections between the producer and consumer circuitry. The standard fast token delivery algorithm takes care of detecting the execution conditions and suppressing or replicating token sets as appropriate. (It is worth noting that there is also the case where a set of tokens must be created if the predecessor has never executed before the first execution of the successor; for this we use the same solution proposed by the authors of fast token delivery in a similar situation [16, Figure 5d].) With this, our circuit to implement the conditional dependence between two arbitrary memory accesses is complete.

6 Discussion

Now that our design is complete, it is useful to reflect on its complexity, especially in comparison to a classic LSQ. The two natural complexity parameters are the number of address slots in all queues and the number of comparators.

For a typical LSQ with an s-entry store queue and an l-entry load queue, the number of address slots is simply $s + l$, while the number of comparators is typically $s \times l + n \times l$, where n is the number of stores that can be committed at once. The first set of $s \times l$ comparators checks every pending load against all preceding stores. The second set of $n \times l$ comparators checks every store candidate for execution against all preceding loads. Finally, stores are often executed strictly in order and therefore do not need to be compared among each other.

The situation is different for our design and somewhat more complex to describe, given the richer design space. For each source

node i in the memory dependency graph (Figure 1b), we require n_i address entries. If multiple edges originate from the same predecessor memory access, these address entries can be shared; thus n_i represents the maximum number of address entries attributed to that predecessor for any number of successors in the graph. Comparators, on the other hand, can never be shared: there are n_{ij} distinct comparators for every edge (i, j). In the worst case of a fully connected memory dependency graph, the number of address entries grows linearly with the number of nodes, while the number of comparators grows quadratically. Although this trend is reminiscent of LSQs (but not directly comparable), it is less favorable for our design: In a classic LSQ all loads share a single queue, whereas in our approach every load has its own private queue. Thus, for similar performance, the total cost of our private queues may exceed that of the single shared load queue, where the overabundance of some accesses can be offset by the rarity of others.

We believe, however, that memory dependency graphs are usually either fairly small (2–3 nodes) or quite sparse. In such cases, our design has a clear advantage. Consider the classic histogram application, where the loop contains two accesses (a load and a store) with a single potential dependency to relax: only the read-after-write needs to be checked, as the write-after-read is already enforced by a true data dependency. A typical LSQ with n slots in both queues and committing one store at a time would need $2n$ addresses in its queues and perform $n(n+1)$ comparisons. Our equivalent design requires only n slots in the store queue and n

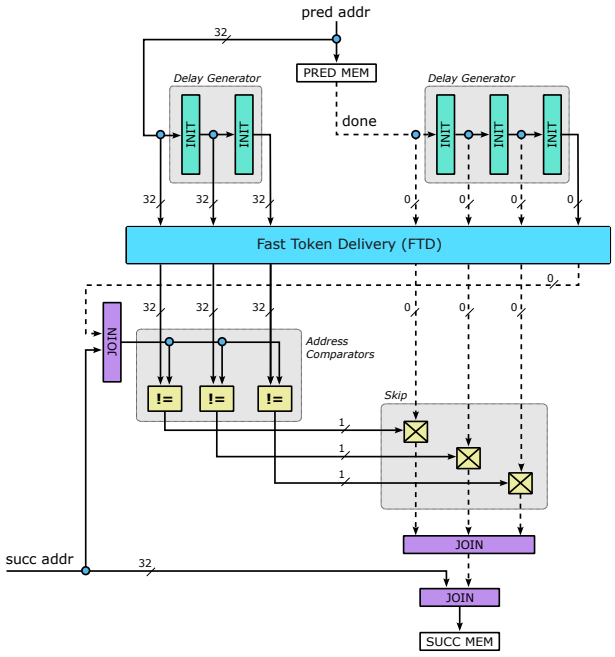


Figure 8: General Circuit to Enforce a Potential Dependency. Example shown for $N = 3$. This circuit differs from that of Figure 5 in only one aspect: the tokens produced by the Pred Mem circuitry are delivered to the Succ Mem circuitry through a suppression and replication network implemented using *fast token delivery* [14].

comparators. Qualitatively, despite the structural differences, both approaches check the same actual dependencies.

