Dynamically Scheduled Memory Operations in Static 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 higher throughput on codes with unpredictable memory accesses compared to static HLS. However, dynamic scheduling results in circuits that use more resources and have a slower critical path, even if only a small part of the circuit exhibits dynamic behavior. In this extended abstract, we propose to introduce dynamically scheduled memory operations into static HLS. Our goal is to reach the same throughput as dynamic HLS on codes with irregular memory accesses while achieving comparable resource usage and critical paths as static HLS.**

I. Limitation of Modulo Scheduling

A loop schedule in static HLS is obtained using modulo scheduling [1], which arrives at a minimum loop initiation interval (II) by calculating for each recurrence i in the Data Dependence Graph (DDG):

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

where the $delay$ is the sum of instruction latencies on the recurrence path, and $distance$ is the minimum iteration distance between the definition of the value calculated by the recurrence and its use. Since modulo scheduling has to arrive at a single II for the loop, it has to necessarily over-approximate if the delay or dependence distance are unknown, for example if memory accesses are data dependent:

```
for (int i = 0; i < N; ++i)
  data[idx[i]] = f(data[idx[i]]);
```

II. Dynamic Memory Operations in Static HLS

Dynamically scheduled memory operations require runtime memory disambiguation machinery such as a Load-Store Queue (LSQ) [2], which relies on the separation of memory address generation from accesses for optimal operation. Such a separation is natural in dataflow circuits. To achieve the same effect in modulo-scheduled HLS, we propose to decouple the address generation instructions into its own loop, similar to the principle of decoupled access/execute architectures. Memory requests from the address generation loop, and the actual loads and stores are connected to an LSQ via latency-insensitive channels, as presented in fig. 1. Since replacing loads and stores with latency-insensitive channel read/writes removes the inter-iteration memory dependencies, modulo scheduling will be able to achieve an II of 1 if no other restrictions are found. The latency-insensitive communication ensures that a component will stall if a memory request needs to wait.

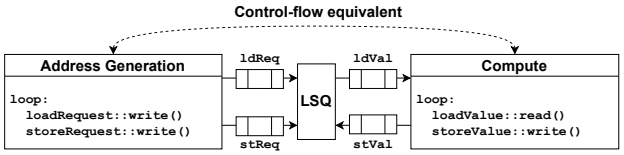

Fig. 1. The communication pattern between decoupled address generation, a Load-Store Queue (LSQ), and compute. The channels are latency-insensitive. Each component is a seperate modulo-scheduling instance.

The problem of LSQ request ordering is solved by the design of our LSQ, which is based on tagged memory operations – each load and store request is tagged with an integer value which represents the state of the memory at that point; stores increment the tag before making a request, loads use the tag directly. The tag can be used to cheaply disambiguate loads, i.e. a load request i will wait if for any store request k:

$$ldReq_i.addr = stReq_k.addr \ \ \& \ \ ldReq_i.tag \geq stReq_k.tag.$$

It is not always possible or profitable to decouple the address-generating instructions. Given a set of address-generating instructions I_{GEN} for a given address, and a set of memory access instructions I_{ACCESS} using addresses from I_{GEN}, we decide to decouple the I_{GEN} instructions if:

> $\forall i \in I_{GEN}$ where i is used by a store, i is not control nor data dependent on any instruction j, such that there is a DDG path from an instruction $k \in I_{ACCESS}$ to j.

In other words, if the execution of a store, or its address calculation, depends on the value of a load from the same base address, then decoupling is not applicable. This restriction is not a limitation of our approach, since even a fully dynamic HLS tool would have to stall in such a situation.

Conclusion: We proposed an approach for introducing dynamically scheduled memory operations into static HLS. Our technique allows us to dynamically schedule only the part of a circuit that exhibits dynamic behavior, keeping the rest of the circuit static.

References

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

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