An Out-of-Order Load-Store Queue for Spatial Computing

LANA JOSIPOVIC, École Polytechnique Fédérale de Lausanne
PHILIP BRISK, University of California, Riverside
PAOLO IENNE, École Polytechnique Fédérale de Lausanne

The efficiency of spatial computing depends on the ability to achieve maximal parallelism. This necessitates memory interfaces that can correctly handle memory accesses that arrive in arbitrary order while still respecting data dependencies and ensuring appropriate ordering for semantic correctness. However, a typical memory interface for out-of-order processors (i.e., a load-store queue) cannot immediately meet these requirements: a different allocation policy is needed to achieve out-of-order execution in spatial systems that naturally omit the notion of sequential program order, a fundamental piece of information for correct execution. We show a novel and practical way to organize the allocation for an out-of-order load-store queue for spatial computing. The main idea is to dynamically allocate groups of memory accesses (depending on the dynamic behavior of the application), where the access order within the group is statically predetermined (for instance by a high-level synthesis tool). We detail the construction of our load-store queue and demonstrate on a few practical cases its advantages over standard accelerator-memory interfaces.

CCS Concepts: • **Computer systems organization** → *Processors and memory architectures*; • **Hardware** → *High-level and register-transfer level synthesis*; *Hardware accelerators*;

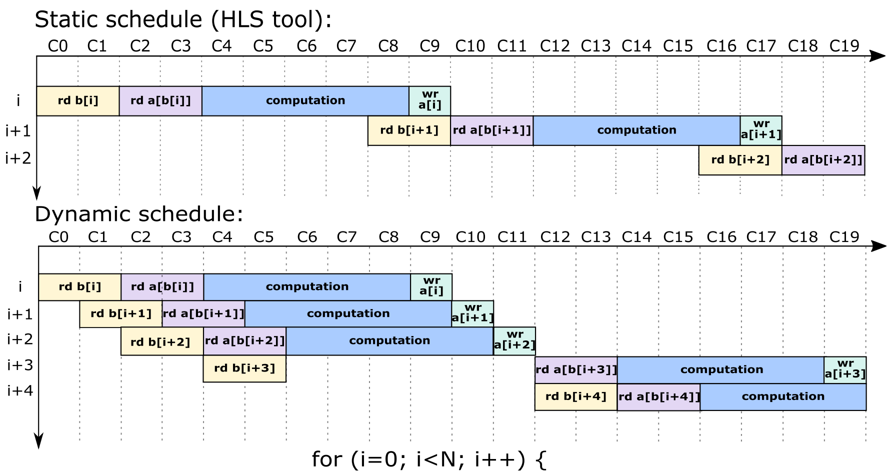
Additional Key Words and Phrases: Load-store queue, spatial computing, allocation, dynamic scheduling

ACM Reference format:
Lana Josipovic, Philip Brisk, and Paolo Ienne. 2017. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 125 (September 2017), 19 pages.
https://doi.org/10.1145/3126525

1 INTRODUCTION

With the proliferation of FPGAs into data centers a clean need has emerged to accelerate applications with irregular memory access patterns. High-level synthesis (HLS) tools, which have matured and achieved commercial viability in recent years, are unable to meet this need due to their reliance on static scheduling. In applications such as graph processing, data analytics, and sparse linear algebra, among others, static memory access disambiguation is impossible. Thus, static HLS tools, which can produce high-throughput pipelined designs in cases where memory accesses *are* provably independent, must make pessimistic assumptions, yielding inferior schedules and lower performance, even when the occurrence of actual dependencies is quite rare.

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: L. Josipovic, École Polytechnique Fédérale de Lausanne, Switzerland; P. Brisk, University of California, Riverside, US; P. Ienne, École Polytechnique Fédérale de Lausanne, Switzerland; emails: lana.josipovic@epfl.ch, philip@cs.ucr.edu, paolo.ienne@epfl.ch.



```
for (i=0; i<N; i++) {
      a[i] = const * a[b[i]];
}
```

Fig. 1. A possible schedule created by a high-level synthesis tool unable to disambiguate dependencies, compared to a dynamic schedule possible with a dataflow approach. The HLS tool will conservatively assume a dependency between every loop iteration, whereas the dynamic design stalls only in the presence of an actual dependency (in the figure, this is the case between iterations $i + 2$ and $i + 3$).

A simple but not unrealistic example is shown in Figure 1: a statically scheduled accelerator would have to delay reading `a[b[i+1]]` until just after the write to `a[i]` of the previous iteration, conservatively assuming a dependence between the two accesses; on the other hand, a dynamically scheduled accelerator could potentially start a new iteration every cycle (in the absence of an address collision) and gain, in this example, up to a factor 8 in performance.

Previous works have explored circuit design techniques that can effectively implement dynamic schedules [5–8, 11, 12, 14, 20]. These circuits—often referred to as *dataflow*, *latency-insensitive*, or *elastic*—could be deployed in a spatial computing environment but would manifest their superiority only with a memory interface that analyzes data dependencies, reorders memory accesses, and stalls in the presence of effective data hazards. Although such behavior has been exploited in out-of-order processors for decades, there has been little effort to create generic accelerator-memory interfaces supporting out-of-order execution. The reason perhaps lies in a fundamental difference between the two systems: In a processor, the notions of fetching and decoding instructions immediately and self-evidently convey the *correct* (or, more precisely, *a* correct) sequential order of requests at the memory interface. In contrast, spatial circuits lack such notions and, in the construction of a dataflow-like accelerator, the information of the original sequential program order is lost unless explicitly maintained in an alternative manner. This makes the design of an out-of-order memory interface more challenging.

In this work, we present an out-of-order load-store queue (LSQ) as an efficient interface between a hardware accelerator and memory. We detail the construction of the LSQ, while emphasizing a novel allocation for spatial computing platforms, which differentiates our LSQ from those found in standard processors.

2 INADEQUACY OF PROCESSOR LOAD-STORE QUEUES

Consider a nonspeculative out-of-order processor with a traditional *load-store queue* (LSQ), similar to the one shown in Figure 2, at the memory interface. The LSQ entry depicted here is generic

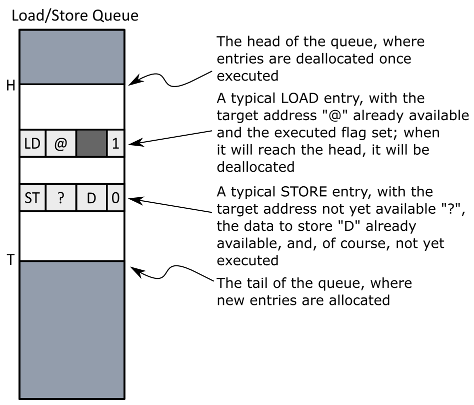

Load/Store Queue
H
T
LD @ 1
ST ? D 0
The head of the queue, where entries are deallocated once executed
A typical LOAD entry, with the target address "@" already available and the executed flag set; when it will reach the head, it will be deallocated
A typical STORE entry, with the target address not yet available "?", the data to store "D" already available, and, of course, not yet executed
The tail of the queue, where new entries are allocated

Fig. 2. A typical processor LSQ with head and tail pointers and two sample entries. In general, entries contain at least these elements: the operation type (“LD” or “ST”), the target address “@”, the data to be stored “D”, if appropriate, and a flag to signal completion. All addresses and data are usually not present at the moment of allocation (indicated with “?” in the figures).

