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) achieves a higher throughput on codes with unpredictable memory accesses compared to statically scheduled HLS. However, the increased throughput comes at the price of increased resource usage and critical path length, resulting in lower clock frequency. The decrease in clock frequency can be significant, often nullifying any throughput improvements over static scheduling. Recent work presented methods for combining static and dynamic scheduling to achieve high throughput circuits with a fast critical path for dynamic codes. However, circuits that require dynamically scheduled memory still suffer from a decreased frequency. This paper fills this gap by presenting a method for achieving dynamically scheduled memory operations in HLS with a high frequency. Dynamic scheduling of memory operations is realized with a load-store queue (LSQ). We present a novel LSQ design adapted to the nature of spatial architectures with aggressive specialization to the target code – a unique opportunity in HLS. Our LSQ design works for both on-chip and off-chip memory and is integrated with a compiler that combines dynamic and static scheduling. 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, our approach adds no overhead on misspeculation. On a set of ten benchmarks, we show that our approach can achieve an up to 10× speedup on average against static HLS, and an up to 4× speedup against dynamic HLS that uses an LSQ from previous work, while also using several times fewer resources and scaling to larger queues.**

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 to the compiler.

Loop pipelining is a critical step in any HLS compiler. 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 minimal number of cycles that need to pass between the start of subsequent loop iterations such that any data dependencies across iterations are honored. 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:

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

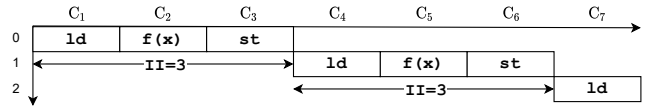
where $delay$ is the number of cycles needed to traverse the recurrence path, and $distance$ is the number of iterations between the definition of a recurrence value and its use.

Static HLS tools rely on an accurate memory dependency analysis to discover the dependence $distance$ of 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. every iteration needs to wait for all previous iterations to finish, resulting in the final II being equal to the loop delay and eliminating any possibility for loop pipelining. In our example, the execution of loop iterations would be sequentialized by static HLS tools, as seen in fig. 1b.

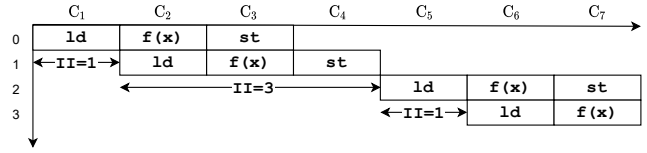
An alternative approach to achieve loop pipelining is to use dynamic scheduling. Dynamic HLS uses dataflow scheduling to trigger the execution of operations at runtime based on the availability of data, rather than a static compile-time schedule. Dynamatic [8], the most recent example of a dynamic HLS tool, has shown that this approach can successfully accelerate

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

(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).

codes where compile-time information needed by modulo scheduling is lacking. For our example code from fig. 1a, dynamic HLS can achieve the ideal schedule in fig. 1c. However, dynamic HLS incurs non-trivial resource and critical path overheads [8]. On codes that require dynamically scheduled memory operations, the highest overhead comes from load-store queue (LSQ) structures. The low frequencies achievable by circuits using LSQs often nullify any throughout advantage over static HLS [9]. Recent work has shown the possibility of intelligently combining static and dynamic scheduling to achieve the high throughput of dynamic scheduling with the low critical path of static HLS [10]. However, whenever an LSQ is needed by the dynamic part of such a combined circuit, the 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 a runtime memory disambiguation mechanism 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 exploiting the unique features of a spatial architecture and the possibility of aggressively specializing to the target code (sec. IV) We show how an LSQ can be used in static HLS to achieve dynamically scheduled memory operations without sacrificing frequency (sec. V). Our LSQ design supports both on-chip and off-chip memory.
- An extension to our LSQ and compiler that enables *speculative address allocations*. We show that this extension enables out-of-order loads on more codes than previous work at no added cost (sec. V-B).
- An evaluation of our work against static HLS (Vivado HLS, Intel HLS), and against Dynamic HLS using a state-of-the art LSQ for spatial computing. We show that our approach achieves both better speedups and lower area overheads than Dynamatic (sec. VI). We achieve an up to $10\times$ speedup on average against Intel HLS, while Dynamatic achieves a maximum speedup of $2.4\times$ against Vivado. We also show that our LSQ can accelerate codes using off-chip multi-cycle 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.* [11] and simplified by Cortadella *et al.* for synchronous circuits [12]. In latency-insensitive designs, the communication between modules is decoupled from their cycle behavior, allowing for dataflow scheduling of compute circuits [8], [13]–[16]. The most recent fully dynamically scheduled HLS tool is Dynamatic proposed by Josipović *et al.* [8]. Dynamatic is based on the LLVM compiler framework [17] translating its Control and Data Flow Graph (CDFG) SSA representation into a dataflow circuit. 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 simple wire. Several works have tackled the critical path overhead of Dynamatic. Josipović *et al.* formalized the problem of buffer sizing in dataflow circuits as a mixed-integer linear programming in order to minimize the buffer sizes and decrease the critical path. Xu *et al.* used linear temporal logic to prove that certain dataflow signals are not needed at all and can be removed. These approaches make incremental improvements to the final quality of results.