Finally, one could note that our design lacks, compared to the classic behavior of an LSQ, a store-to-load forwarding path. While not conceptually difficult to implement (one could imagine a similar architecture to that of Figure 8), we chose instead to develop a single circuit that conditionally enforces all types of dependencies. Of course, store-to-load forwarding circuitry could be added, but our results suggest that it is not particularly important in practice.

7 Experimental Setup

We implement our approach as a compiler pass within *Dynamatic* [1], an open-source dynamically scheduled HLS tool built on MLIR [31]. Our implementation is publicly available [34], and we intend to integrate it into Dynamatic. The memory dependency graph (Figure 1c) is constructed using Dynamatic's built-in memory analysis [21], while its unmodified backend handles RTL generation. Since Dynamatic also provides an implementation of *fast token delivery* [14], we reuse it for the purpose outlined in Section 5. Circuit correctness and cycle counts are verified by simulating all designs with ModelSim 2020.1 [28]. Finally, the designs are synthesized, placed, and routed using Vivado 2024.2 [2], targeting a Kintex-7 AMD FPGA.

Buffers. Buffers play a crucial role in dataflow circuits by (1) ensuring correct operation through opaque buffers that break combinational cycles and (2) maximizing throughput through transparent buffers that prevent token stalls. Dynamatic's buffer placement strategy is based on a mixed-integer linear programming (MILP) model [25], which enforces path constraints to meet a target clock period and throughput constraints to avoid stalls. While the path constraints remain applicable in our setting, the throughput constraints cannot be directly used, as they assume a strict control-flow structure that does not always hold under *fast token delivery*. Moreover, parts of our circuits differ significantly from those generated by conventional C-to-hardware translation, further limiting the applicability of the full model. Consequently, we retain only the path-constraint subset of the MILP formulation and adopt a simple empirical heuristic for throughput: We initially place transparent buffers of size fifteen at each Fork output and then iteratively remove those whose absence does not degrade performance. To accelerate this process, we use binary partitioning: we attempt to remove all buffers at once and, if performance degrades, recursively apply the procedure to two subsets. Once pruning completes, we resize the remaining buffers using the same recursive strategy, first to size one and then incrementally to larger sizes until no further resizing is possible.

Execution Time and Resources. Our objective is to minimize wall-clock execution time. Although our approach supports assigning different N values to edges in the dependency graph, we restrict exploration to a single global N per benchmark to simplify the search space. For each technique (standard Dynamatic and ours) and design point (LSQ size or N), we proceed as follows. We first empirically insert transparent buffers as described earlier. We then sweep the clock period (CP) constraint provided to Dynamatic's MILP solver, generating multiple designs that are placed and routed using Vivado. For each design, place-and-route starts with a CP slightly smaller than the target CP; if timing is met, we record the

Table 1: Number of Memory Accesses and Edges in the Dependency Graph per Benchmark.

	bicg	gaussian	matrix power	get tanh	histogram	atax	jacobi 1d	triangular	mvt float	kernel 2mm	kernel 3mm
Memory accesses	4	3	3	2	2	4	6	2	4	8	11
Edges	2	2	2	1	1	2	8	1	2	9	13

achieved CP, otherwise we increase the CP based on the observed negative slack and repeat until nonnegative slack is achieved. We measure the cycle count of each resulting design and compute execution time as the product of the effective CP and the cycle count. Finally, we select the design with the minimum execution time and report its corresponding place-and-route resource utilization.