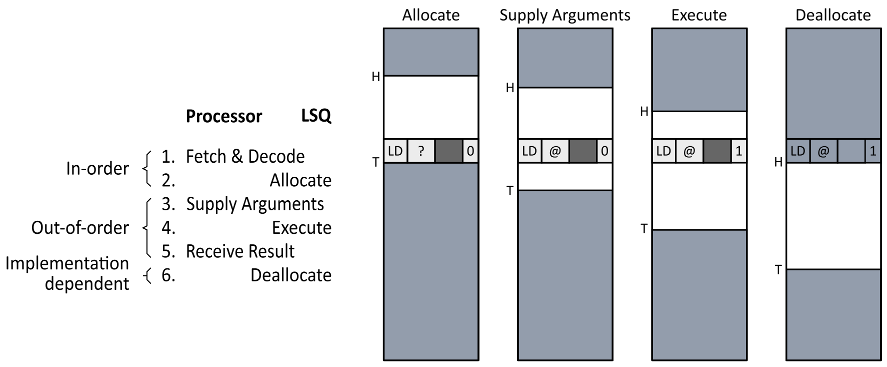


Fig. 3. The basic operation of an LSQ in an out-of-order processor. The allocation of entries in the queue must happen in sequential program order because the position in the queue is used to determine whether an access is safe to execute. Since, in a processor, this happens at *Decode* time, intrinsically in-order, everything is fine; unfortunately, spatial accelerators have no equivalent phase to *Decode*.

and contains at least four fields: (1) An opcode indicating whether the operation is a load or a store; (2) a memory address to access; (3) the data to be written (only used by store instructions); (4) a completion flag initialized to 0 and set to 1 when the LSQ has executed the operation. In this example, the LSQ is organized as a single circular buffer with *head* and *tail* pointers—other organizations are common but differences are irrelevant for this discussion.

Figure 3 depicts the six-step mechanism by which a processor gets load (“LD”) or store (“ST”) instructions executed, emphasizing the interaction with the LSQ. (1) **Instruction Fetch/Decode**: The processor fetches the instruction from the I-cache, decodes it, ascertains that it is a load or a store, and passes it to the LSQ. (2) **Allocate**: The LSQ allocates a new entry for the instruction at the end of the queue and logically connects the present instruction with the new entry. (3) **Supply Arguments**: As the processor pipeline executes other instructions, it will eventually determine the memory address and, for stores, the data value to be written. This information is supplied to the LSQ which writes the actual address and data (if needed) in the reserved entry of the queue.

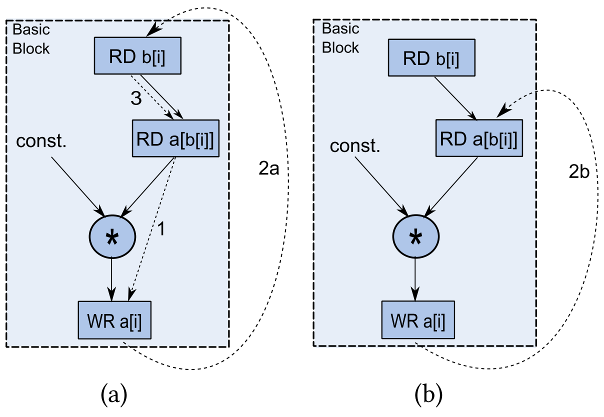


Fig. 4. A partial dataflow graph derived from the code of Figure 1. Nodes are operators (including memory accesses) and solid edges represent data flowing among them. Dashed edges correspond to the ordering of memory accesses in the original program. Static analysis in the compiler may transform or remove some of the edges (as shown in Figure 4b), but, generally, not all of them can be removed.

(4) **Execute**: The LSQ executes the memory operation when *ready*—that is, whenever it is sure that the operation does not depend on any of the accesses coming *before* (i.e., closer to the head) in the queue and not yet executed. The LSQ may be able to execute the operation locally (e.g., a LD operation that reads the value of a ST operation ahead of it in the LSQ), otherwise it transmits the operation to the memory subsystem. The LSQ sets the completion flag and returns the result to the processor. (5) **Receive Result**: The processor receives the result of the memory operation from the LSQ (either a completion signal for store operations or a data value for loads). (6) **Deallocate**: Eventually, the LSQ deallocates the entry for the instruction, freeing it for future use. Steps 1 and 2 (Fetch/Decode and Allocate) **must** occur in the original sequential program order because it is this sequence that implicitly specifies to the LSQ the potential dependencies that need to be respected. In modern superscalar processors, the remaining steps typically occur out-of-order, but the LSQ ensures that the dynamically-arising memory dependencies are correctly sequentialized.

To understand the contribution of this paper, it is important to recognize that the LSQ shown in Figure 3 is built around the notion of a sequential program whose instructions are dynamically fetched and decoded depending on, among other things, the dynamic control flow of the program. The LSQ entries are allocated in the order in which instructions are fetched by the processor; this in-order allocation is critical because, once in the queue, potential dependencies are verified between each entry and all preceding entries in the queue—and, if all dependencies are satisfied, memory operations may then execute out of order.

In a spatial computing system, there is no inherent notion of instructions, program order, PC values, etc. In terms of the described six-stage process, there is no Fetch and no Decode (Step 1); there are no instructions and no I-cache from which to read them. A typical spatial computing system is designed by transforming a dataflow graph into a circuit. In general, the dataflow graph obtained from the compiler will look like the example in Figure 4(a), where solid edges represent actual data dependencies and are transformed into physical wires in the circuit. Dashed edges indicate the sequence of memory operations (and, thus, the potential dependencies) in the original program. The compiler may eliminate or transform some of these edges based on static analysis; however, it cannot safely remove dependency edges unless it is possible to guarantee, statically, that memory dependencies cannot occur. In the example, edges 1 and 3 can be removed because they are implied by actual data dependencies. On the other hand, edges 2a and 3 are unnecessary

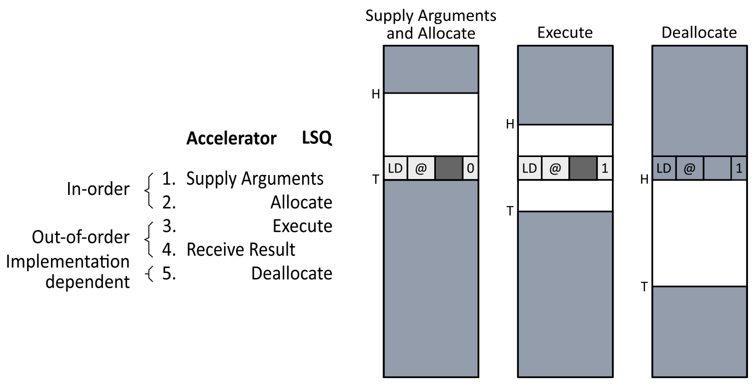


Fig. 5. Allocating entries when the arguments are supplied to the LSQ. This scheme can work correctly only if the circuits incorporate the dashed edges of Figure 4 b: the arguments of a memory access must only reach the LSQ if all memory predecessors have supplied their arguments. The result is a major loss in the usefulness of dynamic dataflow execution and of the LSQ.

because static analysis may determine that the relevant accesses (to `a[]` and to `b[]`) can never conflict. Yet, the potential loop-carried dependence now needs to be represented by edge 2b, resulting in the graph of Figure 4(b). Edge 2b cannot be eliminated because the two connected accesses lead to an incorrect result if reordered when their addresses collide. The problem of correctly optimizing such edges is beyond the scope of our work and has been extensively discussed in literature [3, 4]. The question here is: what to do with them when generating the circuit? Clearly, simply ignoring them would be incorrect, because then memory operations might be triggered in any order and nothing in the circuit would carry the information necessary to respect these dashed edges when addresses collide. Prior research on spatial computing has mentioned the possibilities of employing LSQs to resolve dynamic dependencies (e.g., Huang et al. [10] among others), but has not described when one allocates LSQ entries in a spatial context—or, equivalently, by which mechanism one supplies ordering information to the LSQ. Answering this open question is the foremost contribution of this paper.

3 SUPPLYING A SEQUENTIAL ORDER TO THE LSQ

The allocation of entries to the LSQ must happen in *some* correct sequential order—i.e., an order that respects all dashed edges of Figure 4(b), as this guarantees that all dependencies are satisfied and that memory accesses are correctly executed. We here describe and contrast two design methodologies which ensure such behavior in dataflow-like, dynamically scheduled accelerators.