B. Combining Static and Dynamic Scheduling

Cheng *et al.* extended Dynamatic with the DASS methodology (Dynamic and Static Scheduling) [10], [18], 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 [19]. 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 to handle memory accesses at runtime [20]. 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 [21]. 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.* [22] 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. Value-based, compared to address-based, disambiguation is better able to pipeline loops where the store operation is control-dependent on a load [21]. 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 a load-store queue in codes where the address generation cannot otherwise run ahead of memory accesses.

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 a load-store queue (LSQ). Most LSQs aimed at HLS use a content-addressable memory (CAM) structure to implement the load and store queue [20], [23], with a similar operating principle as LSQs used in out-of-order CPUs [24]. CAMs map poorly to FPGA technology resulting in a high critical path and resource usage overhead [20], [25]. Our LSQ design is fundamentally different from previous LSQs in that we use shift registers instead of CAMs. Our shift-register based queues bare resemblance to hazard resolution units used in many sparse matrix-vector FPGA architectures [26] or in source-to-source approaches to loop pipelining [27]. We generalize these approaches into an LSQ to support out-of-order loads. We also recognize that the LSQs used in HLS don't have to be as general as CPU LSQs, allowing for aggressive specialization to the target code. Another difference in our approach is the support for speculative address allocation, which increases the size of the out-of-order loads window in codes where the address generation would otherwise be impossible.

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. 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 compiler doesn't guarantee the sequential execution of basic blocks. Instead, we tag each memory request with a unique integer representing the state of memory at that time, allowing us to use our LSQ in static HLS where the order of basic block execution is not guaranteed. Our tags are similar to the work by Elakhras *et al.* [28] 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 in our 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 ith 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 $stAllocation_i$: The LSQ outputs a sequence of load values, which correspond to the sequence of previously made load requests.

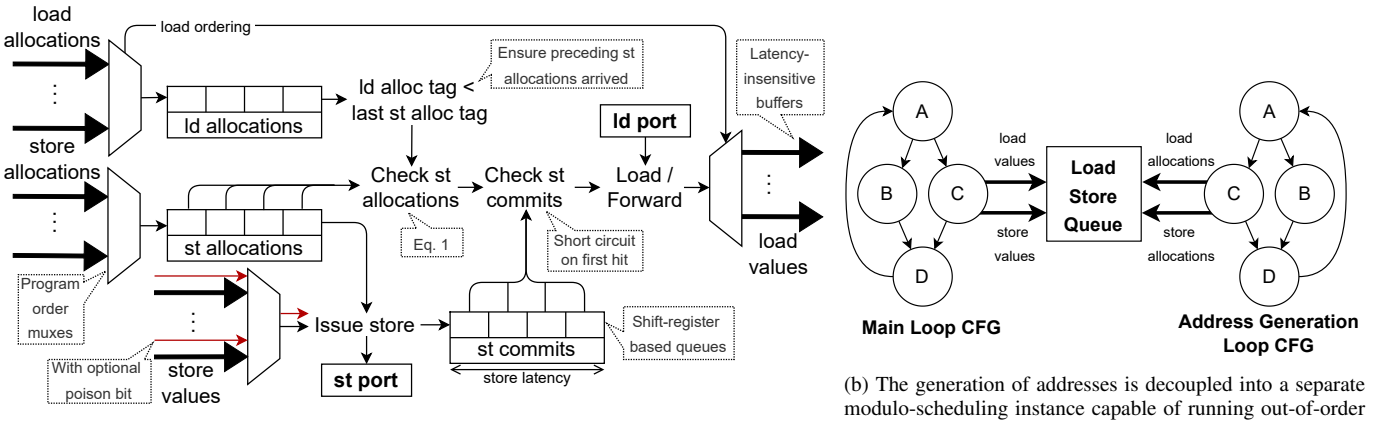
The tag of a load allocation indicates which memory state is expected by the load; the tag of a store allocation represents the new memory state after the store. Given any pair of $ldAllocation_i$ and $stAllocation_k$, if the two conditions hold:

