*To appear in the International Conference on Field Programmable Technology (FPT'23), Yokohama, Japan, 11-14 December 2023.*

# A High-Frequency Load-Store Queue with Speculative Allocations for High-Level Synthesis

Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede
School of Computing Science
University of Glasgow, UK
Email: {robert.szafarczyk, syed.nabi, wim.vanderbauwhede}@glasgow.ac.uk

***Abstract*—Dynamically scheduled high-level synthesis (HLS) enables the use of load-store queues (LSQs) which can disambiguate data hazards at circuit runtime, increasing throughput in codes with unpredictable memory accesses. However, the increased throughput comes at the price of lower clock frequency and higher resource usage compared to statically scheduled circuits without LSQs. The lower frequency often nullifies any throughput improvements over static scheduling, while the resource usage becomes prohibitively expensive with large queue sizes. This paper presents a method for achieving dynamically scheduled memory operations in HLS without significant clock period and resource usage increase. We present a novel LSQ based on shift-registers enabled by the opportunity to specialize queue sizes to a target code in HLS. We show a method to speculatively allocate addresses to our LSQ, significantly increasing pipeline parallelism in codes that could not benefit from an LSQ before. In stark contrast to traditional load value speculation, we do not require pipeline replays and have no overhead on misspeculation. On a set of benchmarks with data hazards, our approach achieves an average speedup of 11× against static HLS and 5× against dynamic HLS that uses a state of the art LSQ from previous work. Our LSQ also uses several times fewer resources, scaling to queues with hundreds of entries, and supports both on-chip and off-chip memory.**

***Index Terms*—high-level synthesis, load-store queue, compiler speculation, dynamic scheduling**

## I. Introduction

High-level synthesis (HLS) tools transform high-level software code into a custom architecture that can be synthesized on an FPGA. Such architectures have the potential to achieve higher performance and energy efficiency than general-purpose CPUs and GPUs [1]. A major obstacle to the wider adoption of FPGA acceleration remains their programmability. HLS tools have lowered the barrier of entry for FPGA programmers dramatically when compared to using hardware description languages, but they still impose a specific structure on the input code, which is not intuitive to software programmers. Our goal is to increase the quality of HLS by shifting the burden of structuring code for a spatial architecture from the programmer to the compiler.

Loop pipelining is a critical step in HLS. It is the process of starting new loop iterations while previous iterations have not yet finished, allowing to achieve higher throughput with the same amount of compute resources. The number of cycles between the start of two subsequent iterations is called the Initiation Interval (II). A loop with a constant II, $N$ iterations, and a latency of $L$ will execute in $L + (N-1) \times II$ cycles, which for $N \gg L$ can be approximated as $N \times II$. Thus, a low loop II is crucial to achieving good performance in HLS.

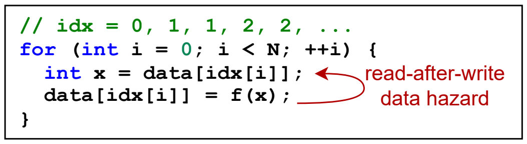
Most HLS tools use modulo scheduling to perform loop pipelining [2]–[4] (such tools are often called static HLS). Modulo scheduling maps operations for a single loop iteration to discrete clock cycles at compile time and then repeats this schedule for all loop iterations. One of the first steps in modulo scheduling is determining the minimum number of cycles between the start of subsequent loop iterations, while honoring data dependencies across iterations. Such data dependencies form recurrences in the Data Dependence Graph (DDG) of the input code. Modulo scheduling finds the maximum recurrence-constrained II across all recurrences for a given loop:

$$recII = max_i \lceil delay_i / distance_i \rceil,$$

where $delay$ is the number of cycles needed to traverse the recurrence path, and $distance$ is the number of iterations between the definition of a recurrence value and its use.

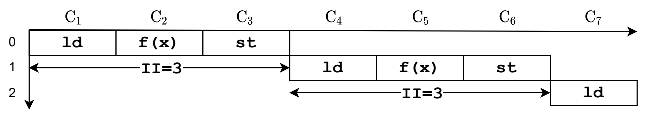
Static HLS tools rely on an accurate memory dependency analysis to discover the dependence $distance$ of a DDG recurrence through memory. Memory dependency analysis from software compilers, such as the polyhedral model, are directly applicable in this case [5]–[7]. However, there is a large class of codes where the calculation of the dependence distance is fundamentally impossible due to limited compile time information. Take the code in fig. 1 as an example. The code contains data-dependent memory reads and writes that form a recurrence in the DDG. For such codes, the dependence distance cannot be obtained and has to be conservatively set to one, i.e. it is assumed that every iteration needs to wait for all previous iterations to finish, eliminating any possibility for loop pipelining as seen in fig. 1b.

An alternative approach to achieving loop pipelining is to use dynamic scheduling. Dynamic HLS uses dataflow scheduling to trigger the execution of operations at runtime based on the availability of data, rather than a static compile-time schedule [8], [9]. Dynamic scheduling enables the use of load-store queue (LSQ) structures that allow for dynamically scheduled out-of-order loads that are essential for pipelining codes with unpredictable memory accesses [10]. For our example code from fig. 1a, dynamic HLS with an LSQ can achieve the ideal schedule in fig. 1c. However, dynamic HLS incurs non-trivial resource and critical path overheads, often nullifying any throughout advantage over static HLS [8], [11].

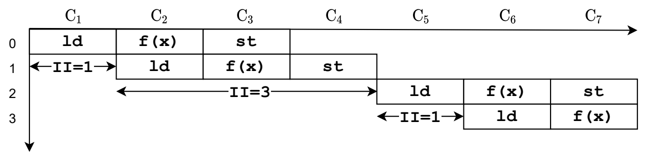


```
// idx = 0, 1, 1, 2, 2, ...
for (int i = 0; i < N; ++i) {
  int x = data[idx[i]];
  data[idx[i]] = f(x);
}
```

(a) Motivating source code with a data hazard.



(b) A static schedule: a new iteration started every 3 cycles for all iterations.



(c) An ideal schedule: a new iteration started every 1.5 cycles on average.

Fig. 1. A motivating example of code with a data hazard. Current static HLS tools need to create a worst case schedule at compile time (b). HLS with dynamically scheduled memory operations can achieve the schedule in (c).

Recent work has shown the possibility of combining static and dynamic scheduling to achieve the high throughput of dynamic scheduling with the low critical path of static HLS [12], [13]. However, whenever an LSQ is needed by the dynamic part of such a combined circuit, the high critical path and area overheads return.