(1) **Allocating entries to the LSQ as their arguments arrive.** The interaction between the accelerator and the LSQ is shown in Figure 5 and the idea is that allocation happens when the address and/or data for a memory access are known. To ensure correctness, the accelerator must be aware of dependencies and delay the transmission of arguments to the LSQ until preceding accesses (in terms of the dashed edges of Figure 4(b)) have already been allocated. This qualitatively corresponds to the scheduling approach of a classic *high-level synthesis* (HLS) tool—the static schedule ensures that memory accesses are allocated in sequential program order. This nullifies the potential advantages of dynamic scheduling for applications that have ample memory parallelism that cannot be discovered statically. For example, such a tool could not produce the (dynamic) schedule shown at the bottom of Figure 1. This limits the benefits of integrating an LSQ into an accelerator [2, 10].

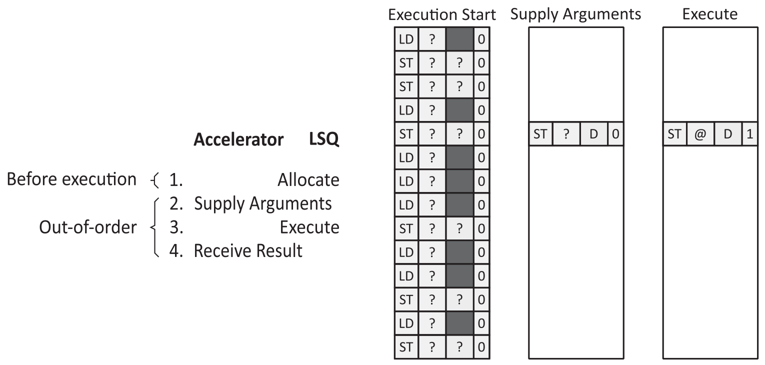


Fig. 6. Allocating entries statically before execution. This scheme depends on the possibility of determining all accesses and their sequential program order at compile time, limiting it to only the most trivial applications. Note that there is no more head and tail of the queue, since the queue contains exactly all memory accesses of the program.

(2) **Static allocation before execution.** As an alternative, the HLS tool could statically allocate all memory accesses through one sequence that respects all memory dependency edges (see Figure 6). Such a system would allow the accelerator to supply operands as soon as they are ready and the LSQ could issue memory accesses as soon as all dependencies are resolved; this would simplify the design of the accelerator and achieve the highest possible memory performance. Although correct in theory, this idea is only feasible for trivial applications that feature statically knowable control flow—which, once again, excludes the example of Figure 1, unless N is a compile-time constant. These scheme is also impractical, as the LSQ would have as many entries as the number of static memory accesses in the application, which would be unrealistically large in all but the most trivial of cases. Others have shown that both the critical path (assuming single-cycle accesses) and resource requirement demands grow as a function of the number of LSQ entries [21]; our implementation results confirm this observation.

Figures 5 and 6 represent these two extremes: The former dynamically allocates LSQ entries when arguments arrive, but must supply them in a statically-determined order which complicates the control mechanism and limits the ability of the LSQ to dynamically resolve dependencies and issue memory accesses in parallel. The design in Figure 6 makes perfect use of the LSQ and allocates each LSQ entry as soon as its arguments are ready, giving the LSQ flexibility in terms of dynamically disambiguating memory accesses; however, the static analysis required of the compiler is unrealistic for non-trivial applications, and LSQ area utilization and performance would degrade significantly as a result of the large number of entries. We desire a policy that combines the flexibility of the former strategy with the effectiveness of the second.

4 AN OUT-OF-ORDER MEMORY INTERFACE FOR SPATIAL COMPUTING

We desire a memory interface that can execute accesses out of order when dependencies allow it. As in ordinary processors, ordering is expressed by organizing the accesses in a linear reference sequence (a queue): essentially, loads can be executed if the addresses of all preceding store entries in the queue are known and do not alias, and vice versa. As indicated in the previous section, the critical difficulty for spatial computing consists in creating such a sequence in the queue, because we cannot rely on the allocation policy of processors (in-order allocation at decode time). We thus propose a design option that is intermediary between the two approaches described in Section 3 and that circumvents their fundamental limitations.

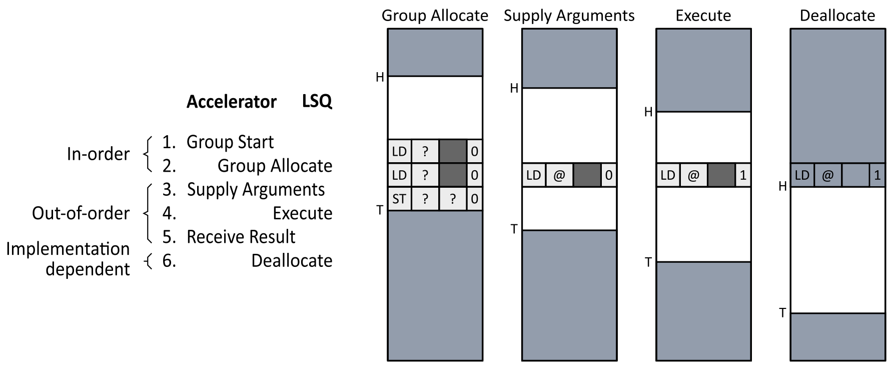


Fig. 7. Allocating entries by groups. Our solution requires the accelerator to announce groups of accesses when they become available. Groups are predefined sequences of accesses which are statically known to a compiler and that can be inserted atomically. Once this is done, our LSQ works essentially like a standard processor LSQ.

Strategy	Pros	Cons
Dynamically allocate entries once address or data are supplied to the LSQ (Figure 5)	- Applicable to any progam	- Accesses must be sequentialized to ensure correctness - LSQ is arguably almost useless
Statically allocate all entries before program execution (Figure 6)	- Fast out-of-order execution	- Applicable only to programs with fully statically determinable accesses (few or none, in practice) - Unfeasible LSQ size in general
Dynamically allocate entries by group (our strategy, Figure 7)	- Applicable to any program - Fast out-of-order execution	

Fig. 8. Comparison of different options for the allocation of entries in an LSQ for spatial computing.

To this end, we introduce the notion of groups: A *group* is a sequence of accesses which cannot be interrupted by a control flow decision (that is, if one access of a group executes, all other accesses belonging to the same group will eventually execute as well). Determining a correct order (i.e., an order that satisfies all the dependence edges) of accesses within a group is trivial using static analysis, as there is no control flow decision that could invalidate it. Our strategy is to dynamically allocate entries for all the memory operations of a group at once as soon as a control flow decision is made (as shown in Figure 7). Once the entries have been allocated, the memory access arguments can arrive out-of-order and will be executed as soon as dependencies can be determined (using the same strategy as a standard processor LSQ and described in Section 2). The table in Figure 8 outlines the pros and cons of the different allocation strategies.

Our group-based allocation principle can be summarized as follows: (1) The order of memory accesses of each group is produced through static analysis and encoded into the LSQ at compile time. (2) The LSQ has as many load/store ports as there are load/store operations in the program. (3) The ports are clustered by groups; every access port must belong to exactly one group. (4) Whenever the accelerator 'activates' a group, all load/store operations belonging to that group are allocated in the LSQ in the sequence that was statically determined for that group. (5) Once a group has been allocated, the LSQ expects each of the corresponding ports to get an access, eventually; dependencies will be resolved based on the order of entries in the LSQ.