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

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

IV. Load-Store Queue Design

We now present the design of our load-store queue (LSQ). We describe how load and store allocations are handled, and how our design can support speculative address allocations. We also discuss how our design scales to multi-cycled and multi-ported memories.



(a) Execution of loads and stores in our load-store queue design.

(b) The generation of addresses is decoupled into a separate modulo-scheduling instance capable of running out-of-order w.r.t. the actual memory operations.

Fig. 2. Our load-store queue design (a), and how it is used to enable out-of-order dynamically scheduled loads in static HLS (b).

A. Load and Store Execution

An overview of our LSQ design is shown in fig. 2a. Queues are implemented as shift registers in FIFO order. Load and store allocations are shifted through the entire allocation queue before being processed. The store queue is broken up into two separate queues: one for allocations and one for commits. The commit queue holds stores from for the duration between store execution and commit – its size is equal to the maximum store latency and is shifted on every cycle.

Load execution: Our LSQ accepts one load allocation per cycle for every available load port to memory. All load requests proceed in parallel. If there are more unique load allocation sequences than load ports to memory, then the sequences are multiplexed according to program order. Once a load allocation reaches the end of the load allocation queue, we check all preceding store allocations in program order for conflicts using eq. 1. If not all previous store allocations in program order have arrived, then we wait – this check amounts to comparing the load allocation tag to the tag of the last accepted store allocation. If there are no conflicts within the store allocation queue, then we check the store commit queue next. If there is a conflict here, then we simply forward the value from the store commit. Selecting the youngest such value in case of multiple hits in the commit queue is trivial because of the FIFO order in the commit queue. If there is no hit in the commit queue, we can safely load the value from memory. The loaded/forwarded value is returned to the datapath using a non-blocking latency-insensitive channel.

Store execution: Multiple store allocation 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. Once the store allocation reaches the end of its allocation queue, it waits for its corresponding store value to arrive. Multiple store value sequences are multiplexed in the same order as store requests. On the arrival of the awaited store value, a store is immediately issued and the store allocation moves to the store commit queue together with the store value. The store commit queue holds on to the address, tag, and store value until the store is committed to memory. Our LSQ can support *speculative store allocations* by extending each store value with a poison bit that, when set, would cause the store allocation to be discarded without issuing a store.

B. Multi-Port and Multi-Cycle Memory

Our LSQ design extends to multi-ported and multi-cycled memories, e.g. DRAM. When protecting DRAM, our LSQ will have as many load allocation queues as requested by the compiler (typically as many loads can be issued in the same cycle). We do not support reusing load values from other load queues. There will still be only one store port to simplify the handling of write-after-write hazards.

To support multi-cycle memory we grow the size of 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 latency.

V. Compiler Integration

In this section, we detail how an HLS compiler can use our LSQ to enable dynamically scheduled loads in static HLS. We show how parts of the LSQ can be specialized based on the target code, and how the target code can be transformed to enable speculative address allocations to the LSQ.