Baseline. As described, we apply this methodology to both *Dynamatic*'s LSQ-based circuits and ours. We observed that a buffering strategy that combines the MILP model for path constraints with empirical throughput adjustments often yields better results than using the full MILP model alone. To ensure a fair comparison, we apply the same buffering technique to *Dynamatic*'s LSQ-based circuits as well, which we use as our baseline. In both cases, the key design parameters (LSQ size and N, respectively) determine a trade-off between execution time and resource usage. We sweep these parameters to compare the Pareto fronts of both techniques and report the configurations that deliver near-optimal performance with minimal resource usage. Where applicable, we rely on prior work to size the LSQ, covering the entire design space [27]. In our approach, setting $N = 0$ enforces unconditional sequentialization (Figure 2c). We report results for eleven *Dynamatic* kernels adapted from the PolyBench suite [36]. These benchmarks represent various dependency graphs and requirements for matching token counts. The numbers of memory accesses (i.e., vertices) and edges in the dependency graph for each benchmark are shown in Table 1.

8 Experimental Results

Table 2 summarizes the comparison of our method, sized with the best N, with the standard LSQ, sized with the optimal parameters where available [27]. The second column lists these parameters as an ordered pair, where the first value denotes the load queue size and the second denotes the store queue size. The same results are also visualized in Figure 9, plotted relative to the LSQ version (bold triangle). The dataset represented with circles in the figure refers to the configuration with the best N while the dataset represented with squares shows our results with $N = 0$, where accesses that have a potential dependency are simply sequentialized. The colors identify the benchmark. The larger data points in black show the geometric means of the two datasets.

One can observe that our solution for the optimal N is always Pareto-optimal with respect to the LSQ-based solutions. In one case, it incurs a higher cost to achieve a larger performance gain; however, in most cases, it achieves comparable performance at a lower resource cost. Despite the spread of results and the variety of benchmark behaviors, the geometric mean clearly captures the overall trend. In general, LSQs and our approach achieve similar timing results, which is unsurprising given that they enforce the same dependency constraints. Nevertheless, our technique may exhibit small advantages or disadvantages in specific cases, primarily due to differences in the resulting critical-path. In contrast, our approach typically requires fewer resources. This outcome is the result of two opposing effects: On the one hand, our implementation is a relatively inefficient design made of dataflow components exchanging handshake signals, whereas the reference LSQ is a highly optimized, state-of-the-art digital design. On the other hand, as argued earlier in the paper in Section 6, our approach implements only the strictly necessary subset of dependency checks. In practice, the latter effect clearly dominates.

Unsurprisingly, the simplest approach, blindly sequentializing all dependencies ($N = 0$), yields the lowest execution cost in terms of resources but also results in the worst execution time. However, except for specific cases such as the *Histogram* benchmark, where an LSQ is essential, the significantly shorter critical path of the fully sequentialized design prevents performance from degrading catastrophically in most benchmarks. This observation indicates that, while an LSQ is often required for correctness (and is therefore inserted by Dynamatic), its performance benefits can be marginal at times; in some cases, the negative impact on the critical path outweighs the reduction in execution cycles it provides. Although our approach does not introduce fundamentally new insights in this regard, it exposes an additional, very low-cost design point in such scenarios.