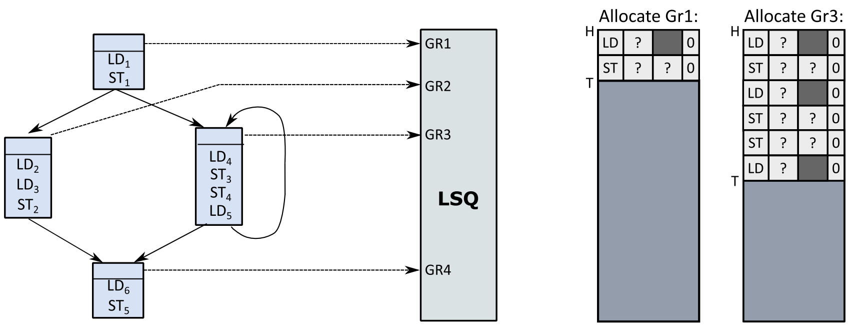


Fig. 9. A program with memory accesses divided into groups which are connected to the LSQ. Each group has a GR signal to indicate its start and to trigger the allocation of its memory accesses into the LSQ. In this case, groups correspond to basic blocks created by HLS tools. The figures on the right show two consecutive allocations (of group 1 and group 3) into the LSQ.

Consider a program with a control flow graph as given in Figure 9 and with memory accesses logically divided into four groups, based on the control flow of the program. The order of accesses within each group can be statically predetermined using compiler analysis (we provide an arbitrary ordering for each group in the figure). Every group is connected to the LSQ with a dedicated GRx signal which indicates to the LSQ the start of the particular group and triggers the allocation of its accesses into the LSQ. The queues on the right of Figure 9 show the allocation of group 1 and group 3 to the LSQ (assuming that a control flow decision determined the execution of group 3 after group 1).

The LSQ proposed here is designed to be independent of the synthesis algorithms that form groups and produce GRx signals. Intuitively, each basic block could correspond to a group (we apply this strategy in the example in Figure 9). The GRx signal could be generated by the HLS compiler and triggered when control passes to the basic block. Alternatively, one could imagine a logical-OR of all ports in a group: as soon as any operand arrives, space is allocated in the LSQ for all accesses.

Our LSQ differs from traditional usage in processors for one key reason: even if an accelerator obtained from a program containing N memory operations does indeed require N memory access ports, those ports seldom need to be on the same LSQ. Only accesses with the potential to conflict demand to share the same LSQ. In the example of Figure 1, if one determines that accesses to `a[]` and to `b[]` cannot conflict, one would need an LSQ with one load and one store port for `a[]` and a simple memory port (without an LSQ) to access `b[]`. Although optimally sizing the LSQs is beyond the scope of this paper, it is clear that using many small queues is beneficial for timing and area compared to a large queue, because both complexity and timing are certainly superlinear in the number of entries.

5 LSQ IMPLEMENTATION

Any LSQ needs to implement the following functionalities: (1) Allocate entries in the queue for new accesses. (2) Enable the access ports and connect them logically to the respective LSQ entries allocated in the previous step so that arguments and results can be dispatched. (3) Accept arguments for the allocated LSQ entries as they arrive out-of-order. (4) Dynamically decide which accesses can be safely executed without violating dependencies in memory. (5) Return as soon as possible available results to the load ports that requested the accesses. And, finally, (6) Deallocate entries in the queue when no longer needed. As pointed out earlier, almost everything related to steps (2) to (6) is identical or very similar to what happens in a traditional processor LSQ, whereas the dynamic behavior of spatial architectures suggests that they will benefit significantly from a rethinking of how function (1) is implemented.

As in any LSQ design, the challenge lies in implementing these functions concurrently with minimal complexity and achieving the lowest possible cycle time. For simplicity, we perform every function in a single cycle: this may result in a higher cycle time but has the significant advantage of naturally making every step atomic (starts on a rising edge and is completed before the next edge), which avoids any difficulty with the concurrence of the operations (a function cannot change the state inconsistently while the others are executing as all start from a consistent state and all produce during a cycle their contribution to a new consistent state). Although a long cycle time could be severely damaging, we will show that even under this constraint the cycle time remains fairly affordable (that is, comparable to the cycle time of simple accelerators). Removing this constraint would result in a more complex design but would not significantly change the architecture and methodology we describe—it is certainly something we will address in the future.

The only aspect where a multicycle nature is essential is the interface to the memory subsystem (caches, etc.): although our current implementation assumes that memory can be read in a single cycle, because this is the case in our system, we will extend our solution to support caches and complex memory hierarchies in future work. It is important to realize that accommodating this aspect of multicycle execution is much easier than the more general multicycle operation discussed above: the execution decision is dependent on the state of the queues and nothing in the system can revert this decision (that is, if a load can be executed before all other pending stores, no further event may invalidate such a decision). Thus, there are no issues of atomicity or concurrent state update here—the fact that the memory hierarchy provides a load result only after a variable number of cycles has virtually no influence on the design and does not present any of the difficulties that other multicycle updates may imply.

In the following sections we discuss the overall structure of the LSQ and detail the six previously-mentioned functions, with particular attention to the allocation by group, which is unique to our architecture.

5.1 The Queues and the Overall Structure

The overall structure of the LSQ is shown in Figure 10. The logic around store and load entries of the queue is quite different, so we chose to implement two separate queues. Both queues have a head and a tail register and contain a power-of-two number of entries.

Each entry of the store queue contains the following fields: (i) store address, (ii) store data, (iii) number of the originating access port, (iv) position in the load queue of the last load preceding this store (indicating which loads need to be checked for conflicts before issuing this store to memory—this mechanism will be discussed in detail in Section 5.2), (v) address validity flag, indicating whether the access port has already supplied the corresponding argument, (vi) data validity flag, indicating if the access port has already supplied the data, (vii) "executed" flag, indicating that the store has been issued to memory.

Each entry of the load queue contains the following fields: (i) load address, (ii) a field for data received from memory, (iii) number of the originating access port, (iv) position in the store queue of the last store preceding this load, (v) address validity flag, indicating if the access port has already supplied the corresponding argument, (vi) data validity flag, indicating whether the data has been received from memory, (vii) "executed" flag, indicating that the load request has been issued to memory, (viii) a flag indicating that the result has been sent back to the corresponding port. Figure 11 visualizes a load queue entry.

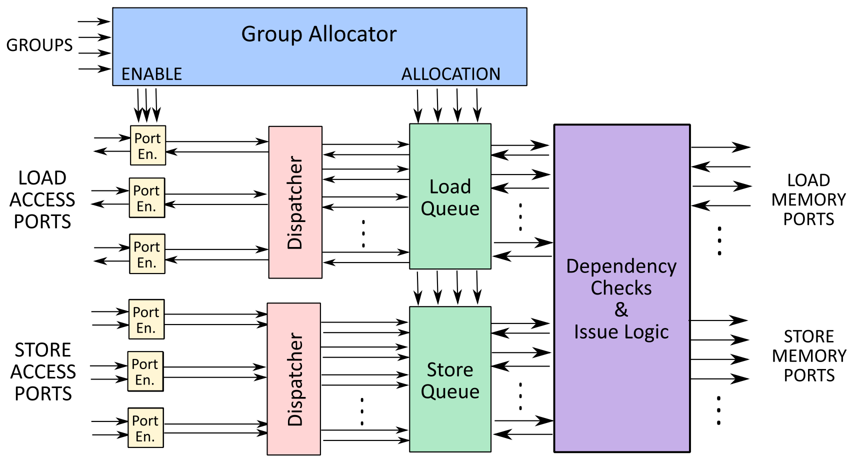


Fig. 10. Overall structure of the LSQ. At the center are separate load and store queues; the group allocator prepares entries in the queues and links them to the access ports, dispatchers connect the entries to the ports; the issue logic checks dependencies and decides which entries are safe to send to the memory subsystem.