A. Dynamically Scheduled Memory in Static HLS

We follow the method presented by Szafarczyk *et. al.* to enable dynamically scheduled loads in static HLS [19], using existing compiler analysis to determine which base memory addresses require an LSQ [29]. Each selected base address will use its own LSQ. All memory operations using the base address will be transformed into read/writes from/to latency-insensitive channels connected to our LSQ. The throughput of circuits using an LSQ depends on the number of addresses

that can be disambiguated ahead of their actual memory operation execution. In dataflow circuits, the generation of memory addresses is naturally decoupled from the memory operation and can run ahead. To achieve the same effect in static HLS, Szafarczyk *et al.* suggest decoupling the generation of memory addresses from the main kernel datapath into a separate modulo-scheduling instance, similar to the principle in decoupled access/execute architectures [30] or the decoupled supply-compute approach used in compilers to attack memory latency bottlenecks [31], [32]. Fig. 2b illustrates the resulting communication pattern. The address-generating loop will only contain basic blocks and instructions needed to generate the addresses (these can be obtained using the DDG).

Our LSQ design uses integer tags to recover the order of memory operations. Each address-generating unit has a tag corresponding to a single LSQ. The tag will be initially set to zero and is carried through the pipeline. Store allocations increment the tag before using it; load allocations use the tag directly. The tags create 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.

B. Loss of Decoupling and Speculative Allocations

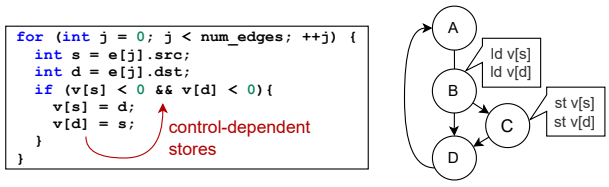
In some cases, the decoupling of address generation cannot result in the run-ahead of address allocations. Such situations, called "loss of decoupling" [31], arise when the address generation for a given base address depends on a value loaded from the same base address. Formally, given a set of address-generating 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.

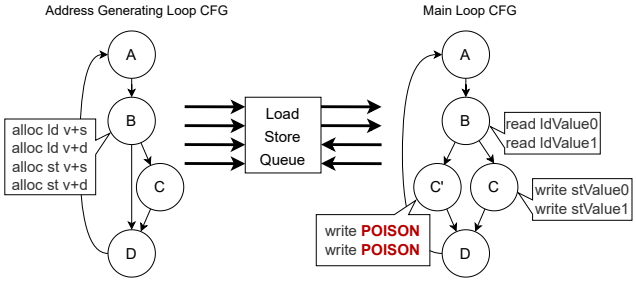
We do not perform address decoupling in such cases, because the address allocations and their memory operations need to be in effect synchronized. This is not a drawback of using static HLS since a fully dynamically scheduled circuit would also have the two sequences synchronized.

We treat control dependencies as a special case. 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. This is the case for the code in fig. 3a. Here, the execution of the stores to `v` is control dependent on the if-condition which uses values loaded from `v`. Under the execution semantics of both static and dynamic HLS, there is no possibility for out-of-order address allocations. In the next paragraphs, we introduce the concept of *speculative address allocations* to relax this restriction.

Consider the code in fig. 3a again. Although the store execution is control-dependent, the store addresses have no data dependency on values loaded from `v`. The address-generating instructions can run ahead in an address-generating loop by hoisting them out of the if condition, as illustrated in fig. 3b. Dually, in the main loop CFG, whenever speculative store address allocations are not needed (basic block C' in



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

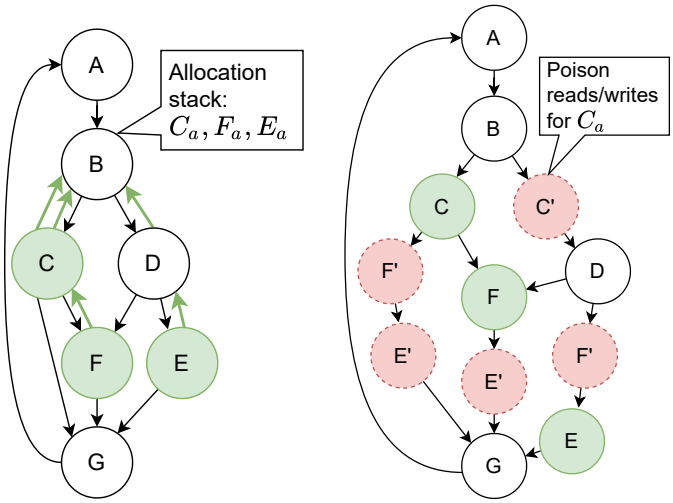


(b) Speculative address allocations (left), coupled with poisoned store values on misspeculation (right).