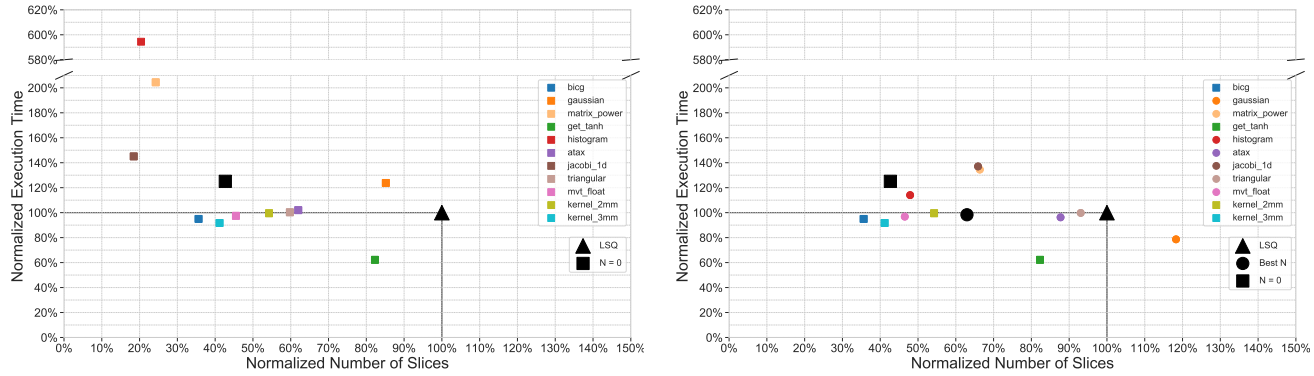
Figure 10 shows more detailed results for the *Histogram* benchmark (other benchmarks follow a similar pattern). Figure 10a shows the execution time and the number of slices for different LSQ sizes and N values, and demonstrates that our approach consistently achieves Pareto-dominant solutions across different LSQ sizes. Figure 10b further confirms that both LSQ-based designs and ours exploit the same access reordering opportunities, leading to similar clock cycle counts. It is important to emphasize that, as shown in Figure 10a, beyond a certain point, there are no further reordering opportunities. Increasing the sizing parameters past this point only increases the critical path without reducing the cycle count, leading to an increase in overall execution time. Finally, Figure 10c shows that for $N \leq 3$, when the critical path is dominated by other parts of the design, our approach achieves a shorter critical path than the LSQ-based implementation. However, beyond this point, once the critical path becomes dominated by the circuitry introduced by our approach, the critical path increases—a drawback of the dataflow-based implementation. A similar pattern appears for individual resources, LUTs and FFs, in Figures 10d–10i.

9 Related Work

Supporting memory disordering in spatial circuits is challenging. A few authors have generated dataflow circuits from imperative programs without supporting memory disordering [7, 19, 26]. Elakhras et al. [15] recently enabled fine-grained tagging in dataflow circuits to tolerate out-of-order execution but they still conservatively reorder in the presence of potential memory hazards. Others [8, 16, 23], as suggested throughout this work, interfaced dataflow circuits with the LSQ design adapted for spatial circuits [22]. The key challenge was the generation of the allocation network that communicates the program order of memory accesses to the LSQs: Early work built a network strictly mimicking

Table 2: Our Results Compared to the Results with an LSQ.

			Clock Cycles			CP (ns)			Execution time (µs)				Slices				LUTs				FFs			
Benchmark	LSQ	Best N	Best	N=0	LSQ	Best	N=0	LSQ	Best	N=0	LSQ	Best LSQ	Best	N=0	LSQ	Best LSQ	Best	N =0	LSQ	Best LSQ	Best	N=0	LSQ	Best LSQ
bicg	(5, 6)	0	2024	2024	1877	3.70	3.70	4.20	7.49	7.49	7.88	36%	334	334	938	95%	881	881	2949	30%	892	892	1795	50%
gaussian	(4, 2)	1	12336	20528	17476	3.91	3.70	3.51	48.23	75.95	61.34	79%	550	396	465	118%	1744	1198	1345	130%	1256	909	1076	117%
matrix_power	(8, 11)	4	1057	2670	797	4.97	2.99	4.90	5.25	7.98	3.91	135%	700	256	1055	66%	2182	688	3564	61%	1838	579	1438	128%
get_tanh	(2, 5)	0	3197	3197	5010	6.63	6.63	6.80	21.20	21.20	34.07	62%	859	859	1044	82%	2542	2542	3352	76%	1399	1399	1814	77%
histogram	(7, 17)	7	1054	8011	1058	7.67	5.26	6.70	8.08	42.14	7.09	114%	700	298	1461	48%	2229	861	4909	45%	1622	701	1955	83%
atax	(6, 6)	3	3265	3353	3320	6.92	7.14	7.07	22.59	23.94	23.47	96%	1415	1000	1613	88%	3902	2749	5048	77%	2879	2021	2737	105%
jacobi_1d	(8, 8)	2	1345	2359	1182	6.13	3.70	5.09	8.24	8.73	6.02	137%	1068	299	1621	66%	3159	878	5432	58%	2368	676	1982	119%
triangular	(6, 6)	3	130	269	176	7.70	3.74	5.70	1.00	1.00	1.00	100%	791	508	850	93%	2338	1447	2557	91%	1745	1311	1540	113%
mvt_float	(8, 8)	1	10444	10471	10446	7.02	7.05	7.25	73.32	73.82	75.73	97%	1016	994	2185	46%	2889	2775	6843	42%	1982	1925	3339	59%
kernel_2mm	(8, 8)	0	4613	4613	4437	4.70	4.70	4.91	21.68	21.68	21.79	100%	1146	1146	2112	54%	3345	3345	7069	47%	2668	2668	3223	83%
kernel_3mm	(8, 8)	0	6463	6463	5809	4.70	4.70	5.70	30.38	30.38	33.11	92%	1120	1120	2721	41%	3394	3394	8939	38%	2439	2439	3661	67%
GeoMean	-	-	2408	3720	2518	5.64	4.65	5.49	13.60	17.30	13.83	**98%**	825	559	1311	**63%**	2436	1599	4193	**58%**	1818	1236	2085	**87%**