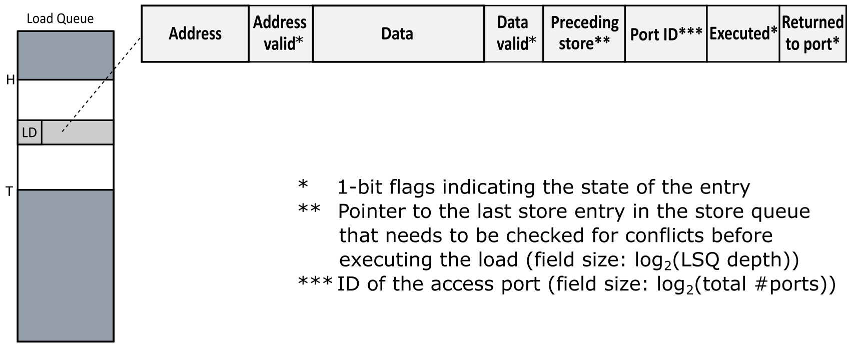

Load Queue
H
LD
T
Address
Address valid*
Data
Data valid*
Preceding store**
Port ID***
Executed*
Returned to port*
* 1-bit flags indicating the state of the entry
** Pointer to the last store entry in the store queue that needs to be checked for conflicts before executing the load (field size: log2(LSQ depth))
*** ID of the access port (field size: log2(total #ports))

Fig. 11. Detailed load queue entry of the LSQ. Apart from the address and data fields, each queue entry contains the information about the port associated to the entry, the last preceding store request in the sequence, and multiple flags describing the state of the entry. The entries of the store queues are identical, apart from the last flag which can be omitted.

Despite the apparent complexity of the LSQ, as shown in Figure 10, concurrency is relatively straightforward since every element of the state of the LSQ is produced independently. For instance, the tails of both queues are only updated by the group allocator when new entries are added to the LSQ. The group allocator is also responsible for filling in the field pointing to the last preceding store/load position for each entry of the load/store queue. The access ports themselves, through the dispatchers, are the only components affecting the argument entries of both lists (memory addresses and store data). The parallel dependency checks and issue logic on the right side of Figure 10 are the only entities that can modify the executed flags and update the head pointers.

The queues are therefore not implemented as centralized monolithic data stores, but are instead distributed throughout the circuit in proximity to the producers and the consumers of the various fields; in practice, connectivity is far less problematic than the layout shown in Figure 10 suggests. For the implementation of the queues, or of any component thereof, we cannot use RAMs. An ASIC LSQ implementation should be approximately the same as that of a processor-based LSQ

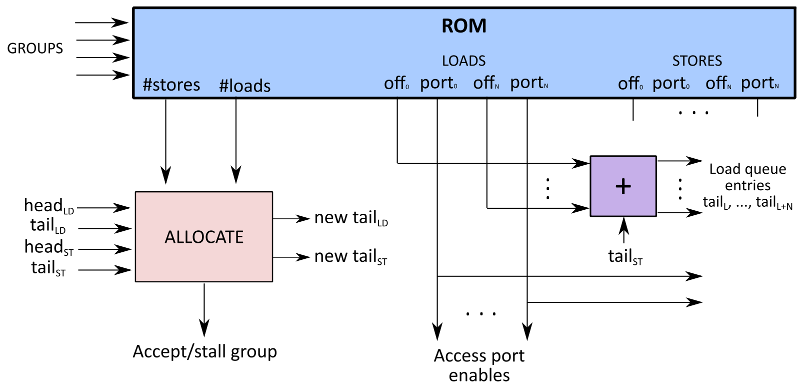


Fig. 12. The group allocator. When a statically-determined sequence of accesses (a group) begins, this unit allocates in parallel all the corresponding entries into each of the queues.

of comparable capacity, as only the allocation policy has changed. An FPGA implementation, in contrast, must implement the LSQ state in programmable logic, as it is not compatible with block RAMs, which may lead to significant but unavoidable inefficiencies.

5.2 Group Allocator

The group allocator (Figure 12) is unique to our memory interface. When a group begins to execute, the first step is to allocate locations in the LSQ for the group's loads and stores, while ensuring that the allocation process respects their sequential program order, which is known statically. This enables the corresponding access ports (on the left of Figure 10) to accept memory requests and allows them to arrive in arbitrary order, while appropriately maintaining a mapping between memory operations and their LSQ entries.

Group allocation requests are sequential by definition, and thus only one new request can arrive at any moment in time. Once a group has been allocated to the queue, its accesses can execute before or in parallel with those in the other groups preceding it in the queue. Each group allocation request addresses a ROM at the core of the group allocator. On a fairly wide data bus, this ROM outputs the following information: (1) number of loads in the group, (2) number of stores in the group, (3) for each load in the reference sequence, the number of stores preceding it in the group (we refer to this value as the offset value of the entry), (4) for each load in the reference sequence, the access port ID, (5) for each store in the reference sequence, the number of loads preceding it in the group (i.e., offset), and (6) for each store in the reference sequence, the access port ID. For instance, group 3 from Figure 9 contains two loads and two stores whose sequential program order is load, store, store, load and these come from access ports LD4, ST3, ST4, LD5; the output of the ROM for this group is 2, 2, 0, 4, 1, 3, 1, 4, 2, 5. The first two values correspond to the numbers of the loads and the stores in the group, respectively, and sets (0,4), (1,3), (1,4), (2,5) represent (offset, port ID) pairs for each of the load and store ports of the group. The group allocator checks using the first two output values from the ROM whether there is enough space in the two queues (in this example, both the load and the store queue need to contain two empty slots). If not, the allocation is deferred until the queues have enough space for the new group and no further allocations are accepted. If the allocation can take place, the group allocator tags each store by adding the number of preceding loads in the group (defined in the ROM) to the value of the load tail prior to allocation. This sum is a pointer to the position that the last preceding load occupies in the load queue (meaning that everything before it in the queue comes earlier in program order and needs to be checked for

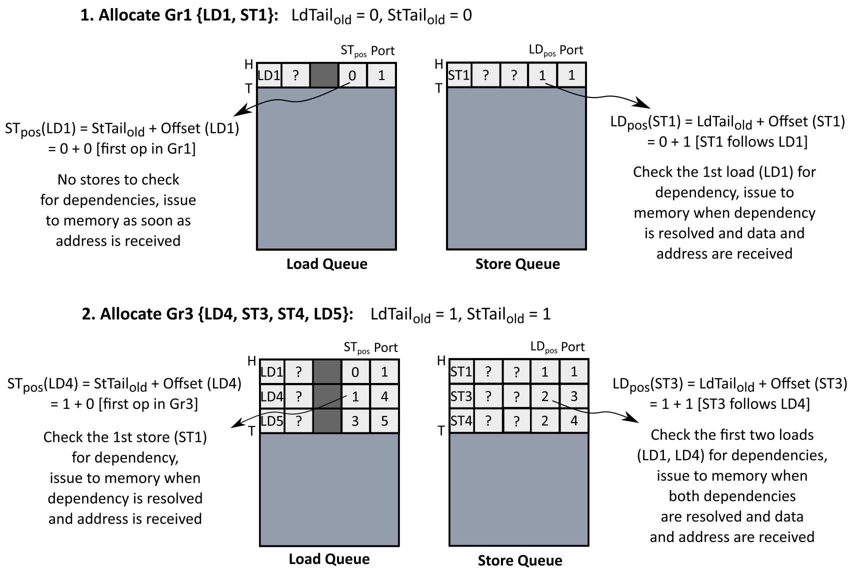

1. Allocate Gr1 {LD1, ST1}: LdTail_old = 0, StTail_old = 0
ST_pos Port
LD_pos Port
LD1 ? 0 1
ST1 ? ? 1 1
ST_pos(LD1) = StTail_old + Offset (LD1) = 0 + 0 [first op in Gr1]
No stores to check for dependencies, issue to memory as soon as address is received
LD_pos(ST1) = LdTail_old + Offset (ST1) = 0 + 1 [ST1 follows LD1]
Check the 1st load (LD1) for dependency, issue to memory when dependency is resolved and data and address are received
Load Queue
Store Queue
2. Allocate Gr3 {LD4, ST3, ST4, LD5}: LdTail_old = 1, StTail_old = 1
LD4 ? 1 4
LD5 ? 3 5
ST3 ? ? 2 3
ST4 ? ? 2 4
ST_pos(LD4) = StTail_old + Offset (LD4) = 1 + 0 [first op in Gr3]
Check the 1st store (ST1) for dependency, issue to memory when dependency is resolved and address is received
LD_pos(ST3) = LdTail_old + Offset (ST3) = 1 + 1 [ST3 follows LD4]
Check the first two loads (LD1, LD4) for dependencies, issue to memory when both dependencies are resolved and data and address are received

Fig. 13. Allocating groups to the LSQ. The figure details the allocation of group 1 and group 3 from Figure 9. The offset values and port IDs are obtained from the ROM. The group allocator calculates the last preceding load/store for each instruction, which the queue will use to correctly perform dependency checks.

conflicts before issuing the store in question). This tag can correspond either to the previous load in the current group or to the last load of the previous group (in case the value provided by the ROM was zero). The tagging is done in parallel for each of the stores (and vice versa for each of the loads). Additionally, the allocator stores the ID of the originating port for each of the entries, which enables inserting values into the correct queue positions, and, for loads, returning the data fetched from memory to the appropriate port. Figure 13 shows the allocation process of group 1 and group 3 from the example in Figure 9, detailing the calculation of the tags of each queue entry.