Thus, to unleash the full potential of circuits combining dynamic and static scheduling, there is a clear need for an LSQ with a faster critical path, lower area overhead, and better scalability than previous work. We make the following contributions toward this goal:

- A novel load-store queue (LSQ) design for HLS with shift-register based queues enabled by the opportunity to specialize queue sizes to a target code in HLS (sec. IV). We show how the decoupled access/execute architecture model can be applied to address generation to enable the use of an LSQ in static HLS (sec. V).
- An extension to our LSQ and compiler that enables *speculative address allocations* – a compiler speculation algorithm applied to LSQ address allocations in HLS that doesn't require replays and has no misspeculation penalty (sec. V-C and V-D).
- An evaluation of our work against static HLS (Vivado HLS, Intel HLS), and against dynamic HLS. We show that our work achieves both better speedup and lower area overhead compared to a dynamic HLS compiler with a state of the art LSQ [8]. We demonstrate that our LSQ supports larger queues (tab. II), and that our speculative LSQ address allocations enable to accelerate a broader range of codes (sec. VI). We show that our LSQ can be used equally well to protect on-chip and off-chip memory.

## II. Background & Related Work

### A. Dynamically Scheduled High-Level Synthesis

Dynamically scheduled circuits rely on the theory of latency-insensitive design formalized by Carloni *et al.* [14] and simplified by Cortadella *et al.* for synchronous circuits [15]. In latency-insensitive designs, the communication between modules is decoupled from their cycle behavior, allowing for dataflow scheduling of compute circuits [8], [9], [16]–[19]. By using more resources to defer scheduling to runtime, dataflow circuits can achieve perfect throughput on codes with unpredictable inter-iteration dependencies. The disadvantage of dataflow circuits mapped to FPGA technology is firstly their significantly higher critical path, and secondly their higher area usage. The higher area usage is often acceptable, but a higher critical path means that the final design synthesized on FPGA hardware is not able to achieve the frequencies achievable by static HLS. The critical path increase is due to using LSQs, and due to using buffers with a zero cycle write-to-read latency (called transparent buffers in Dynamatic [8]) where static HLS can use a wire with a finite state machine controller.

### B. Combining Static and Dynamic Scheduling

Cheng *et al.* extended Dynamatic with the DASS methodology (Dynamic and Static Scheduling) [12], [20], which identifies static islands in an otherwise dynamically scheduled circuit. This improves the resource usage of the final circuits but the critical path stays often the same.

Szafarczyk *et al.* extended modulo-scheduled HLS tools with support for selective dynamic scheduling by breaking up the DDG of an input code into multiple modulo-scheduling instances based on compiler analysis that determines where dynamic scheduling is beneficial [13]. The separate modulo scheduling instances communicate via latency-insensitive channels – a construct available in most static HLS tools. Their approach achieves virtually the same frequency as static HLS on codes that don't require an LSQ. If an LSQ is required, the frequency of their approach matches that of Dynamatic. Since most codes amenable to dynamic scheduling do have unpredictable memory accesses that do require an LSQ, their approach is of limited value without an LSQ that can provide a low critical path. In this work, we combine their scheduling methodology with a novel LSQ design that is able to achieve such low critical paths.

### C. Runtime Memory Disambiguation in HLS

To avoid pipeline stalls due to unpredictable memory accesses, a circuit can use additional logic, such as load-store queues (LSQs), to handle memory accesses at runtime [21]. If proven safe to do so, the logic should allow loads from later loop iterations to be executed without waiting for stores from earlier iterations to commit. There are two main approaches to enable such out-of-order loads: address-based approaches compare addresses of loads and stores; value-based approaches speculatively execute loads and replay the datapath on misspeculation.

*Value-based disambiguation:* Thielmann *et al.* investigated the use of load speculation in reconfigurable hardware [22]. In their framework, if a speculated load value turned out to be incorrect, then only the computation depending on the load had to be repeated, not the whole pipeline. Nonetheless, codes with loop-carried dependencies, which are the focus of our work, had a high misprediction penalty that was a problem. Dai *et al.* [23] also used value speculation to enable pipelining of loops with irregular memory accesses. They proposed a source-to-source transformation that replaces hazardous accesses with virtualized accesses to an independent array. These independent array accesses are then handled by a custom Hazard Resolution Unit which speculatively executes loads, performs store-load forwarding, and sends misprediction signals to the datapath. Misprediction triggers a squash and replay action, which adds overhead. The benefit of value-based disambiguation is that it can pipeline loops where the store operation is control-dependent on a load [22]. The disadvantage is that squash and replay is expensive. We use an address-based approach to avoid costly replays needed for misspeculated values. However, we use the idea of speculation to speculatively produce address allocations for our LSQ in codes where the store operation is control-dependent on a load. Our compiler speculation approach doesn't require replays, allowing us to support the same codes as value-based disambiguation approaches without their overhead.

*Address-based memory disambiguation* compares the addresses of loads and stores out-of-order with the actual load/store operations, allowing non-conflicting loads to execute even if earlier stores have not yet committed. Such functionality is most often implemented as an LSQ. Most LSQs aimed at HLS have a similar operating principle as LSQs used in out-of-order CPUs [21]. For example, the Dynamatic LSQ [10] has a single store queue buffer which holds stores in-flight to memory, together with metadata needed to recover program order. Dependent loads check this structure for aliasing using the memory address and other metadata, deciding if a load is safe to perform, if a store value can be forwarded, or if the load has to wait. It is this single-cycle Content-Addressable Memory (CAM) access that maps poorly to FPGA technology, resulting in a high critical path and area usage [24].

Our LSQ design is fundamentally different. We recognize that LSQs for HLS don't have to be as general as CPU LSQs. We propose to break up the single store queue CAM into two separate shift-register based queues, one holding just store address allocations and the other store commits. Compiler analysis allows us to size the shift-registers exactly. Instead of the single-cycle CAM access in Dynamatic, we spread our memory disambiguation checks into multiple pipeline stages for an improved critical path and resource usage. Another major difference is our support for speculative address allocations, enabled by having separate store allocation and commit queues. Our LSQ approach can be seen as a generalization of shift-registers based approaches to pipelining of loops with statically analyzable dependency distances [25], e.g., sparse matrix-vector multiply accelerators [26].

The central question in any LSQ design aimed at spatial computing is how to recover program order of memory requests without a program counter. Josipović *et al.* proposed to allocate LSQ addresses from a single basic block in parallel and sequentialize the execution of basic blocks [10]. Memory operations within a single basic block can be disambiguated statically, while the semantics of their dataflow circuits guaranteed the sequential execution of basic blocks in program order. Our LSQ doesn't rely on the sequential execution of basic blocks. Instead, we recover program order by tagging each memory request with a unique integer representing the state of memory at that time. Our tags are similar to the work by Elakhras *et al.* [27] who addressed the sequentialized block allocation problem of the Dynamatic LSQ by introducing virtual data dependencies between blocks with LSQ accesses. However, in addition to ordering the allocation of addresses, we also use the actual tag values for disambiguation inside the LSQ.

## III. The Memory Disambiguation Problem

We define an LSQ allocation as an $(address, tag)$ tuple. The tag is an integer indicating the state of memory expected by the allocation. We define memory states as a sequence $\sigma = \{0, 1, 2, ...\}$, where each $i \in \sigma$ corresponds to the memory state of the original sequential program after the $i$-th store, with the state at $i = 0$ representing the initial memory state.

The inputs to our LSQ are: a sequence of load allocations; a sequence of store allocations; a sequence of store values where each $stValue_i$ corresponds to the $stAllocation_i$. We require that store allocations and store values arrive in program order. The LSQ outputs a sequence of load values, which correspond to the sequence of previously made load requests.