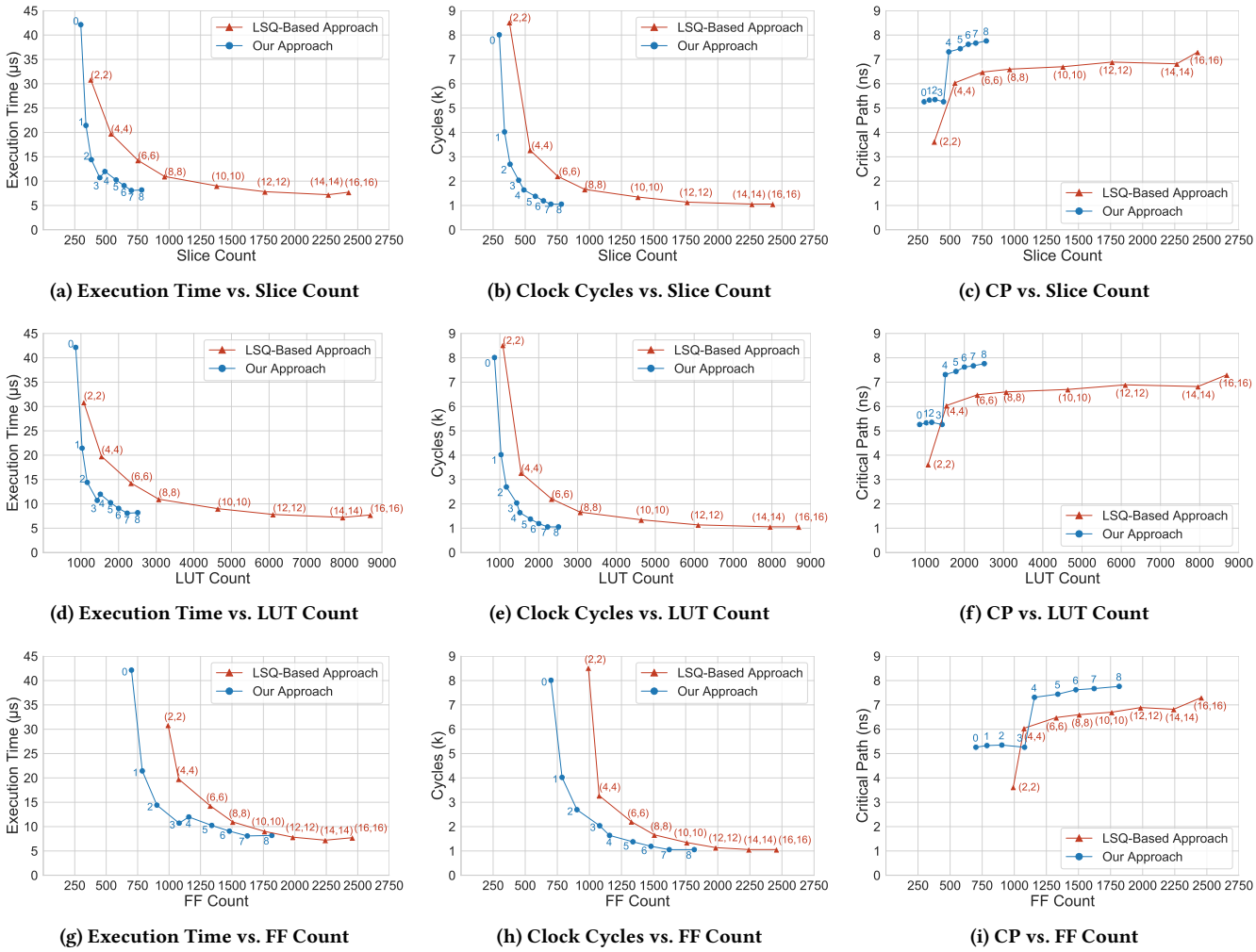


(a) N = 0 **(b) Best N**

Figure 9: Comparison with LSQ. The graphs plot execution time versus resources for all benchmarks, relative to the LSQ solution (where available, with optimal sizing [27]). The data marked with circles uses the best value for N, and the data marked with squares corresponds to $N = 0$ (i.e., simple sequentialization). When the best N is 0, a square is used. The larger black points are the geometric means of each data series. Our best-N solution is always Pareto-optimal with respect to LSQ solutions.

the sequential control flow [23]. More recent work has adopted static analysis [8] or control dependency analysis to simplify this network [16]. These authors use costly general-purpose LSQs for performance, whereas our approach crafts circuits that are optimized to particular dependencies, promising the same performance at a cheaper cost. Gorius et al. [17] introduced memory dependency speculation in statically scheduled HLS-generated circuits; their analysis could help extend dataflow circuits to memory speculation.

A recent paper [29] is probably the most similar to our work and needs thorough analysis. While extending an intermediate representation to support the generation of dataflow circuits, the authors pursue essentially the same goal as we do in their context: building a network that conditionally delays potentially dependent memory operations in the presence of aliases. The paper, however, provides few details on the implementation and some explanations remain difficult to interpret unambiguously. Essentially, the functionality of their ADDR-Q in their Figure 5b corresponds approximately to our circuit in Figure 5. A key difference, though, is that ports enq and deq of ADDR-Q operate in a logically independent manner from ports check and addr; in other words, at