Clearly, the width of the ROM is determined by the size of the group with the largest number of memory accesses (e.g., group 3 in the case of the program in Figure 9, discussed above). It is worth noticing that, although in principle fairly wide, this ROM remains a small component in practical cases. In the small kernels that we evaluate here, the number of groups is quite small, noting that only accesses that cannot be statically disambiguated must be issued to the same queue. In the histogram kernel reported in Section 6, which is certainly a quite simple but not unrealistic application, the ROM contains only a single word of 6 bits—an extreme but not outright artificial situation.

The critical path of the group allocator is fairly short and is essentially composed of the small ROM, an adder whose number of bits is sufficient to address the elements in the queues (to compute the absolute offset to save in the queues), and a multiplexer to bring the right offset to the corresponding queue entry.

5.3 Access Port Enable and Dispatchers

An access port is enabled by the group allocator once the allocator assigns queue entries for the group that this port belongs to. If the port sends a request prior to the allocation, it will be stalled until the corresponding position in the queue has been allocated. On the other hand, a group may be triggered again before all accesses of previous instances of the same group have received the operands and have executed (think of many iterations of a loop executing concurrently); this implies that ports may be linked to multiple entries of the queue. Therefore, the port has a counter to determine how many arguments it can accept; every time the corresponding group is allocated, the counter is incremented and every time the port receives an argument, the counter is decremented. The port is disabled (that is, it stalls arriving arguments) when the counter is null. Store ports contain separate enable logic for the data and the address, since they might not arrive simultaneously to the LSQ.

All ports search concurrently all elements of the corresponding queue for the earliest entry they are related too. The result of this search is used by the access port both to forward to the queue newly arrived arguments and to pull newly available load results. The prioritization of queue entries related to the same access port acknowledges the fact that accesses on a specific port arrive in order and, most importantly, are returned in order. The critical path of the dispatchers is essentially a comparator for the port numbers, a priority encoder, and either the control path of a large multiplexer (with as many inputs as the elements of each of the queues, for returning load results) or the decoder logic of the same size (for writing arguments into the queue).

5.4 Checking Dependencies and Executing

The rest of the LSQ is practically identical to any processor LSQ: dependencies must be checked and accesses issued.

The load queue checks concurrently all loads, comparing each available address with all store addresses which precede it (and determined thanks to the pointer to the last preceding store that is part of the queue entry). If any of the preceding stores misses the address, the load is left waiting. If all preceding stores have an address and there is no collision (the load address is different from all store addresses), the load can be executed (i.e., sent to the memory subsystem). A priority encoder takes the oldest executable loads and passes as many as possible to the memory subsystem (that is, as many as there exist ports to memory). If the load address equals one or more of the store addresses, and if the latest of the colliding stores has already received the data argument, the store is bypassed and the store data used as result of the load access. The load result (either received from memory or bypassed from the store queue) is entered in the appropriate field of the load queue. As mentioned, load access ports pull concurrently all new results from the load queue (which would make it fairly easy to accommodate variable-latency memories, as discussed previously).

The store queue acts similarly but only checks as many oldest stores (that is at the head of the queue and not yet executed) as there are store ports in the memory subsystem. For each, it compares the address, if available, with all load addresses which precede it (and determined thanks to the pointer to the last preceding load that is part of the queue entry). A store is executed only if (1) the address and the data for the store are known, (2) all preceding stores have executed, (3) the addresses of all preceding loads are known, and (4) there is no collision with any of the previous loads. If any of the tests fail, the store is kept waiting.

Executed accesses at the heads of the queues are simply deallocated. The queues can simultaneously deallocate as many entries as can be sent to the memory subsystem.

6 EXPERIMENTS AND RESULTS

In this section, we discuss the resource and timing characteristics of our load-store queue and evaluate how different queue parameters affect its performance and resource utilization. We demonstrate the benefits that the load-store queue brings by comparing our designs of applications containing irregular loops with that created by a commercial HLS tool on FPGAs. All timing and resource information for our VHDL designs are from the post-routing analysis of Vivado. We provide the resource usage as the number of CLB slices, with the corresponding LUT and FF count.

Table 1. LSQ *clock Period* (CP) in ns and Resource Utilization for Different Numbers of Groups

Groups	**Depth**	**Ports**	**CP**	**Slice**	LUT	FF
1	8	8	5.7	1,190	3,482	1,346
2	8	8	5.9	1,326	4,457	1,354
4	8	8	5.7	1,511	5,268	1,388

Table 2. LSQ *Clock Period* (CP) in ns and Resource Utilization for Different Numbers of Ports

Groups	**Depth**	**Ports**	**CP**	**Slice**	LUT	FF
1	8	2	5.7	1,144	3,200	1,260
1	8	4	5.7	1,228	3,967	1,270
1	8	6	5.7	1,297	4,458	1,341
1	8	8	5.7	1,190	3,482	1,346

6.1 Resource Utilization and Timing Analysis

Our queue designs are generated based on the required parameters (queue sizes, number of groups, number of ports, etc.). We here evaluate and discuss the timing and resource requirements when these parameters change.

Sensitivity to Group Count. We discuss the effect of the number of groups connected to the LSQ on the resources and the timing of the design. Table 1 provides the evaluation for queues with a varying number of groups, while keeping the overall number of load and store ports (distributed equally over the existing groups) and the queue depth constant. The change in the number of groups and the way the ports are organized impacts only the group allocator and has no effect on the rest of the design. Due to the parallel nature of this unit, changing its parameters has barely any influence on the cycle time, which is only slightly affected by the explored modifications. The resource requirements increase only moderately with the number of groups, as the logic requirements for allocating the groups are minimal. This shows that our approach can effectively implement designs with different (larger) numbers of groups, implying the applicability of our solution to a wide range of applications.

Sensitivity to Port Count. In Table 2, we explore the effect of the number of ports on the resource and timing characteristics of our LSQ by comparing designs of equal queue depths and group count but containing different numbers of ports. We choose to implement designs with an equal number of load and store ports (i.e., the 2-port design has one load, and one store port; we add one port of each type in every subsequent design). With the number of ports, the resource utilization of the design naturally increases, without influencing the cycle time. As previously explained, each port adds the following logic to the design: in case of a store port, counters for the data and address, and a dispatcher connecting the port to the queue; in case of a load port, only one counter is needed at the input, but the dispatcher contains additional logic for returning the data fetched from memory to the accelerator. However, it can be noted that the overhead of a single port is minor compared to the overall resources of the LSQ. In some cases, the logic synthesizer can reduce the complexity of the logic around the queues thanks to the characteristics of the specific application it is customized for. This happens, for instance, for the 8-port LSQ in Table 2: In this case, because of the integer ratio between the queue length (8) and the length of the only possible group (4 loads, 4 stores), the LD/ST ports need to be connected only to specific queue entries.

Table 3. LSQ *Clock Period* (CP) in ns and Resource Utilization for Different Queue Depths

Groups	**Depth**	**Ports**	**CP**	**Slice**	LUT	FF
1	2	2	3.5	181	333	312
1	4	2	3.9	353	953	603
1	8	2	5.7	1,144	3,200	1,260
1	16	2	7.2	2,998	9,891	3,524

The logic synthesizer can use this information to reduce the design complexity and the effect is counterintuitive.

