Dynamic Loop Fusion in High-Level Synthesis

Robert Szafarczyk[*,1], Syed Waqar Nabi[*], Wim Vanderbauwhede[*]

[*]*School of Computing Science, University of Glasgow, Scotland, UK*

ABSTRACT

Loop fusion is a powerful compiler transformation that can reduce latency, increase throughput, and decrease memory traffic. However, existing optimizing compilers impose strict requirements on the types of loops that can be fused, often only considering loops with affine memory accesses and equal loop bounds. This problem is exacerbated in High-Level Synthesis (HLS) because of the sensitivity of its scheduling algorithm to control-flow changes.

We propose a new approach to loop fusion specific to HLS that is able to fuse strictly more loops than previous approaches, with the only requirement being that the address expression increases monotonically within any given loop depth. Our approach decouples each loop into its own hardware thread of execution. Memory dependencies across loops are disambiguated dynamically by program-specific logic generated by the compiler.

1 Introduction

Given a set of loops that fulfill certain conditions, loop fusion will merge one or more loops into one to increase parallelism and data locality. For example, these loops can be merged,

```
// Before fusion
for i = (startA, endA) A[f(i)] = compute();
for i = (startB, endB) B[g(i)] = compute();
for i = (startC, endC) C[h(i)] = A[h(i)] + B[h(i)];

// After fusion
for i = (startA, endA)
    A[f(i)] = compute();
    B[g(i)] = compute();
    C[h(i)] = A[h(i)] + B[h(i)];
```

if the following requirements are met:

1. $startA = startB = startC$.

2. $endA = endB = endC$.

[1]E-mail: robert.szafarczyk@glasgow.ac.uk

3. For a given integer $startA \leq i < endA$, the following holds $h(i) < f(j) \wedge h(i) < f(k)$, where $i \leq j < startA \wedge i \leq k < startA$. In other words, the fusion of the loop bodies does not introduce a dependency with a negative dependency distance [KA01].

Although the previous example is trivial to fuse, many real codes are not. Requirement 1 and 2 (equal loop bounds), can usually be relaxed by masking of individual iterations of specific loops using if-conditions. Requirement 3 is harder to relax because it involves an analysis of the address values which can evolve throughout the execution of the loop. The current state-of-the-art loop fusion approaches represent loops using integer polyhedra. [BPCB10] This representation requires that the address expressions are affine functions in the loop iteration variables (and optional constant parameters). With such a representation, loop fusion and other transformations can be represented as affine function transformations, whose legality and optimality can be searched for using integer linear programming.

Extensive research has been done on static loop fusion to make it applicable to a broader range of codes. For example, iteration skewing and loop hoisting can enable loop fusion [KA01]. However, such transformations are not well suited for HLS, and more importantly, the polyhedral model cannot represent non-affine address expressions and loop bounds. We propose a new approach to loop fusion in HLS that is capable of fusing non-affine loops, with the only requirement being address expression monotonicity.

2 Background on High-Level Synthesis

High-Level Synthesis (HLS) compilers generate a hardware description of a program-specific accelerator from a high-level software language such as C or Fortran. The key advantage of accelerating a given code using HLS, as compared to using a CPU or GPU, is the ability to generate much deeper execution pipelines. Together with shedding the overhead of interpreting an instruction set and the support for arbitrary precision numbers, HLS can achieve orders of magnitude better performance-per-watt than CPUs and GPUs.

Recovering pipeline parallelism from sequential code is the job of scheduling algorithms. Scheduling in HLS is NP-hard: we try to maximize pipeline parallelism under the constraint of available chip resources. Most current approaches use some form of the modulo scheduling algorithm originally proposed to schedule instructions on VLIW machines [RG81]. Modulo scheduling tries to find the smallest possible number of cycles that have to pass between the starting of two subsequent loop iterations. This number (the loop initiation interval) depends on the relation between operation latency and the dependency distance between operations. The problem with such a static scheduling approach is that schedules generated for irregular programs are often suboptimal because unknown operation latencies and dependency distances have to be conservatively overapproximated.

Recent years have seen renewed interest in dynamically scheduled HLS [JGI22]. The idea here is to augment functional units with additional handshaking hardware that can perform scheduling at runtime, thus adapting to runtime conditions. While this approach improves the throughput of irregular codes, it does so at the expense of increased resource usage and circuit critical path.