The tag of a load allocation indicates which memory state is expected by the load; the tag of a store allocation represents the new memory state after the store. Given any pair of $ldAllocation_i$ and $stAllocation_k$, if the two conditions hold:

$$\begin{aligned} ldAllocation_i.address = stAllocation_k.address, \\ ldAllocation_i.tag \geq stAllocation_k.tag, \end{aligned} \tag{1}$$

then $ldAllocation_i$ cannot be served before observing the side-effect of $stAllocation_k$.

Finally, we define a store commit as an $(address, value)$ tuple. Our LSQ holds a sequence of store commits internally, representing values in-flight to memory. Store commits can be used to forward stored values directly to aliasing loads. Note the omission of program ordering information from the store commits. In previous LSQs, in the case when a load aliases multiple store commits, the forwarding logic had to pick the youngest store commit. In our case, this would require adding a $tag$ field to the store commit tuple, and finding a store commit with the maximum $tag$ value. We avoid the need for this logic by keeping store commits ordered, and by ensuring that the store commits don't contain stores that in program order come after a load that has not yet been served.

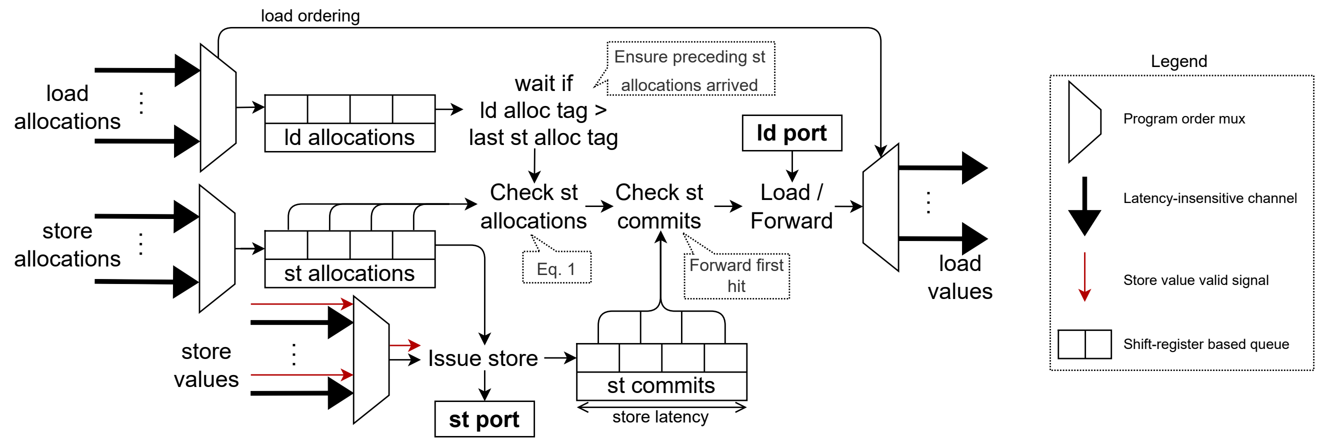


Fig. 2. Our shift-register based load-store queue design. Load allocations are checked for aliasing over multiple pipeline stages. Store values carry an optional $valid$ bit, allowing the LSQ to drop store allocations corresponding to invalid store values, and thus enabling speculative address allocations.

## IV. Load-Store Queue Design

We now present the design of our load-store queue (LSQ). We show how loads and stores are executed, and how we support speculative address allocations. We also discuss how our LSQ can protect both on-chip and off-chip memory.

### A. Load-Store Queue Overview

Fig. 2 shows an overview of our LSQ design protecting a memory with one load and one store port. Load/store allocation queues and the store commit queue are implemented as shift registers in FIFO order. The store queue is broken up into two separate queues: one for store allocations and one for store commits. The store commit queue holds stores for the duration between store issue and memory commit – its size is equal to the maximum store latency which is program specific.

Our LSQ accepts one load allocation per cycle for every available load port to memory. Multiple load allocation sequences can be served in parallel as long as the number of sequences is not greater than the number of load ports. Our LSQ does not reuse load values between unique load ports. If there are more load allocation sequences than available load ports to memory, then the sequences are multiplexed according to program order (as is the case in fig. 2). Multiple store allocation sequences, and their corresponding store value sequences, are always multiplexed in program order, regardless of the amount of memory store ports available. This restriction protects against write-after-write hazards by construction.

### B. Load and Store Execution

*Load execution:* A given $ldAllocation_i$ at the head of the load allocation queue compares its tag to the latest accepted store allocation tag, and waits if its tag is higher. This tag check ensures that all store allocations coming before $ldAllocation_i$ in program order have arrived to the LSQ. Next, $ldAllocation_i$ checks all store allocations in the store allocation queue in program order for conflicts using eq. 1 (in our implementation, this is done in two pipeline stages). If there are no conflicts within the store allocation queue, then we check the store commit queue next. At this point, the store commit queue is guaranteed to only hold stores that come before $ldAllocation_i$ in program order. In the store commit queue, we check from the youngest to the oldest store and forward the first (i.e. youngest) value that matches the address of $ldAllocation_i$. If there is no hit in the commit queue, then we can safely load the value from memory and return it via a non-blocking latency-insensitive channel to the datapath.

*Store execution:* A given $stAllocation_j$ at the head of the store allocation queue waits for its corresponding store value to arrive. On the arrival of the awaited store value, a store is immediately issued to memory and a $stCommit_j$, holding the store address and store value, is shifted into the store commit queue. The corresponding $stAllocation_j$ is shifted away from the store allocation queue. A store can only be in the store allocation or store commit stage, but never both. The store commit queue holds on to the store value until it is guaranteed to have been committed to memory. This requires to set the size of the store commit shift register to the maximum memory store latency for the given code using the LSQ.

*Speculation support:* Our LSQ can support *speculative store allocations* by extending each store value with a valid bit. Valid store values are handled without change. Invalid store values are not stored to memory and are not shifted into the store commit queue. Invalid store values still cause the corresponding store allocation to be shifted away. This mechanism allows to speculatively allocate store addresses to the LSQ with no requirement for replays because *a misspeculated store allocation is never actually committed.* Sec. V-D shows how the compiler creates speculative allocations.

### C. Scalability to Off-Chip Memories

Our LSQ design can be used to protect both on-chip and off-chip memory from data hazards. Our LSQ can exploit multiple load ports in parallel. Multiple store ports cannot be exploited by our design – to protect write-after-write hazards, we multiplex multiple store sequences onto one store port.

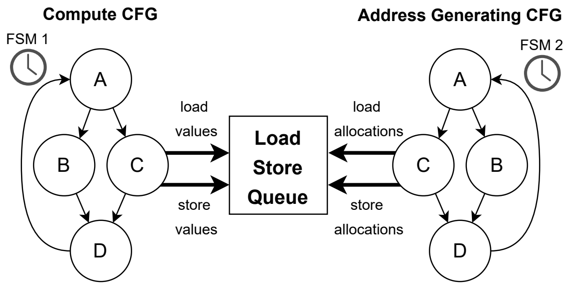


Fig. 3. The generation of addresses is decoupled into a separate modulo-scheduling instance, capable of generating load-store queue address allocations out-of-order w.r.t. the actual memory accesses in the compute loop.