Fig. 3. Speculative store address allocations in the maximal matching code.

fig. 3b), we set a poison bit in the store value supplied to the LSQ that will cause the speculative address allocation to be discarded (sec. IV-A). As a result, we can achieve a higher degree of out-of-order loads on these types of codes than previous address-based memory disambiguation works [23], without having to suffer the cost of expensive misspeculation replays common in value-based approaches [21].

We now generalize the above transformation to arbitrary reducible loop CFGs. The key question is how to preserve the relative order between speculative address allocations made in the address-generating loop and the poison read/writes in the main computation loop. We define a "special control-dependency" to mean a control-dependency whose branch instruction uses values loaded from a given base address thus causing loss of decoupling. 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. Let B_a be the set of all address allocations for a given block $B \in \mathbb{B}$. In the address-generating loop, we iteratively move up the address allocations B_a for every $B \in \mathbb{B}$ to 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. Take the CFG in fig. 4a as an example, with blocks C, F, and E containing LSQ address allocations. There will be two iterations of hoisting on this CFG. On the first iteration, F_a moves to C, E_a moves to D, and C_a moves to B. On the



(a) Iterative hoisting of address allocations (green blocks).

(b) Insertion of poisonous read/writes to remove misspeculated allocations.

Fig. 4. A visualization of speculative LSQ address allocations in an address generating loop (left), and poisonous read/writes inserted in the main loop CFG to deallocate misspeculated addresses.

second iterations, F_a (now in block C) moves to B, and E_a (now in block D) moves to B.

The second part of the transformation inserts poison read/writes in the main loop CFG that will cause the LSQ to deallocate a misspeculated address allocation. Alg. 1. details the steps needed to achieve this. It takes the basic block B_{spec} with speculative address allocations as input, together with the stack L of speculative address allocations hoisted to B_{spec}. For each block in the main loop CFG that is dominated by B_{spec}, we check if a given speculative address allocation B_a in the stack of B_{spec} is no longer possible at this point in the CFG and if it was not used or poisoned already on the currently evaluated CFG path. If B_a was neither used nor poisoned on the current path ($Cond_1$), and there is no possibility to encounter block B on the current path anymore ($Cond_2$), then we insert a new block at that point in the CFG that contains poisonous read/writes to the LSQ (block B'). A poisonous read simply discards the loaded value; a poisonous store supplies a poison bit that is recognized by our LSQ design (sec. IV). Fig. 4b shows how poisonous basic blocks would be inserted given the address generation loop from fig. 4a. The transformation has practically no misspeculation overhead. Any superfluous basic blocks can be removed using existing CFG simplification algorithms. The size of the store allocation queue might have to be increased to still guarantee perfect throughput, but this is not a requirement (sec. V-C).

The number of required poison read/writes can be decreased if two speculative address allocations in symmetrical branches use the same address and the same memory operation. At each control-dependency source block, we check allocations hoisted from the left and right sub-graph. Each pair of allocations that has the same memory operation and the same address expression is merged into one speculative allocation.

Algorithm 1 Insertion Of Poison Basic Blocks

Input: loop CFG; basic block B_{spec}; allocation stack L
for $B_{pred}, B_{succ} \in$ cfg_traversal(B_{spec}) **do**
  for $B \in L$ **do**
    if $B = B_{succ}$ **then break**
    $Cond_1 \leftarrow$ no $B_{spec} \rightarrow B_{pred}$ path containing B or B'
    $Cond_2 \leftarrow$ no $B_{succ} \rightarrow loopLatch$ path containing B
    if $Cond_1$ **and** $Cond_2$ **then**
      create poison block B' between $B_{pred} \rightarrow B_{succ}$

Theorem. Alg. 1 transforms the main loop CFG such that on every $B_{spec} \rightarrow loopLatch$ path the speculated address allocations are used or poisoned in the order in which they were allocated in the address-generating CFG.