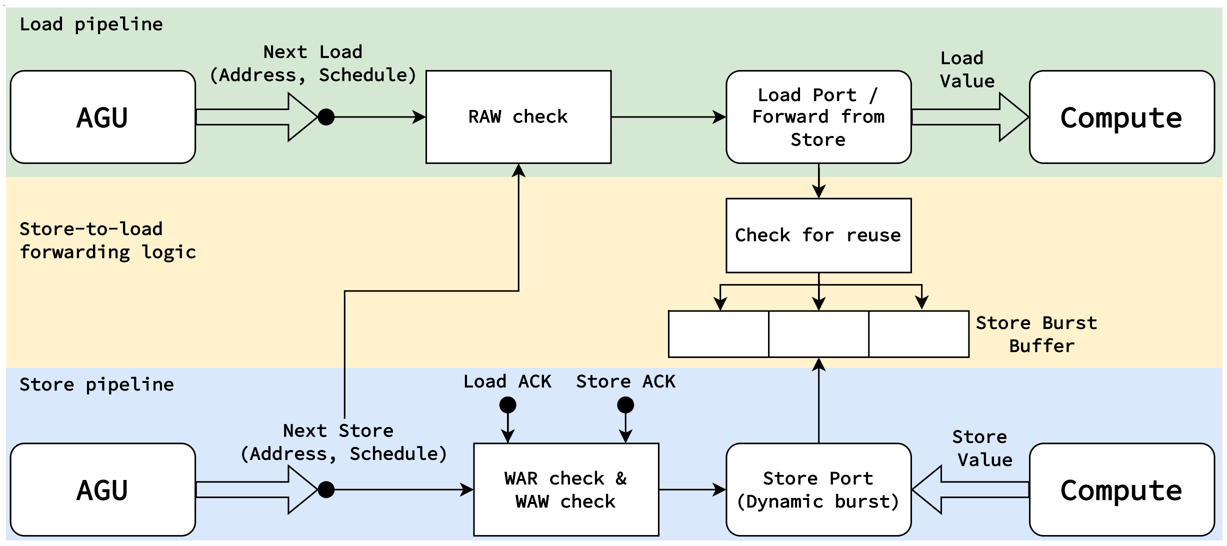

Figure 1: An overview of our memory controller template enabling dynamic loop fusion. The RAW, WAR, WAW checks, and the AGUs are compiler generated based on data dependencies and address expressions in the input program.

2.1 Our Hybrid Approach to Scheduling

We are interested in a hybrid approach where the compiler introduces dynamic scheduling only for parts of the circuit that cannot be scheduled efficiently using modulo scheduling. Our approach uses compiler analysis to decouple the data dependency graph of the program into multiple modulo-scheduling instances. Data dependencies across instances are communicated using handshaking logic that has the ability to stall individual instances. We have shown that this approach can achieve the same throughput as dynamically scheduled HLS, but without significant area and critical path overhead [SNV].

3 Dynamic Loop Fusion

Research on dynamically scheduled HLS up to now has mostly focused on pipeline parallelism within a single loop. However, most irregular programs consist of multiple loops. With the current HLS tools, these loops need to be executed sequentially. As outlined in the introduction, existing loop fusion approaches are not able to analyze non-affine programs.

We propose to fuse loops by decoupling each loop into its own hardware thread. Memory dependencies across loops are detected and resolved at runtime by a custom memory controller (figure 1 shows a high-level overview of the logic). The controller performs read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) hazard resolution using information supplied by the address generation units (AGUs). The AGUs are generated by the compiler from the input program. They supply the memory request addresses, together with a schedule (an n-tuple that is used to recover the relative order between requests). Our schedule representation is equivalent to the schedule representation used in polyhedral frameworks, but is optimized for efficient hardware execution. The memory controller is also able to directly forward values from producers to consumers without issuing memory requests.

3.1 Monotonicity of Address Expressions

Our hazard resolution logic assumes that the address expression is monotonic in the inner most loop. If the address expressions of two memory requests are monotonic, then given their n-tuple schedule, we can check if the requests will alias in a given schedule span.

The monotonicity requirement only applies to the innermost loop depth, meaning that the value of the address expression can still decrease as a result of advancing in the iteration space in a different loop. Given a loop nest,

```
for i_1 = (start_1, end_1) // loop depth 1
    for i_2 = (start_2, end_2) // loop depth 2
        ...
            for i_n = (start_n, end_n) // loop depth n
                A[f(i_1, i_2, ..., i_n)] = ...
```

we require that for loop depth n, there does not exist an iteration i_n, such that $f(i_1, ..., i_n) > f(i_1, ..., i_{n'})$, where $i_n < i_{n'}$. In other words, within the loop depth n, the address expression f is monotonically increasing. Analysis of this requirement is feasible in today's compilers (both LLVM and GCC can represent such expressions as trees of recurrences). For data-dependent address expressions, a monotonicity guarantee can be provided by the programmer by using a pragma annotation. Many sparse codes have a monotonic address expression in the inner most loop (e.g. codes using the Compressed Sparse Row (CSR) format).

4 Conclusion

We introduced the idea of dynamic loop fusion in HLS. The transformation is enabled by a custom memory controller that can disambiguate memory requests from different loops at runtime, and by a compiler pass that decouples loops into separate hardware threads and augments memory requests with scheduling information needed for disambiguation.

References

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, 2010.

[JGI22] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. From c/c++ code to high-performance dataflow circuits. *IEEE TCAD*, 2022.

[KA01] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2001.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, MICRO 14, 1981.

[SNV] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. Compiler discovered dynamic scheduling of irregular code in high-level synthesis. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*.