To support multi-cycle memory we grow the size of the store commit queue to cover the maximum store latency. To avoid stalls in the LSQ when issuing a multi-cycle variable-latency memory operation, we decouple the load and store ports from the LSQ pipeline and connect them using latency-insensitive buffers with a deterministic write-to-read latency. To preserve the correctness of memory disambiguation, we grow the store commit queue by this added latency.

## V. Compiler Integration

In this section, we first show how an HLS compiler can use our LSQ. Then, we describe how to enable dynamically scheduled out-of-order loads in static HLS. Next, we present a compiler algorithm for introducing speculative address allocations to our LSQ. Finally, we show how parts of the LSQ can be specialized based on the target code.

We use existing compiler analysis' to find memory base addresses with data hazards [28]. Each selected base address uses its own LSQ. All memory operations using a selected base address are be transformed into read/writes from/to latency-insensitive channels connected to an LSQ. The channels to an LSQ can be reused across basic blocks if they are guaranteed not to execute in the same clock cycle, similar to how FPGA block RAM ports can be shared.

Our LSQ design uses integer tags to recover program order of memory operations. Each address generating unit has a tag corresponding to a single LSQ, initially set to zero. Store allocations increment the tag before using it; load allocations use the tag directly. This creates a data dependency between a store allocation and any other LSQ allocation following that store allocation in program order, thus ensuring the correct order of the store allocation sequence.

### *A. Dynamically Scheduled Memory in Static HLS*

Altough our LSQ can be used by any HLS tool, in this paper we assume it is used by a statically scheduled HLS compiler. This subsection describes the transformation needed in static HLS to enable the efficient use of our LSQ and can be omitted if the LSQ is used in fully dynamic HLS.

The throughput of circuits using an LSQ depends on the number of addresses that can be disambiguated ahead of their actual memory operation execution – call this the *out-of-order address window*. In a statically scheduled pipeline, the out-of-order address window can be at most one – address generation and memory access proceed in lockstep. In dataflow circuits, the generation of memory addresses is naturally decoupled from the memory operation and allows for much larger out-of-order address windows. To achieve the same effect in static HLS, we follow a method from our previous work [13] to decouple the generation of memory addresses into a separate static pipeline, similar to the principle in decoupled access/execute architectures [29]–[31]. Fig. 3 illustrates the resulting communication pattern. The address generating unit will contain only address generating instructions and will run ahead w.r.t. the compute unit, increasing the out-of-order address window in our LSQ.

### *B. Loss of Address Decoupling*

In some cases, the decoupling of address generation cannot result in the run-ahead of address allocations. Such situations, called "loss of decoupling" [30], arise when the address generation for a given base address depends on a value loaded from the same base address, i.e. a load value from an array is used to generate a load/store address to the same array. Formally, given a set of address generating instructions $G$ for a given base address, and a set of memory access instructions $A$ using addresses generated by instructions in $G$, there is a loss of decoupling if:

$\exists i \in G$, such that $i$ depends on an instruction $j \in A$, i.e. there is a path from $i$ to $j$ in the DDG.

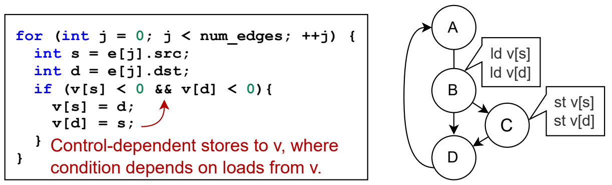
We do not perform address decoupling in such cases, because the address allocations and their memory operations need to be in effect synchronized. This is not a drawback of using static HLS since a fully dynamically scheduled circuit would also synchronize the two sequences.

Note that our loss of decoupling definition is more relaxed than previous work. We only consider direct data dependencies, ignoring control dependencies. We next show how we use speculation to maintain decoupling of address generation in cases where a memory operation is control dependent, such that the control decision itself depends on a loaded value from the LSQ. Our approach allows us to maintain a high out-of-order address window, even in cases where a fully dynamic HLS compiler would suffer a loss of decoupling.

### *C. Intuition for Speculative Address Allocations*

A memory operation using a given base address can be control-dependent on a branch condition that itself is data-dependent on a value loaded from the same base address. Consider the code in fig. 4a as an example. Here, the execution of the stores to `v` is control dependent on the if-condition which itself uses values loaded from `v`. Under the execution model of both dynamic HLS [8] and our decoupled address architecture (sec. V-A), there is no possibility for out-of-order address allocations in this code. We propose the concept of *speculative address allocations* to relax this restriction.

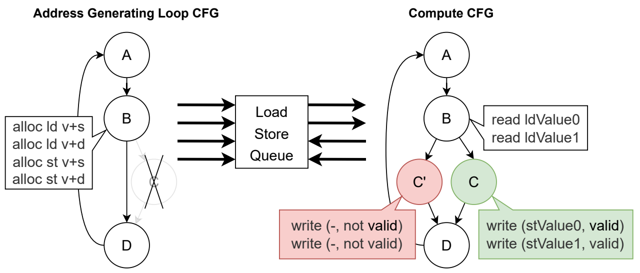
Consider the code in fig. 4a again. Although the store execution is control-dependent, the store addresses have no



```
for (int j = 0; j < num_edges; ++j) {
  int s = e[j].src;
  int d = e[j].dst;
  if (v[s] < 0 && v[d] < 0){
    v[s] = d;
    v[d] = s;
  }
}
```

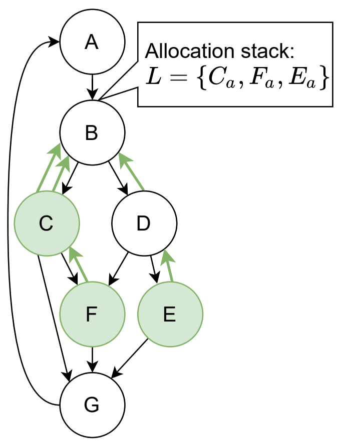
Control-dependent stores to v, where condition depends on loads from v.
A
B
C
D
ld v[s]
ld v[d]
st v[s]
st v[d]

(a) Maximal Matching code and its control-flow graph (CFG).

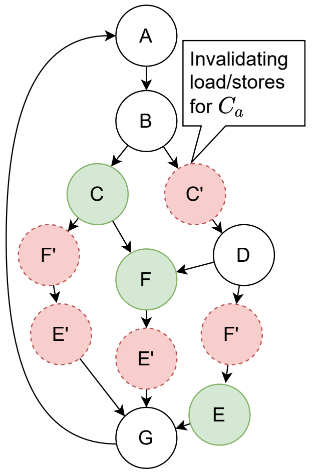


(b) Our transformation: speculative address allocations in the address generation loop (left), and invalidated store value writes on misspeculation (right).