Proof. The correctness of alg. 1 follows from its construction. Alg. 1 visits every block dominated by B_{spec} in control-flow order. At every such block, it goes over all blocks in $B \in L$ in their allocation order, deciding if B' should be inserted at that point in the CFG. $Cond_1$ guarantees that on every path there will be only one use or one poisoning of a given speculation, but never both. Thus, the speculated address allocations will be used or poisoned in the allocation order on every path through the CFG. ■

C. 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 is equal to the maximum dependence distance between a load and a store – for most codes, this will be equal to the recurrence constrained II discussed in the introduction. The optimal size will increase if there are multiple stores before a given load in program order. All of the above information is static, allowing us to find an optimal store allocation queue size at compile time:

$$\frac{maxLoadToStoreDelay}{targetII} \times numStoresBeforeLoad$$

VI. Evaluation

In this section, we evaluate our work against the commercial Intel HLS compiler and against a dynamically scheduled academic HLS compiler that uses a state-of-the-art LSQ. We also evaluate how our LSQ design scales to DRAM.

A. Methodology

We integrated our compiler passes and LSQ with the Intel SYCL HLS compiler and have made our implementation publicly available[1]. We evaluate our work against the dynamic

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

TABLE I
A COMPARISON OF OUR WORK AGAINST VIVADO, DYNAMATIC [28], AND INTEL HLS. ALL CODES USE ON-CHIP BRAM.

Benchmark	Cycles (thousands)				Freq. (MHz)				Execution time (μs)						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**	5.6	3–6	**0.55–1.08**	129	5582	**43.3**	407	1535	**3.8**
getTanh	68	2.5–79	56.2	1.1–57	266	89	377	261	256	28.1–888	**0.11–3.47**	149	4.3–219	**0.03–1.47**	572	22399	**39.2**	433	6490	**15**
getTanhDouble	14	1–19	13.2	1.1–15.7	304	96	330	297	46.1	10.7–198	**0.23–4.3**	39.9	3.9–53	**0.1–1.33**	245	22103	**90.2**	616	3145	**5.1**
vecTrans	30	1.5–31	30.1	1.1–32	304	97	365	291	98.7	15.9–320	**0.16–3.24**	82.6	3.8–110	**0.05–1.33**	125	22997	**184**	412	4997	**12.1**
spmv	2.3	0.8–2.7	3.6	0.8–2.8	263	152	317	288	8.7	5.2–17.6	**0.6–2.02**	11.3	2.9–9.7	**0.26–0.85**	494	5628	**11.4**	1028	7730	**7.5**
chaosNCG	72	37–74	74.3	2.1–76	308	155	335	237	234	239–477	**1.02–2.04**	222	8.9–321	**0.04–1.45**	779	2017	**2.6**	1042	16447	**15.8**
BNN	20	15–30	20.7	10.5–21.6	258	116	362	293	77.5	129–259	**1.67–3.34**	57.3	35.8–73.8	**0.62–1.29**	1214	7466	**6.2**	998	4846	**4.9**
histogramIf	2	5–6	2.1	1-2	388	117	379	338	5.15	42.7-51.3	**8.29–8.3**	5.5	3–6	**0.54–1.08**	155	5395	**34.8**	496	1576	**3.2**
matching	6	6–8	7.1	2.1–8.1	404	110	234	273	14.9	54.6–72.7	**3.67–4.9**	30.5	7.7–29.7	**0.25–0.97**	141	3778	**26.8**	1901	4542	**2.4**
floydWarshall	6.2	7–11	6.3	3.1–3.2	366	90	229	225	16.9	77.8–122	**4.59–7.2**	27.3	13.8–14.2	**0.5–0.52**	255	2226	**8.7**	729	4505	**6.2**
Harmonic mean		**0.14–1.4**		**0.07–0.92**		**0.35**		**0.86**			**0.41–3.53**			**0.1–1.04**			**11.3**			**4.9**

