

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

Lana Josipovic\*, Philip Brisk†, and Paolo Ienne\*

\*École Polytechnique Fédérale de Lausanne, Switzerland, †University of California, Riverside, USA  
 lana.josipovic@epfl.ch, philip@cs.ucr.edu, paolo.ienne@epfl.ch

## ABSTRACT

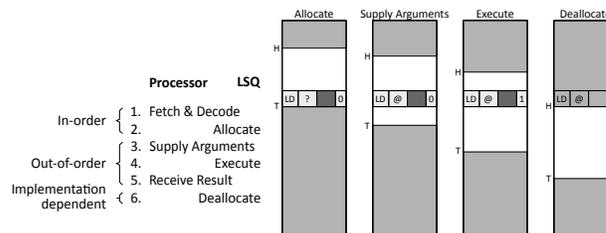
The efficiency of spatial computing depends on the ability to achieve maximal parallelism. This needs memory interfaces that can correctly handle memory accesses arriving in arbitrary order while respecting data dependencies and ensuring appropriate ordering for semantic correctness. However, a typical memory interface for out-of-order processors (i.e., a load-store queue) cannot immediately fulfill these requirements: a different allocation policy is needed to achieve out-of-order execution in spatial systems because they naturally omit the notion of sequential program order, fundamental for correct execution. We demonstrate a practical way to organize the allocation for an out-of-order load-store queue for spatial computing by dynamically allocating groups of memory accesses.

## 1 INADEQUACY OF PROCESSOR LOAD-STORE QUEUES

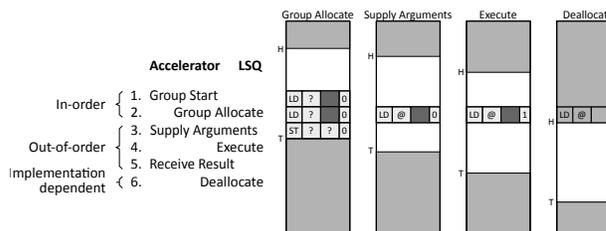
With reconfigurable devices being deployed in datacenters and the likelihood of new, irregular applications that could benefit from FPGAs, different authors have explored techniques which can effectively implement dynamic, dataflow-like spatial circuits. Although dynamic behavior has been exploited in processors for decades [1], there has been little effort on creating generic accelerator-memory interfaces that support out-of-order execution. The reason lies in a fundamental difference of the two systems: In a processor, the notions of fetching and decoding instructions immediately convey the *correct* sequential order of requests at the memory interface (Figure 1). 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. Prior research on spatial computing has mentioned the possibilities of employing LSQs [2], but has not described how one performs allocation in a spatial context nor how the LSQ is able to decide which reorderings are legal. Answering this open question is the foremost contribution of this work.

## 2 AN OUT-OF-ORDER MEMORY INTERFACE FOR SPATIAL COMPUTING

Our load-store queue implements the following functionalities: (1) Allocating entries in the queue. (2) Enabling the access ports and connecting them to the respective LSQ entries. (3) Accepting arguments for the allocated LSQ entries as they arrive out-of-order. (4) Deciding dynamically which accesses can be safely executed without violating dependencies in memory. (5) Returning as soon as possible available results to the load ports. Almost everything related to steps (2) to (5) is identical or similar to what happens in a processor LSQ, whereas function (1) is totally different for our architecture and is based on a group allocation policy (Figure 2). We define a *group* as a sequence of accesses with predetermined sequential order. The start of a group indicates that all accesses belonging to that group need to be performed (there is no conditional



**Figure 1: A traditional LSQ of an out-of-order processor. Although the LSQ executes out-of-order, the allocation of entries in the queue must happen in sequential program order, at Decode time. However, spatial accelerators have no equivalent phase to Decode.**



**Figure 2: Group allocation. 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.**

branching in between the accesses of the group) and the groups need to be triggered in the correct sequential order. When a group starts, we dynamically allocate positions in the queue for all memory operations that belong to it. The memory requests might come out-of-order—since their order is statically determined and defined within the LSQ at runtime, it can appropriately order them before issuing them to memory. This allows multiple groups to execute in parallel and concurrently issue operations. We form the groups in a way that each group corresponds to a *basic block* generated by the compiler of a high-level synthesis tool.

## 3 RESULTS AND CONCLUSION

We show the advantages of our LSQ over standard accelerator-memory interfaces on simple but practical cases, achieving significant speedups (of up to 3×) compared to designs created by a commercial high-level synthesis tool. With an affordable resource cost (i.e., with an LSQ of depth 8 that occupies only under 5% of a typical FPGA), we attain almost the maximum parallelism available in the application. This indicates that LSQs like ours have the potential to become standard for dynamically scheduled accelerators.

## REFERENCES

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Fifth edition, 2012.  
 [2] J. Huang, Y. Huang, Y. Chen, P. Ienne, O. Temam, and C. Wu. A low-cost memory interface for high-throughput accelerators. In *CASES*, Oct. 2014.