Fig. 4. Speculative store address allocations in the maximal matching code.



(a) Iterative hoisting of speculative address allocations (green blocks) to their special-control dependency source block.



(b) Insertion of poison basic blocks (red blocks) with invalidating loads/stores to deque misspeculated address allocations.

Fig. 5. A visualization of our CFG transformations to enable speculative LSQ address allocations. Left: address generating CFG; right: compute CFG.

data dependency on values loaded from v. We can hoist the address generating instructions out of the if-condition in the address generating CFG (illustrated in fig. 4b). As a result, store address allocations will be produced without having to evaluate the if-condition.

Addresses allocated to the LSQ, but later not used, are said to be *misspeculated*. Misspeculations are handled in the compute CFG by inserting invalid LSQ store value writes on CFG paths containing misspeculations (e.g. basic block $C'$ in fig. 4b). An invalid LSQ store has the $valid$ bit set to 0 and will result in the deallocation of the misspeculated address allocation in the LSQ (sec. IV-B describes the LSQ support). Handling misspeculated loads is trivial, since a load doesn't have side effects and the loaded value can simply be discarded.

This compiler speculation approach can achieve a high degree of out-of-order loads such as in fig. 4a, without having to suffer the cost of expensive misspeculation replays common in load-value-based speculation approaches.

### D. Compiler Generated Speculative Address Allocations

We now formalize and generalize the transformation from the previous section. Assume a single LSQ connected to an address generating CFG and a compute CFG. Assume the CFG is a single loop; the ordering of speculative allocations across loops is trivial. The key question that we answer is how to preserve the relative order between speculative address allocations made in the address generating CFG and the invalidating load/stores in the compute CFG.

*Definition 1:* Let a *special control-dependency* relationship be a control-dependency between basic blocks $A$ and $B$ (written $A \xrightarrow{scd} B$), such that $A$ is control-dependent on $B$, $B$ is not a loop header, and the branch in $B$ depends on values loaded from an LSQ. If $A \xrightarrow{scd} B$, then we say that $B$ is the special control-dependency source block of $A$.

*Definition 2:* Let $\mathbb{B}$ be the set of basic blocks with memory operations selected to be routed through an LSQ, such that each $B \in \mathbb{B}$ has a special control dependency.

*Definition 3:* Let $B_a$ be the set of all address allocations for a given block $B \in \mathbb{B}$. Since the special control-dependency relation applies to whole blocks, all $B_a$ allocations are speculative. Ordering of invalidations within a block is trivial.

*Definition 4:* Let a *poison basic block* $B'$ be a basic block which invalidates misspeculated LSQ address allocations $B_a$ corresponding to memory operations in block $B$. Each CFG path should contain either $B$ or $B'$, but never both (a speculated address allocation should either be used or invalidated on every path, but never both).

*Definition 5:* Block $B$ becomes unreachable when the CFG $edge = (E_{start}, E_{end})$ is taken if there exists a CFG path from $E_{start}$ to $B$ but not from $E_{end}$ to $B$. Only paths within a loop are considered, loop back edges constitute the end of a path.

*In the address generating CFG*: we iteratively move up the address allocations $B_a$ for every $B \in \mathbb{B}$ to the end of its special control-dependency source block. If a $B \in \mathbb{B}$ block has multiple such source blocks, then we pick one at random. Every special control-dependency source block keeps a stack of address allocations moved to it. We first push on the stack allocations moved from the left sub-graph, then the right (the choice between left and right is arbitrary but has to be consistent). When there are no more basic blocks with LSQ allocations that have a special control-dependency, then we stop. At this point, each block in the CFG that contains speculative allocations will also have a stack exactly representing the order of these allocations.

*Example:* fig. 5a shows an address gemerating CFG with basic blocks $C$, $F$, and $E$ containing LSQ address allocations,

**Algorithm 1** Insertion Of Poison Basic Blocks

```
Input: loop CFG; basic block $B_{spec}$; allocation stack $L$
for $B_a \in L$ do
    for $edge \in$ cfg_traversal($B_{spec}$) do
        $C_1 \leftarrow B$ becomes unreachable when $edge$ is taken
        $C_2 \leftarrow$ no $D_a \in L$ s.t. $D$ is reachable from
                $edge$ and $D_a$ precedes $B_a$ in $L$
        if $C_1$ and $C_2$ then
            create poison block $B'$ on $edge$
```

and $F \xrightarrow{scd} C$, $F \xrightarrow{scd} D$, $E \xrightarrow{scd} D$, $C \xrightarrow{scd} B$. There will be two iterations of hoisting. On the first iteration, $F_a$ moves to $C$, $E_a$ moves to $D$, and $C_a$ moves to $B$. On the second iterations, $F_a$ (now in block $C$) moves to $B$, and $E_a$ (now in block $D$) moves to $B$. Block $B$ has no special control-dependency, so we stop after two iterations. Block $B$ will have the following speculative allocation stack: $\{C_a, F_a, E_a\}$.

*In the compute CFG*: we insert new basic blocks with invalidating LSQ loads/stores on CFG edges where the memory operation corresponding to a given speculated address allocation becomes unreachable. Alg. 1 presents pseudocode for our insertion procedure. It takes as input a basic block $B_{spec}$ and stack $L$. Stack $L$ contains ordered speculative address allocations hoisted to $B_{spec}$ (the result of the hoisting procedure from the previous paragraph).

*Theorem.* Alg. 1 transforms the compute CFG such that on every CFG path starting at $B_{spec}$ each speculated address allocation $B_a \in L$ is either used or invalidated, but never both. And the relative order of uses or invalidations matches the order of speculated allocations in the address generating CFG, i.e. in stack $L$ order.

*Proof.* The proof follows from the construction of alg. 1. The algorithm goes over all $B_a \in L$ in their allocation order. For each such $B_a$, it visits every CFG edge dominated by $B_{spec}$ in control-flow order. At each edge, an invalidating $B'$ will only be inserted if taking that edge will make block $B$ unreachable (condition $C_1$), and if preceding allocations in $L$ have already been used or invalidated (condition $C_2$). Thus, on every CFG path starting at $B_{spec}$, each $B_a \in L$ will have either been used or invalidated (but not both) in the $L$ allocation order. ■

*Example:* fig. 5b shows how poison blocks would be inserted given the address generation loop from fig. 5a. Note how $E'$ is not inserted on the $(B, C)$ edge because of condition $C_2$ in the algorithm: $C_a$ precedes $E_a$ in $L$ and is still reachable from the $(B, C)$ edge.

The transformation in the compute CFG has no misspeculation overhead. Any superfluously created basic block will be removed using existing CFG simplification algorithms. After simplification, the example CFG from fig. 5b would only have two, not five, new basic blocks (one on the $(C, G)$ edge, and one on the $(D, E)$ edge). For some speculation scenarios there need not be any poison blocks, e.g., in if-then-else branches, where each branch contains the same memory operations using the same address expressions.