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

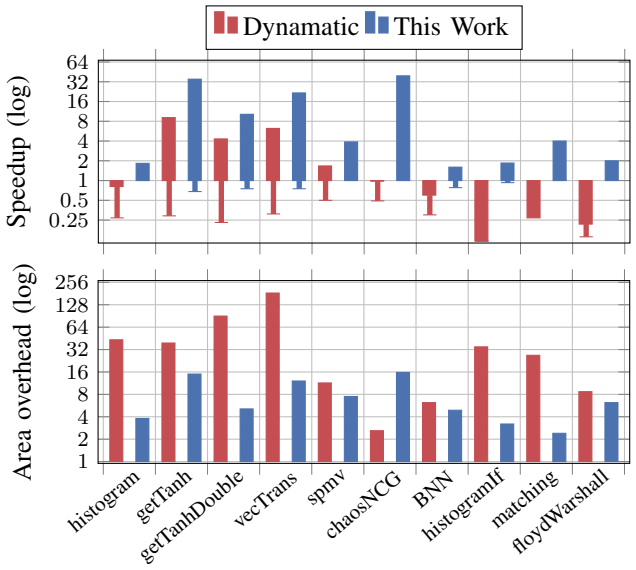


Fig. 5. Speedup and area overhead of our work and Dynamatic [28] compared to their static HLS baselines (Intel HLS and Vivado, respectively). The range bars represent the speedup range, with a value below 1 indicating a slowdown.

HLS tool Dynamatic using a research artifact from their most recent paper [28]. We do not evaluate against DASS [10] since it uses the same LSQ as Dynamatic. Dynamatic uses Vivado for synthesis, while we use Intel tools. This makes a direct comparison in absolute terms difficult. Instead, we compare the *normalized* speedup 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. When comparing against Dynamatic, we only consider on-chip memory (we were unable to use off-chip memory in Dynamatic).

We applied our approach to ten benchmarks with data hazards used in previous work [8], [10], [33]. We plan to make the benchmark codes and results for all approaches available as a public artifact upon publication [34]. 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 speculated 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. 3a (loop II=7).
10) *floydMarshall* finds shortest paths in a weighted digraph (loop IIs=1,1,6).

We report worst- and best-case performance, which depends on the data distribution. We choose our store allocation queue sizes according sec. V-C. For Dynamatic, we choose the smallest queue size that enables perfect pipelining in the case of no data hazards, following their approach [25].

B. BRAM Results

Fig. 5 shows that our approach achieves a higher speedup than Dynamatic when comparing each tool to their respective static HLS baseline, and we do so at a lower area overhead. On most codes, the higher speedup is due to the higher frequency achievable by our shift-register-based LSQ, compared to the CAM-based LSQ used in Dynamatic. Tab. I shows detailed benchmark results. On average, designs with our LSQ achieve 86% 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

TABLE II
SCALABILITY OF OUR STORE ALLOCATION QUEUE COMPARED TO THE STORE QUEUE IN DYNAMATIC [28] ON THE HISTOGRAM BENCHMARK.

Queue Size	Freq (MHz)				Area (Slices / ALMs)			
	Dyn	×	Ours	×	Dyn	×	Ours	×
No LSQ	379	1	379	1	129	1	407	1
2	173	0.46	331	0.87	409	3.2	1416	3.5
4	178	0.47	337	0.89	684	5.3	1535	3.8
8	163	0.43	314	0.83	1554	12	1799	4.4
16	155	0.41	315	0.83	5582	43.3	2157	5.3
32	92	0.24	300	0.79	22580	175	3162	7.8
64	-	-	252	0.66	-	-	4269	10.5
128	-	-	224	0.59	-	-	7419	18.2
256	-	-	194	0.51	-	-	17228	42.3

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 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. In contrast, our approach has on average no overhead in execution time compared to its Intel HLS baseline, even when the data distribution favors static scheduling. The last three codes benefit from our speculative address allocation scheme, allowing for non-trivial speedups compared to static HLS, where the Dynamatic LSQ is not able to increase throughput at all resulting in an effective execution time slowdown.

Our LSQ has a lower area overhead than Dynamatic. For example, Dynamatic uses a queue size of 32 for *getTanhDouble* resulting in an area overhead of 90×. We use the same size of 32 for the store allocation queue, but suffer an area overhead of only 5×. On average, our LSQ results an 4.9× area overhead compared to 11.3 for Dynamatic, and that is despite the fact that for several codes we use a larger queue size than Dynamatic. Our largest overhead of 15.8× is on *chaosNCG* which uses a store allocation queue with 78 entries.