Sensitivity to LSQ Depth. We compare now the timing and resource requirements for different queue depths, always with a single group and a fixed number of ports (one load, one store port).

The results in Table 3 show a nonnegligible increase in resources and cycle time. Although our design exhibits ample parallelism and performs most operations concurrently, some functionalities cannot be implemented in constant time—for instance, to bypass data from the store to the load queue, one needs to check the store queue from the head to the tail to find the last conflicting data, and this requires at best $O(\log n)$ time for an n-depth queue. This sensitivity to the number of queue entries is in line with results reported by others in conventional LSQ designs—previous efforts to implement conventional LSQs in FPGAs have exhibited the same trends of resource and clock degradation with queue size [21]. These results motivate us to consider alternative design options in the future—our group allocation policy is generally applicable and can be incorporated into different queue architectures. It should be noted that the contribution of this paper is not improving the design of a conventional LSQ, but to propose a practical and efficient way to adapt conventional LSQs for spatial computing. As we will demonstrate in the following section on real-life examples, we successfully accomplish this task.

6.2 Benchmarks

In this section, we demonstrate the benefits or our LSQ on applications occurring in different domains which exhibit irregular and conditional data-dependencies:

(1) *Weighted Histogram* calculates the histogram of an array of features. In each loop iteration, the value of one of the histogram bins needs to be increased, based on the input data and its corresponding weight. The loop may contain an inter-iteration read-after-write dependency if one of the following iterations needs to read the same histogram bin that a previous iteration is writing into (similar to the example discussed in Figure 1). Although the inter-iteration dependency rarely occurs, the HLS tool cannot statically determine when dependencies are present. Hence, it creates a conservative schedule with an *initiation interval* (II) equal to the number of cycles needed to write into the histogram bin of one iteration before reading a bin in the next iteration.

(2) *Maximal Matching* is a graph algorithm which iterates through the edges of a graph and checks them for matching, i.e., determines if two edges share a common vertex. In case they are determined independent, the algorithm computes the new vertex values and updates the vertices using conditional stores. The kernel contains conditional loop-carried dependencies which exist only when the stores need to be executed. In such cases, the HLS tool conservatively stalls the beginning of the next loop iteration until the stores have been completed.

(3) *Matrix Power* multiplies the elements of a sparse matrix and a vector. In each iteration of a nested loop, a row and a column coordinate are read from memory and the corresponding matrix element is updated. As inter-loop read-after-write dependencies can occur, the HLS tool fully sequentializes every loop iteration.

We manually designed dynamically scheduled hardware implementations of the computational kernels. Our hardware construction is based on a known latency-insensitive protocol [7] and we connect each kernel to our LSQ as the memory interface. We compare it with a statically scheduled design of the same computation, which we generate using Vivado HLS. To provide an accurate and fair comparison of our designs against the one generated by Vivado, we employ the same arithmetic units produced and used by the HLS tool into our dynamic designs. The LSQ is connected to the very same memory subsystem that Vivado creates, thus limiting overall differences to a minimum. We form the groups for our designs corresponding to the basic blocks of the application. In the three explored designs, this strategy results in the following: (1) the Weighted Histogram application has a single group connected to the LSQ containing one load and one store port, (2) Maximal Matching has two groups connected to the LSQ; one of the groups contains two load ports, and the other contains two store ports, (3) Matrix Power contains two load ports and one store port in a group.

Exactly as Vivado HLS does and as explained at the end of Section 4, we connect the ports to different memories or different LSQs if it can be proven that they cannot alias. This is the case for the weight values in Weighted Histogram, for the edges in Maximal Matching, and for the row and column indexes in Matrix Power.

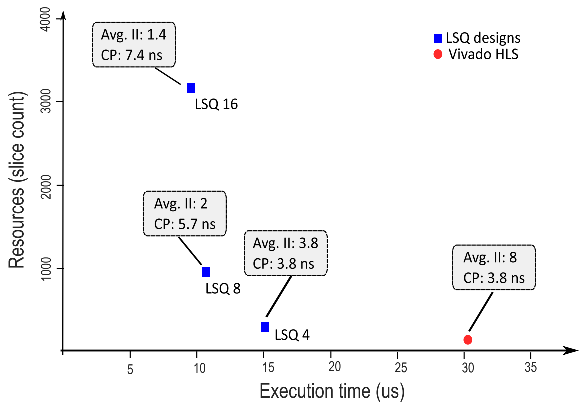
Figure 14 shows the timing and resource utilization of the reference HLS designs together with our dynamically scheduled designs connected to LSQs of different sizes (4, 8, and 16). We calculate the total execution time as the product of the *clock period* (CP), obtained from the Vivado post-routing analysis, and the number of cycles, acquired by simulating the designs in Modelsim.

In all applications, the HLS tool creates the worst-case schedule with conservative *initiation intervals* (II). This is the almost universal static approach we discussed in Section 1. In contrast, our LSQ dynamically resolves memory access dependencies, improving processing efficiency while increasing throughput and lowering execution time. Even with the increased cycle time, particularly for some LSQ sizes, the difference in cycle time is sufficiently small compared to the potential improvement in II. With a still affordable resource cost (i.e., the design with an LSQ of depth 8 occupies only under 5% of a typical FPGA), one can attain almost the maximum parallelism available in the application, and, in many cases, achieve the maximal performance (Maximal Matching, Matrix Power). It is perhaps worth noticing that all discussed applications consists of little more than the memory accesses and thus require very few FPGA *slices*, making our LSQ large in comparison but not necessarily in absolute terms.

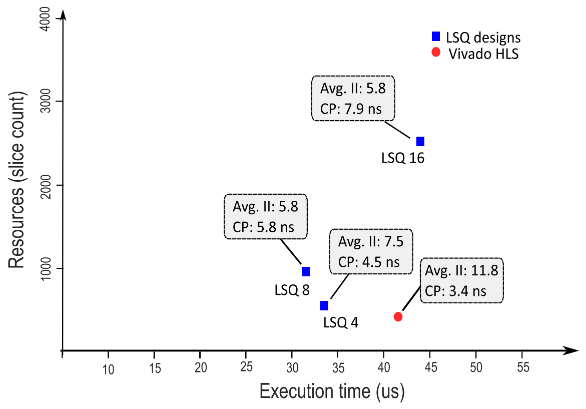
7 RELATED WORK

The processor architecture community has studied LSQs for out-of-order processors for decades and has reported some innovations even in this millennium [15, 17, 18]. Although these previous designs have informed our implementation, they have not been specialized for spatial computing.

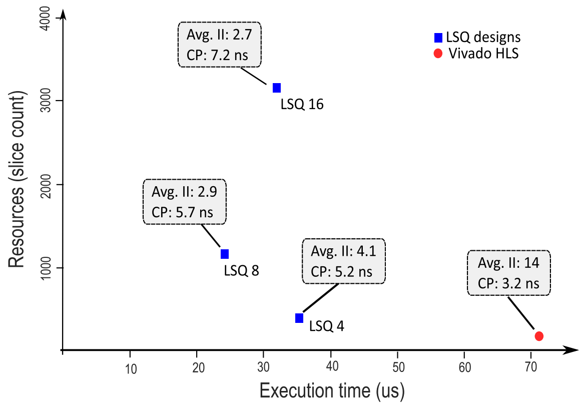
Traditional high-level synthesis systems typically rely on state machine control while leveraging techniques such as scheduling, pipelining, etc., to statically extract instruction level parallelism. Recent advances in HLS have explored methods to overcome the conservatism of static scheduling and to remove the inability of HLS tools to handle dynamic dependencies. Several techniques [1, 14] are based on generating multiple schedules which are dynamically selected during or immediately before runtime, once the values of all parameters are known. They naturally rely on the capabilities of current HLS tools by simply replicating the source code and dynamically selecting which copy of the code needs to be executed. The drawback of these approaches is that they apply to only some very particular cases of dependencies through memory; they are also affected by the area (or reconfiguration) overhead of synthesizing two or more versions of an accelerator and the cost of switching between them. Our approach, in contrast, augments a truly



(a) Weighted Histogram



(b) Maximal Matching



(c) Matrix Power

Fig. 14. Execution time and resource utilization of the applications as created by Vivado HLS compared to the dynamically scheduled designs with our LSQ in different sizes (sizes 4, 8 and 16, as indicated by the labels next to the points). Although the LSQ designs increase the resource requirements, they achieve significant speed-ups over the static designs. For each point, we provide the achieved *initiation interval* (II) and *clock period* (CP).

dynamically scheduled accelerator with an LSQ (whose size may be nontrivial) while avoiding the costs associated with replicas and the functional limits of these specific techniques.