### *E. Optimal Store Allocation Queue Size*

The optimal size of our store allocation queue depends on the target loop initiation interval (II). Assume a target II of 1, and a loop datapath as presented in our motivating example in fig. 1a. Assume `f(x)` has a latency of $L$ and that there are no true data hazards, so an actual II of 1 is possible at runtime. To achieve this II, at iteration $N$ our LSQ should be able to disambiguate a load address for iteration $N + L$. This requires the LSQ to be able to hold $L$ store allocations to cover all store addresses for the $[N, N + L]$ iteration range. Thus, the optimal store allocation queue size in this case is equal to the maximum latency between a dependent load and a store, call this $maxLoadToStoreDelay$ (for most codes, this is equal to the recurrence constrained II discussed in the introduction). The optimal size will increase if there are multiple stores in the loop datapath, call this number $numStoresInLoop$. All of the above information is static, allowing us to find an optimal store allocation queue size at compile time:

$$\left\lceil \frac{maxLoadToStoreDelay}{targetII} \times numStoresInLoop \right\rceil$$

Tab. II shows how the resource usage and critical path of our LSQ scales with the size of the store allocation queue.

## VI. Evaluation and Results

In this section, we evaluate our work against two commercial HLS compilers (Intel HLS [32] and Vivado HLS [33]), and against a dynamically scheduled academic HLS compiler that uses a state of the art LSQ [8]. We also show how our LSQ design scales with the size of its store allocation queue.

### *A. Methodology*

Our compiler analysis' and transformations are implemented as LLVM passes, and we integrated them with the Intel SYCL HLS compiler [34]. We automatically find data hazards in the input code, decouple the address generation into separate modulo-scheduled pipelines (separate SYCL kernels), and connect memory requests to an LSQ specialized to the input code. Our compiler passes and LSQ are publicly available[1].

We evaluate our work against the dynamic HLS tool Dynamatic using a research artifact from their most recent paper [27]. Cycle counts were obtained using ModelSim and are compared directly between all tools. Dynamatic uses Vivado for synthesis, while we use Intel tools, making a direct comparison of area and circuit frequency in absolute terms difficult. Instead, we compare the *normalized* frequency, execution time, and area overhead of Dynamatic and our approach against their respective static HLS baseline. For Dynamatic we used Vivado 2019.2 and the Xilinx xc7k160tfbg484 FPGA. For our approach, we used Quartus 19.2 and the Altera 10AX115S FPGA. All benchmarks are integer based to avoid differences in floating point performance across FPGAs. When comparing

[1]https://github.com/robertszafa/elastic-sycl-hls

TABLE I
A COMPARISON OF OUR WORK AGAINST VIVADO, DYNAMATIC [27], AND INTEL HLS. ALL CODES USE ON-CHIP BRAM.

| **Benchmark** | **Cycles (thousands)** | | | | **Freq. (MHz)** | | | | **Execution time ($\mu$s)** | | | | | | **Area (Slices / ALMs)** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | V | D | I | O | V | D | I | O | V | D | D/V | I | O | O/I | V | D | D/V | I | O | O/I |
| histogram | 2 | 1–3 | 2.1 | 1-2 | 379 | 155 | 379 | 337 | 5.3 | 6.5-19.4 | **1.23–3.68** | 5.5 | 3–6 | **0.55–1.09** | 129 | 5582 | **43.3** | 1814 | 9847 | **5.4** |
| getTanh | 68 | 2.5–79 | 56.2 | 1.1–59 | 266 | 89 | 377 | 263 | 256 | 28.1–888 | **0.11–3.47** | 149 | 4.1–224 | **0.03–1.51** | 572 | 22399 | **39.2** | 1825 | 14753 | **8.1** |
| getTanhDouble | 14 | 1–19 | 13.1 | 1–16 | 304 | 96 | 330 | 297 | 46.1 | 10.7–198 | **0.23–4.3** | 39.8 | 3.5–53.9 | **0.1–1.36** | 245 | 22103 | **90.2** | 3803 | 18730 | **4.9** |
| vecTrans | 30 | 1.5–31 | 30.1 | 1.1–33 | 304 | 97 | 365 | 291 | 98.7 | 15.9–320 | **0.16–3.24** | 82.5 | 3.7–113 | **0.05–1.38** | 125 | 22997 | **184** | 1811 | 11672 | **6.5** |
| spmv | 2.3 | 0.8–2.7 | 3.6 | 0.8–2.8 | 263 | 152 | 328 | 280 | 8.7 | 5.2–17.6 | **0.6–2.02** | 10.9 | 3.1–9.9 | **0.28–0.9** | 494 | 5628 | **11.4** | 5255 | 23406 | **4.5** |
| chaosNCG | 72 | 37–74 | 74.3 | 2.1–77 | 308 | 155 | 335 | 237 | 234 | 239–477 | **1.02–2.04** | 222 | 8.9–325 | **0.04–1.47** | 779 | 2017 | **2.6** | 5274 | 36266 | **6.9** |
| BNN | 20 | 15–30 | 20.7 | 10.4–20.4 | 258 | 116 | 365 | 284 | 77.5 | 129–259 | **1.67–3.34** | 56.9 | 36.8–71.7 | **0.65–1.26** | 1214 | 7466 | **6.2** | 4214 | 20222 | **4.8** |
| histogramIf | 2 | 5–6 | 2.1 | 1-2 | 388 | 117 | 379 | 328 | 5.15 | 42.7-51.3 | **8.29–8.3** | 5.5 | 3–6 | **0.56–1.12** | 155 | 5395 | **34.8** | 1814 | 10452 | **5.8** |
| matching | 6 | 6–8 | 7.6 | 2–8.8 | 404 | 110 | 246 | 291 | 14.9 | 54.6–72.7 | **3.67–4.9** | 30.9 | 7–30.2 | **0.23–0.98** | 141 | 3778 | **26.8** | 11551 | 18310 | **1.6** |
| floydWarshall | 6.2 | 7–11 | 6.3 | 3.4 | 366 | 90 | 229 | 263 | 16.9 | 77.8–122 | **4.59–7.2** | 27.3 | 13.2 | **0.48** | 255 | 2226 | **8.7** | 807 | 5757 | **7.1** |
| bitonicSort | 3.1 | 2.6–6.1 | 9.6 | 1.5 | 300 | 97 | 248 | 273 | 10.4 | 26.9–62.8 | **2.58–6** | 38.8 | 5.3 | **0.14** | 51 | 5683 | **111** | 12310 | 18252 | **1.5** |
| Harmonic mean | | **0.15–1.4** | | **0.07–0.49** | | **0.35** | | **0.87** | | | **0.45–3.67** | | | **0.09–0.52** | | | **12.3** | | | **3.8** |

V – Vivado HLS    D – Dynamatic    I – Intel HLS    O – Our work

against Dynamatic, we only consider codes using on-chip BRAM (we could not use DRAM in Dynamatic).