any point in time, ADDR-Q can check a new successor address against the current state of the queue. While this seems a natural functionality to implement, in a dataflow circuit this leaves open the problem whether all instances of the predecessors preceding the successor in program order have been enqueued (ports enq and deq) before proceeding with the disambiguation check. For this, the authors insert a network of state gates (SGs in their Figure 5b) that conservatively propagate the program-order token through the red state edge; their Algorithm guarantees correctness but enforces strict sequentialization in complex control-flow cases where finer-grain overlap would be possible. In contrast, our circuit in Figure 5 internally coordinates all functions proper of ADDR-Q: for any predecessor execution, it expects *exactly* a done signal and a *single* successor address for a set of comparisons. If there is any count mismatch between the predecessor tokens (address and *Done*) and the successor token (address), it is left to fast token delivery [14] to take care of any necessary compensation—in other words, we simply rely on a general solution to handle any possible control-flow situation. We also notice that, whereas we relax all types of potential dependencies, the authors conservatively enforce write-after-read in program order (SG4 in their Figure 5b) and do not relax write-after-write either. Given this, in the critical case of nondisambiguable addresses in a loop



(a) Execution Time vs. Slice Count (b) Clock Cycles vs. Slice Count (c) CP vs. Slice Count

(d) Execution Time vs. LUT Count (e) Clock Cycles vs. LUT Count (f) CP vs. LUT Count

(g) Execution Time vs. FF Count (h) Clock Cycles vs. FF Count (i) CP vs. FF Count

Figure 10: Detailed Timing and Area Results of Our Approach Compared to the LSQ-Based Approach for *Histogram*.