Store queue size scalability: Some benchmarks require a large out-of-order address allocation window for perfect pipelining. We could not synthesize designs with a Dynamatic LSQ of more than 32 store entries because of resource constraints. As a result, some benchmarks that require a large out-of-order address window were not able to achieve perfect throughput when using the Dynamatic LSQ (e.g. *getTanh*). CAM based LSQs, including the Dynamatic LSQ, are notorious for their poor scalability [20], [25]. Our shift-register-based queues scale better, allowing for hundreds of entries. 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 – the potential throughput increase from using dynamic scheduling is higher for larger store allocation queues (sec. V-C).

C. DRAM Results

Tab. III shows the speedups over static 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 DDR3 DRAM. On average, using our LSQ results in an 4.5–11× speedup compared to static HLS. There are a number of interesting points to make when comparing the performance of our LSQ on BRAM and DRAM. Firstly, and most interestingly, the increased size of the store commit queue needed to cover the maximum store latency to DRAM has a cache-like effect. A dependent load can use the value from the commit queue, rather than waiting for a load from memory. As a result, our LSQ still offers a significant speedup even if most of the iterations have a true data hazard. Secondly, using DRAM rather than BRAM results in a lower achievable circuit frequency for the evaluated benchmarks. This in itself is not surprising, and has been noted by previous authors [35], [36], but it means that the critical path overhead of our LSQ is less noticeable. Similarly, codes using DRAM use more resources, making the resource usage of our LSQ less noticeable (2.2× vs. 4.9×).

TABLE III
PERFORMANCE OF OUR LSQ ON CODES USING DRAM.

Benchmark	Exec. Time (μs)			Freq. (MHz)			Area (ALMs)		
	I	O	O/I	I	O	O/I	I	O	O/I
histogram	397	35–63.5	0.09–0.16	273	267	0.98	6001	10895	1.8
getTanh	587	41.8–144	0.07–0.25	281	233	0.83	11384	22106	1.9
getTanhDouble	425	39.9–128	0.09–0.3	281	241	0.86	8208	13023	1.6
vecTrans	459	43.3–199	0.09–0.43	305	221	0.72	6229	11106	1.8
spmv	179	16–32.6	0.09–0.18	287	282	0.98	3654	10307	2.8
chaosNCG	746	61.8–429	0.08–0.58	270	191	0.71	7111	26076	3.7
BNN	463	42.3–64.1	0.09–0.14	264	268	1.02	4824	10401	2.2
histogramIf	396	35.9–67.8	0.09–0.17	274	267	0.97	6029	13191	2.2
matching	515	56.1–165	0.11–0.32	289	229	0.79	3799	10450	2.8
floydWarshall	381	57.6–58.4	0.15–0.15	257	243	0.95	8911	19248	2.2
Harmonic mean			**0.09–0.22**			**0.87**			**2.2**

I – Intel HLS O – Our work

VII. CONCLUSION

We have presented a novel 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. Our design achieves a higher frequency compared to previous LSQs used in HLS, resulting in an average speedup of up 11× compared to static HLS and up to 4.5× compared to dynamic HLS. Our design is scalable, supporting both low latency on-chip and long variable, latency off-chip memories, and allowing store queues with hundreds of entries. Finally, we have presented how the novel concept of speculative LSQ address allocations can enable dynamically scheduled loads on more codes than previous work at no added cost.

ACKNOWLEDGMENT

For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

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] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004.

[18] 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

[19] 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.

[20] 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.

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

[22] 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

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

[24] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proc. IEEE*, vol. 83, 1995.

[25] 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.

[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] 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.

[28] 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

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

[30] 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.

[31] 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

[32] 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.

[33] J. Cheng, "JianyiCheng: HLS_Benchmarks_First_Release," Dec. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3561115

[34] Anonymized, "Artificat to follow," 2023.

[35] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, "Rapidstream: Parallel physical implementation of fpga hls designs," 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. 1–12. [Online]. Available: https://doi.org/10.1145/3490422.3502361

[36] C.-J. Johnsen, T. D. Matteis, T. Ben-Nun, J. de Fine Licht, and T. Hoefler, "Temporal Vectorization: A Compiler Approach to Automatic Multi-Pumping," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 10 2022.