We applied our approach to eleven benchmarks with data hazards used in previous work [8], [12]. The codes and evaluation results for all tools are available as a public artifact [35]. The addresses in the first seven benchmarks can be decoupled without speculation:

1) *histogram* is the code from fig. 1a (loop II=2).
2) *getTanh* performs a *tanh(x)* approximation on a sparse array (loop IIs=56, 1, 1).
3) *getTanhDouble* is similar but uses only one loop, not three (loop II=13).
4) *vecTrans* applies a polynomial expression on elements of a sparse array (loop II=30).
5) *spmv* is a sparse matrix-vector multiply (loop IIs=1, 9).
6) *chaosNCG* is a function from a chaos engine with data-dependent loads and stores (loop II=74).
7) *BNN* is a binarized neural network (loop IIs=1, 2, 2).

The remaining benchmarks have control-dependent stores, making our speculative address allocation approach applicable:

8) *histogramIf* is similar to *histogram*, but the store is control dependent on the load value (loop II=2).
9) *matching* is the code example from fig. 4a (loop II=7).
10) *floydMarshall* finds shortest paths in a weighted digraph (loop IIs=1, 1, 6).
11) *bitonicSort* sorts a list of integers using a bitonic merge network (loop IIs=1, 1, 7).

We report worst- and best-case performance, which depends on the true number of data hazards in the input data distribution. We automatically choose our store allocation queue size according to sec. V-E. For Dynamatic, we manually choose the smallest queue size that enables perfect pipelining in the case of no data hazards, following their approach [24].

### B. BRAM Results

*Speedup:* Fig. 6 shows that our approach achieves a higher speedup than Dynamatic when comparing each tool to their respective static HLS baseline. On most codes, the higher speedup is due to the higher frequency achievable by our LSQ. On some codes (e.g. *chaosNCG* or *getTanh*), our approach is also able to achieve a better throughput, because we can support the large store queue sizes required by these codes while Dynamatic cannot (the maximum store queue size that we could use in Dynamatic was 32).

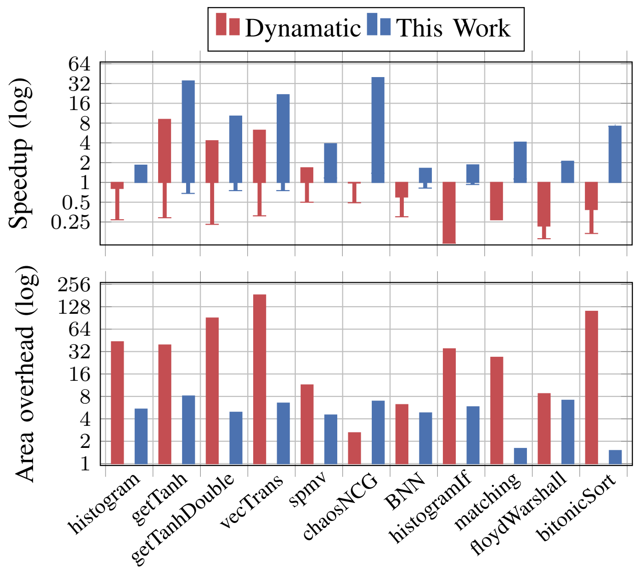


Fig. 6. Speedup and area overhead of our work and Dynamatic [27] compared to their static HLS baselines (Intel HLS and Vivado, respectively). The range bars represent the speedup range, with a value below 1 indicating a slowdown.

Tab. I shows detailed benchmark results. On average, designs with our LSQ achieve 87% of the frequency achieved by Intel HLS, whereas Dynamatic LSQ designs achieve a frequency of 35% compared to Vivado. We also see that the Dynamatic LSQ results in throughput overhead when the data distribution favors static scheduling, i.e. when most iterations have a true data hazard. In such cases, the Dynamatic designs

TABLE II
Scalability of our store allocation queue compared to the store queue in Dynamatic [27] on the histogram benchmark.

| Queue Size | Freq (MHz) | | | | Area (Slices / ALMs) | | | |
|---|---|---|---|---|---|---|---|---|
| | Dyn | × | Ours | × | Dyn | × | Ours | × |
| No LSQ | 379 | **1** | 379 | **1** | 129 | **1** | 1814 | **1** |
| 2 | 173 | **0.46** | 340 | **0.9** | 409 | **3.2** | 9134 | **5** |
| 4 | 178 | **0.47** | 330 | **0.87** | 684 | **5.3** | 9369 | **5.2** |
| 8 | 163 | **0.43** | 337 | **0.89** | 1554 | **12** | 9847 | **5.4** |
| 16 | 155 | **0.41** | 281 | **0.74** | 5582 | **43** | 10264 | **5.7** |
| 32 | 92 | **0.24** | 294 | **0.78** | 22580 | **175** | 11608 | **6.4** |
| 64 | - | - | 244 | **0.64** | - | - | 14309 | **7.9** |
| 128 | - | - | 224 | **0.59** | - | - | 20515 | **11** |
| 256 | - | - | 175 | **0.46** | - | - | 40407 | **22** |

need on average 1.4× more cycles to finish than the Vivado designs, rising to 3.5× more execution time due to their lower frequency. On average, our approach has no overhead in execution time compared to its Intel HLS baseline. The slight increase in the number of cycles for some codes is only for data distributions that repeatedly read and write to the same memory location, which is a highly unlikely scenario.

The last four codes benefit from our speculative address allocation scheme, allowing for non-trivial speedups compared to all other evaluated tools. These codes contain stores which depend on a control decision that uses loads from the same array. A dataflow circuit, as produced by Dynamatic, suffers a loss of address generation decoupling and doesn't give throughput improvements compared to a static pipeline. Our speculation approach doesn't suffer from loss of decoupling, allowing for improved pipelining. The results also confirm that our approach doesn't suffer any misspeculation overhead.

*Area overhead:* In addition to a better speedup, our LSQ also has a lower area overhead than Dynamatic. On average, our LSQ results an 3.8× area overhead compared to 12.3× for Dynamatic, and that is despite the fact that for several codes we use a larger queue size.

*Store queue size scalability:* Some benchmarks require a large out-of-order address allocation window for perfect pipelining and thus large store queues in the LSQ. Previous LSQ designs targeting FPGAs are notorious for their poor scalability [21], [24]. Our shift-register-based queues scale better, allowing for hundreds of store queue entries, compared to a maximum of 32 in the Dynamatic LSQ. Tab. II shows how the frequency and area usage changes with the size of our store allocation queue. There is a correlation between the required size of our store allocation queue, and the potential throughout increase of using our LSQ – even though the maximum circuit frequency of circuits using large LSQs will go down, the potential throughout increase will go up.