(e.g., histogram), it seems that their ADDR-Q appears unable, in practice, to receive more than one element and would degenerate into a single-entry buffer; their solution would then essentially correspond to our queue of size one. We thus consider our solution to be more complete and more general.

Prior work on running software on dataflow architectures [4, 32] has struggled to ensure the correct execution of memory operations. Some works supported only memory operations with constrained semantics and provided tailored structures for them, such as write-once I-structures [5] and mutable M-structures [6]. Others proposed an instruction-set architecture that encodes memory dependencies within instructions, thereby adding overhead [37]. In contrast, we, like our immediate predecessors, enable safe memory reordering under the generic memory semantics of imperative languages.

10 Conclusion

Reordering memory operations during execution based on actual access addresses is a key advantage of dynamically scheduled computing systems. Prior work on dynamically scheduled HLS has followed the strategy of modern processors, relying on highly optimized load-store queues with interfaces adapted to elastic circuits. While this leverages a well-understood and efficient paradigm, it imposes a generic, application-agnostic structure and demands non-scalable components that may not be available in all contexts [12]. In this paper, we have shown that an HLS compiler can instead generate application-specific elastic circuitry, sized as needed, that achieves the same conditional reordering effect. Although our circuits rely on less efficient components with pervasive handshake signalling, they achieve execution cycles comparable to those of LSQs, as they make the same decisions at similar points in the execution. At the same time, the simpler logic they implement results in a tangible reduction in resources (37% fewer slices on average) for a slightly improved execution time (by 2% on average). We implement our approach as a compiler pass in the open-source *Dynamatic* toolchain [1] and release it publicly [34]. We hope this will encourage further exploration of dataflow circuits as an efficient design paradigm, particularly within HLS tools.

Acknowledgments

Ayatallah Elakhras is supported by a Google PhD Fellowship in Distributed Systems and Parallel Computing.

References

[1] 2024. *Dynamatic v2.0.0.* https://dynamatic.epfl.ch/

[2] AMD 2024. *Vivado Design Suite.* AMD. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2024-2.html Version 2024.2.

[3] J. R. Appel and K. Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach* (first ed.). Morgan Kaufmann.

[4] Arvind and Rishiyur S. Nikhil. 1990. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers* C-39, 3 (March 1990), 300–18. doi:10.1109/12.48862

[5] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems* 11, 4 (Oct. 1989), 598–632. doi:10.1145/69558.69562

[6] Paul S. Barth and Rishiyur S. Nikhil. 1991. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*. Springer, Berlin, 538–568.

[7] 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. doi:10.1109/ISPASS.2005.1430572

[8] Jianyi Cheng, Lana Josipović, George A. Constantinides, and John Wickerson. 2022. Dynamic Inter-Block Scheduling for HLS. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 243–52. doi:10.1109/FPL57034.2022.00045

[9] George Z. Chrysos and Joel S. Emer. 1998. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. Barcelona, 142–53.

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

[11] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. Synthesis of Synchronous Elastic Architectures. In *Proceedings of the 43rd Design Automation Conference*. San Francisco, Calif., 657–62. doi:10.1145/1146909.1147077

[12] Louis Coulon, Lucas Ramirez, Jason Anderson, Mirjana Stojilović, and Paolo Ienne. 2025. FRIDA: Reconfigurable Arrays for Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 33rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 147–58. doi:10.1145/3706628.3708880

[13] Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. 2019. Compositional Dataflow Circuits. *ACM Transactions on Embedded Computing Systems* 18, 1 (Jan. 2019), 5:1–5:27.

[14] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2022. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. In *Proceedings of the 30th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Belfast, UK, 253–61. doi:10.1109/FPL57034.2022.00046

[15] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, 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, Calif., 44–54. doi:10.1145/3626202.3637556

[16] 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, Calif., 39–45. doi:10.1145/3543622.3573050

[17] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. 2024. A Unified Memory Dependency Framework for Speculative High-Level Synthesis. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. Edinburgh, 13–25. doi:10.1145/3640537.3641581

[18] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly - Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*. Chamonix, 1–6.