Tan et al. [19] describe an approach called ElasticFlow to apply loop pipelining on a class of irregular loop nests with no inter-iteration dependencies in the outer loops. In their approach, multiple pipeline instances of a dynamic-bound inner loop are scheduled to execute in parallel. Dai et al. [8] evaluated methods for pipeline flushing by statically scheduling multiple pipeline initiation intervals to resolve different possible resource collisions, and later augmented their designs with application-specific dynamic hazard detection and resolution circuitry [9]. The underlying methodology in all these techniques is still based on standard static scheduling, opportunistically adapted and optimized. Although this eliminates the need for a complex memory interface which supports out-of-order execution, it strongly limits the achievable performance improvements and enables only some level of dynamic behavior. The out-of-order LSQ presented in this paper is a generic solution which fully supports dynamic and irregular application behavior and can be integrated into any architectural template.

Spatial or dataflow computing distributes the control and eliminates the requirement and expectation of static reasoning: each operation executes as soon as all of its inputs arrive, and physical operators pass along control "tokens" and the data they produce [16]. Different authors exploited latency-insensitive protocols to construct dynamic, high-performance circuits. Kam et al. [13] created dynamic pipelines using elastic circuits [7] and showed that the resulting pipelines can resolve dynamic dependencies that static HLS scheduling cannot. Huang et al. [11] introduced an elastic *coarse-grain reconfigurable array* (CGRA), which can tolerate variable latency operations; however, they serialize potentially dependent accesses at the memory interface ("The memory dependence is implemented by creating a lockstep between the corresponding [...] memory ports" [11]). Similarly, Budiu et al. [2] described a spatial architecture that uses an LSQ to detect and resolve dynamic data dependencies, but inadvertently serialize memory accesses whose dependencies cannot be resolved statically ("we insert a token edge between two instructions only if their points-to sets overlap and they do not commute" [3]). These tokens correspond exactly to enforcing the dashed edges of Figure 4(b). Both approaches are essentially the same as the allocation process that we showed in Figure 5, which nullifies the benefits of dynamic scheduling in hardware; Budiu et al. maintain the LSQ, while Huang et al. omit it, most likely due to its seemingly limited value in their contexts. We believe that this is the most common approach taken in prior dynamically scheduled spatial designs. Our LSQ, with its group allocation policy, qualitatively improves upon the state of the art, enabling spatial architectures to exploit memory access parallelism; our results show that including our LSQ in spatial designs enables highly optimized dynamic scheduling.

8 CONCLUSION

Spatial architectures for applications with irregular behaviour require dynamic dataflow-like scheduling to achieve their potentials. Dataflow execution naturally calls for memory interfaces capable of dynamically reordering accesses based on actual dependencies through memory instead of potential collision that a compiler cannot exclude. Traditional LSQs for out-of-order processors seem the right component for such memory interfaces. Yet, the elusive aspect of allocation of memory accesses in the LSQ may have made most past designs more conservative than needed and significantly less performant than it would be possible. In this paper we discussed the possible LSQ allocation strategies and focused on a novel policy which we believe is perfectly matched to the information available in HLS tools and silicon compilers, and achieves the level of performance one would expect. We have presented a microarchitecture for such an LSQ and we have shown on practical examples that we can indeed achieve significant speedups compared to commercial, statically scheduled, HLS tools. This indicates that LSQs similar to ours have the potential to become standard for dynamically scheduled designs.

REFERENCES

[1] M. Alle, A. Morvan, and S. Derrien. 2013. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Proceedings of the 50th Design Automation Conference*. Austin, Tex, June 2013.

[2] M. Budiu, P. V. Artigas, and S. C. Goldstein. 2005. Dataflow: A complement to superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. Austin, Tex., 177–186, Mar. 2005.

[3] M. Budiu and S. C. Goldstein. 2002. Pegasus: An Efficient Intermediate Representation. Technical Report CMU-CS-02-107. Carnegie Mellon University, May 2002.

[4] M. Budiu and S. C. Goldstein. 2003. Optimizing memory accesses for spatial computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization*. San Francisco, Calif., 216–27, Mar. 2003.

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

[6] S. Cheng and J. Wawrzynek. 2016. Synthesis of statically analyzable accelerator networks from sequential programs. In *Proceedings of the International Conference on Computer Aided Design*. Austin, Tex., 126–133, Nov. 2016.

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

[8] S. Dai, M. Tan, K. Hao, and Z. Zhang. 2014. Flushing-enabled loop pipelining for high-level synthesis. In *Proceedings of the 51st Design Automation Conference*. San Francisco, Calif., 1–6, June 2014.

[9] S. Dai, R. Zhao, S. S. Gai Liu, U. Gupta, C. Batten, and Z. Zhang. 2017. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., 189–94, Feb. 2017.

[10] J. Huang, Y. Huang, Y. Chen, P. Ienne, O. Temam, and C. Wu. 2014. A low-cost memory interface for high-throughput accelerators. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*. New Delhi, 11:1–11:10, Oct. 2014.

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

[12] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. 2002. Synchronous interlocked pipelines. In *Proceedings of the 8th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. Manchester, 3–12, Apr. 2002.

[13] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms. 2008. Correct-by-construction microarchitectural pipelining. *Proceedings of the International Conference on Computer Aided Design* (Nov. 2008), 434–41.

[14] J. Liu, S. Bayliss, and G. A. Constantinides. 2015. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. In *Proceedings of the 23rd IEEE Symposium on Field-Programmable Custom Computing Machines*. Vancouver, B.C., 159–62, May 2015.

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

[16] M. Pellauer, A. Parashar, M. Adler, B. Ahsan, R. L. Allmon, N. C. Crago, K. Fleming, M. Gambhir, A. Jaleel, T. Krishna, D. Lustig, S. Maresh, V. Pavlov, R. Rayess, A. Zhai, and J. S. Emer. 2015. Efficient control and communication paradigms for coarse-grained spatial architectures. *ACM Trans. Comput. Syst.* 33, 3 (2015), 10:1–10:32.

[17] M. Pericàs, A. Cristal, F. J. Cazorla, R. González, A. V. Veidenbaum, D. A. Jiménez, and M. Valero. 2008. A two-level load/store queue based on execution locality. In *Proceedings of the 35th International Symposium on Computer Architecture*. Beijing, 25–36, June 2008.

[18] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. 2007. Late-binding: Enabling unordered load-store queues. In *Proceedings of the 34th International Symposium on Computer Architecture*. San Diego, Calif., 347–57, June 2007.

[19] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang. 2015. ElasticFlow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the International Conference on Computer Aided Design*. Austin, Tex., 78–85, Nov. 2015.

[20] M. Vijayaraghavan and Arvind. 2009. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 9th ACM/IEEE International Conference on Formal Methods and Models for Codesign*. 171–80, July 2009.

[21] H. Wong, V. Betz, and J. Rose. 2013. Efficient methods for out-of-order load/store execution for high-performance soft processors. In *Proceedings of the IEEE International Conference on Field Programmable Technology*. Kyoto, 442–445, Dec. 2013.

Received April 2017; revised May 2017; accepted July 2017