### C. DRAM Results

Tab. III shows the speedups over static Intel HLS that are possible when using our LSQ to protect DRAM. In this experiment, we report execution time when running in hardware on the Intel PAC Arria 10 GX FPGA board using

TABLE III
Performance of our LSQ when protecting off-chip DRAM.

| Benchmark | Exec. Time ($\mu$s) | | | Freq. (MHz) | | | Area (ALMs) | | |
|---|---|---|---|---|---|---|---|---|---|
| | I | O | O/I | I | O | O/I | I | O | O/I |
| histogram | 363 | 43.7–61.3 | **0.12–0.17** | 273 | 272 | **1** | 19832 | 19647 | **1** |
| getTanh | 564 | 36.9–150 | **0.08–0.27** | 281 | 205 | **0.73** | 27365 | 35373 | **1.3** |
| getTanhDouble | 396 | 35.8–122 | **0.09–0.31** | 281 | 235 | **0.84** | 26018 | 29051 | **1.1** |
| vecTrans | 441 | 40.6–182 | **0.09–0.37** | 305 | 241 | **0.79** | 20217 | 22621 | **1.1** |
| spmv | 158 | 40.8–63.5 | **0.26–0.34** | 287 | 256 | **0.89** | 7826 | 18313 | **2.3** |
| chaosNCG | 687 | 63.3–502 | **0.09–0.54** | 270 | 170 | **0.63** | 21190 | 37314 | **1.8** |
| BNN | 4167 | 336–636 | **0.08–0.15** | 264 | 241 | **0.91** | 10916 | 17459 | **1.6** |
| histogramIf | 362 | 34.2–61.8 | **0.09–0.17** | 274 | 248 | **0.91** | 19903 | 20950 | **1.1** |
| matching | 496 | 53.5–175 | **0.11–0.35** | 289 | 227 | **0.79** | 8655 | 18970 | **2.2** |
| floydWarshall | 300 | 59.9–98.4 | **0.21–0.33** | 257 | 250 | **0.97** | 31280 | 32214 | **1** |
| bitonicSort | 319 | 33.9–53.3 | **0.11–0.17** | 270 | 241 | **0.89** | 12587 | 24781 | **2** |
| Harmonic mean | | | **0.1–0.25** | | | **0.84** | | | **1.4** |

I – Intel HLS  O – Our work

dual-channel DDR4 memory. On average, using our LSQ results in an 4–10× speedup compared to Intel HLS. The store commit queue, needed to cover the maximum store latency to DRAM, has a cache-like effect which is more noticeable long latency DRAM accesses, compared to using BRAM. As a result, our LSQ still offers a significant speedup even if most of the iterations have a true data hazard. Circuits with DRAM connections use more resources, making the area overhead of our LSQ smaller (1.4× for DRAM vs. 3.8× for BRAM).

The DRAM benchmarks achieve on average a 7–10× lower throughput than the BRAM codes. Our DRAM load-store units do not take advantage of burst reads or writes that amortize the large off-chip memory latency in typical HLS designs. It is unlikely that DRAM bursts could be used effectively in an LSQ, because the memory access pattern of codes using LSQs is seldom contiguous.

## VII. Conclusion

We presented a novel, shift-register-based load-store queue (LSQ) design adapted to spatial architectures and tightly coupled with an HLS compiler that can specialize parts of the LSQ to a given target code. We introduced the concept of speculative address allocations to the LSQ, which enables out-of-order loads on a broader range of codes than before with no misspeculation overhead. Our LSQ design achieves a higher frequency and lower area overhead compared to previous LSQs used in HLS, resulting in an average speedup of 11× compared to static HLS and 5× compared to dynamic HLS. Our LSQ scales to queues with hundreds of entries, and can protect both on-chip and off-chip memory.

Acknowledgments

This work was partly supported by the UK EPSRC. We thank Intel for access to FPGAs through the FPGA DevCloud, and the anonymous reviewers for improving this paper.

## References

[1] M. Pellauer, A. Parashar, M. Adler, B. Ahsan, R. Allmon, N. Crago, K. Fleming, M. Gambhir, A. Jaleel, T. Krishna, D. Lustig, S. Maresh, V. Pavlov, R. Rayess, A. Zhai, and J. Emer, "Efficient control and communication paradigms for coarse-grained spatial architectures," *ACM Trans. Comput. Syst.*, 2015.

[2] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[3] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo sdc scheduling with recurrence minimization in high-level synthesis," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[4] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen, "Ilp-based modulo scheduling for high-level synthesis," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2968455.2968512

[5] A. Morvan, S. Derrien, and P. Quinton, "Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion," in *2011 International Conference on Field-Programmable Technology*, 2011.

[6] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, 2018.

[7] J. Cheng, J. Wickerson, and G. A. Constantinides, "Exploiting the correlation between dependence distance and latency in loop pipelining for hls," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 341–346.

[8] 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*, 2022.

[9] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017.

[10] L. Josipovic, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Transactions on Embedded Computing Systems*, 2017.

[11] J. Cheng, L. Josipović, G. A. Constantinides, and J. Wickerson, "Dynamic inter-block scheduling for hls," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 243–252.

[12] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Dass: Combining dynamic amp; static scheduling in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[13] R. Szafarczyk, S. W. Nabi, and W. Vanderbauwhede, "Compiler discovered dynamic scheduling of irregular code in high-level synthesis," to appear in 2023 33nd International Conference on Field-Programmable Logic and Applications (FPL), 2023.

[14] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[15] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006.

[16] K. E. Fleming, "Scalable reconfigurable computing leveraging latency-insensitive channels," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2013. [Online]. Available: https://hdl.handle.net/1721.1/79212

[17] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 434–441.

[18] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic cgras," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13, 2013.

[19] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 78–85.

[20] J. Cheng, J. Wickerson, and G. A. Constantinides, "Finding and finessing static islands in dynamically scheduled circuits," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 89–100. [Online]. Available: https://doi.org/10.1145/3490422.3502362

[21] H. Wong, V. Betz, and J. Rose, "Efficient methods for out-of-order load/store execution for high-performance soft processors," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 442–445.

[22] B. Thielmann, J. Huthmann, and A. Koch, "Memory latency hiding by load value speculation for reconfigurable computers," *ACM Trans. Reconfigurable Technol. Syst.*, 2012.

[23] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 189–194. [Online]. Available: https://doi.org/10.1145/3020078.3021754

[24] J. Liu, C. Rizzi, and L. Josipović, "Load-store queue sizing for efficient dataflow circuits," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–9.

[25] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[26] A. Jain, C. Ravishankar, H. Omidian, S. Kumar, M. Kulkarni, A. Tripathi, and D. Gaitonde, "Modular and lean architecture with elasticity for sparse matrix vector multiplication on fpgas," to appear in 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines, 2023.

[27] A. Elakhras, R. Sawhney, A. Guerrieri, L. Josipovic, and P. Ienne, "Straight to the queue: Fast load-store queue allocation in dataflow circuits," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 39–45. [Online]. Available: https://doi.org/10.1145/3543622.3573050

[28] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[29] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Washington, DC, USA: IEEE Computer Society Press, 1982, p. 112–119.

[30] T. J. Ham, J. L. Aragón, and M. Martonosi, "Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, jun 2017. [Online]. Available: https://doi.org/10.1145/3075620

[31] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[32] "Intel HLS." [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[33] "Xilinx Vivado HLS." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/high-level-design.html

[34] Intel, "Intel/llvm." [Online]. Available: https://github.com/intel/llvm/tree/sycl

[35] R. Szafarczyk, "robertszafa/elastic-sycl-hls: Artifact FPT23," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10040353