[19] Hagen Gädke-Lütjens. 2011. *Dynamic Scheduling in High-Level Compilation for Adaptive Computers*. Ph.D. Thesis. Technischen Universität Braunschweig, Braunschweig, Germany. doi:10.24355/dbbs.084-201105300920-0

[20] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. 2000. Introducing the IA-64 Architecture. *IEEE Micro* 20, 5 (Sept.–Oct. 2000), 12–23.

[21] Lana Josipović, Atri Bhattacharyya, Andrea Guerrieri, and Paolo Ienne. 2019. Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*. Tianjin, 197–205.

[22] 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), 125:1–125:19. doi:10.1145/3126525

[23] 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, Calif., 127–36. doi:10.1145/3174243.3174264

[24] 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* CAD-41, 7 (July 2022), 2142–55. doi:10.1109/TCAD.2021.3105574

[25] 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, Calif., 186–96. doi:10.1145/3373087.3375314

[26] Nico Kasprzyk. 2005. *COMRADE: Ein Hochsprachen-Compiler für Adaptive Computersysteme*. Ph.D. Thesis. Technischen Universität Braunschweig, Braunschweig, Germany.

[27] Jiantao Liu, Carmine Rizzi, and Lana Josipović. 2022. Load-Store Queue Sizing for Efficient Dataflow Circuits. In *Proceedings of the IEEE International Conference on Field Programmable Technology*. Hong Kong SAR, 1–9. doi:10.1109/ICFPT56656.2022.9974425

[28] Mentor Graphics 2020. *ModelSim.* Mentor Graphics. https://www.intel.com/content/www/us/en/software-kit/750666/modelsim-intel-fpgas-standard-edition-software-version-20-1-1.html

[29] David Metz, Nico Reissmann, and Magnus Sjalandar. 2024. R-HLS: An IR for Dynamic High-Level Synthesis and Memory Disambiguation based on Regions and State Edges. In *Proceedings of the 43rd International Conference on Computer-Aided Design*. Newark, N.Y., 1–9. doi:10.1145/3676536.3676671

[30] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. 1997. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colo.). 181–93.

[31] Multi-Level IR Compiler Framework 2020. *https://mlir.llvm.org/.* Multi-Level IR Compiler Framework. https://mlir.llvm.org/

[32] Gregory Michael Papadopoulos. 1998. *Implementation of a General-Purpose Dataflow Multiprocessor*. Ph.D. Thesis. Massachusetts Institute of Technology, Laboratory for Computer Science. http://hdl.handle.net/1721.1/27967

[33] Il Park, Chong Liang Ooi, and T.N. Vijaykumar. 2003. Reducing Design Complexity of the Load/Store Queue. In *Proceedings of the 36th International Symposium on Microarchitecture*. San Diego, Calif., 411–22.

[34] Rouzbeh Pirayadi, Ayatallah Elakhras, Mirjana Stojilović, and Paolo Ienne. 2025. *Out with LSQs: Custom Circuits for Memory Access Reordering in Dynamic HLS: Artifacts*. doi:10.5281/zenodo.17975149

[35] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 29–38.

[36] Louis-Noël Pouchet. 2012. *Polybench: The Polyhedral Benchmark Suite.* https://sourceforge.net/p/polybench/wiki/Home/

[37] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The WaveScalar Architecture. *ACM Transactions on Computing Systems (TCS)* 25, 2, Article 4 (May 2007), 54 pages. doi:10.1145/1233307.1233308

[38] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. A High-Frequency Load-Store Queue with Speculative Allocations for High-Level Synthesis. In *Proceedings of the 33rd IEEE International Conference on Field Programmable Technology*. IEEE, Yokohama, Japan, 115–124. doi:10.1109/ICFPT59805.2023.00018