Compiler-Hardware Co-Design in High-Level Synthesis

Robert Szafarczyk

Submitted in fulfillment of the requirements for the
Degree of Doctor of Philosophy

School of Computing Science
College of Science & Engineering
University of Glasgow

April 2025

Abstract

High-Level Synthesis (HLS) simplifies the hardware design process by generating specialized hardware directly from an algorithmic software description. Current HLS tools work well on regular code, but are suboptimal on irregular code with data-dependent memory accesses and control-flow. This is because they follow a Finite State Machine with Datapath (FSMD) model of computation, which requires the compiler to schedule operations statically at compile time, failing to adapt to runtime conditions. A Dynamic Dataflow (DDF) model of computation in HLS augments each functional unit with additional scheduling logic that enables dataflow scheduling at runtime, naturally adapting to the unpredictable conditions in irregular codes. However, the resulting hardware generated by DDF HLS uses more area and produces longer critical paths than necessary, because every operator in the circuit is scheduled dynamically, even if only a few exhibit irregular behavior.

In this thesis, we propose a closer compiler-hardware co-design to make the HLS of irregular codes more efficient. We make four significant contributions. First, we show how the FSMD computational model can be extended with DDF behavior without having to schedule the entire circuit dynamically. This is achieved by letting the compiler discover sources of irregularity that prevent efficient static scheduling and by decoupling the original code into multiple FSMD instances along the discovered sources of irregularity. Second, we show how a compiler can automatically generate a Decoupled Access/Execute (DAE) architecture to enable efficient out-of-order dynamic memory scheduling in HLS, and we show how a compiler can automatically parametrize hardware structures, such as a Load-Store Queue (LSQ), to maximize throughput at minimal area usage. Third, we introduce compiler support for speculation in DAE architectures with two algorithms: one that speculates memory requests in the access program slice, and another that poisons mis-speculations in the compute slice, all without the need for mis-speculation recovery or synchronization. And finally, we show that a close compiler-hardware co-design can enable new optimization opportunities by presenting dynamic loop fusion. This novel technique is able to fuse the execution of sibling loops dynamically at runtime by resolving inter-loop memory dependencies in a hardware structure parametrized by the compiler. To enable dynamic loop fusion, we introduce a new hardware-optimized program-order schedule inspired by polyhedral compilers and we exploit the concept of monotonically non-decreasing address expressions—a larger class of functions than affine expressions required in static loop fusion. Our FPGA-based experiments show that our four contributions consistently result in at least an order of magnitude area-delay improvement over state-of-the-art HLS tools on irregular codes.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to thank my advisors, Wim Vanderbauwhede and Syed Waqar Nabi, for guiding me through this academic journey and for the many discussions, not only about my research. Your advice was invaluable and has shaped me as a scientist.

My thanks go to Michel Steuwer and Nikela Papadopoulou, who were my viva examiners, for providing a thorough yet enjoyable “grilling” and for the many suggestions on improving the presentation of this work. I would also like to thank Simon Fowler for convening the viva.

I am grateful for the scholarship I received from the UK Engineering and Physical Sciences Research Council, which enabled me to pursue a PhD.

Thanks to all the other colleagues in the School of Computing Science who made this three-and-a-half-year journey special. Thanks to Youseff Moawad for sharing the pains of using FPGA tools and the occasional lunch. Thank you to my office mates—Dejice Jacob, Duncan Lowther, and Jacob Trevor—for making the day-to-day more enjoyable. I also appreciate the efforts of our Systems Research Section head, Jeremy Singer, in organizing paper reading groups and similar events that improved my experience as a PhD student.

I would also like to thank my former colleagues from my undergraduate internship at Intel—Clare Somerville, Daniel Towner, Chris Luke, Steve Wood, and Nick Whinnett—for inspiring me to pursue a PhD.

I am grateful for my family and friends who have supported me through all these years, especially my partner Patrycja, who has agreed to move with me to Glasgow, despite its dreich weather, and who is the bedrock of my life.

Declaration & List of Publications

I declare that, except where explicit reference is made to the contribution of others, this thesis is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution. The work described in this thesis was published in:

- [213] 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)*. 2023, pp. 1–9. DOI: `10.1109/FPL60245.2023.00009` **(Best Paper Award)**
- [217] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Dynamically Scheduled Memory Operations in Static High-Level Synthesis”. In: *IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2023, pp. 220–220. DOI: `10.1109/FCCM57271.2023.00048` **(Extended Abstract)**
- [212] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “A High-Frequency Load-Store Queue with Speculative Allocations for High-Level Synthesis”. In: *2023 International Conference on Field Programmable Technology (ICFPT)*. 2023, pp. 115–124. DOI: `10.1109/ICFPT59805.2023.00018`
- [215] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Compiler Support for Speculation in Decoupled Access/Execute Architectures”. In: *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. CC ’25. Las Vegas, United States, 2025. DOI: `10.1145/3708493.3712695`
- [216] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Dynamic Loop Fusion in High-Level Synthesis”. In: *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’25. Monterey, CA, USA: Association for Computing Machinery, 2025. DOI: `10.1145/3706628.3708871`

The following unrelated work was also performed during the work on this thesis:

- [218] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Reducing FPGA Memory Footprint of Stencil Codes through Automatic Extraction of Memory Patterns”. In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022, pp. 148–152. DOI: `10.1109/FPL57034.2022.00033` **(Short Paper)**

Robert Szafarczyk

Abbreviations

- **AGU**: Address Generation Unit
- **BRAM**: Block Random-Access Memory
- **CDG**: Control Dependency Graph
- **CGRA**: Coarse Grain Reconfigurable Architecture
- **CFG**: Control Flow Graph
- **CU**: Compute Unit
- **DAE**: Decoupled Access/Execute
- **DAG**: Directed Acyclic Graph
- **DDG**: Data Dependency Graph
- **DDF**: Dynamic Dataflow
- **DRAM**: Dynamic Random-Access Memory
- **DSE**: Design Space Exploration
- **DU**: Data Unit
- **FPGA**: Field Programmable Gate Array
- **FSMD**: Finite-State Machine with Datapath
- **GAA**: Guarded Atomic Action
- **HLS**: High-Level Synthesis
- **II**: Initiation Interval
- **ISA**: Instruction Set Architecture
- **LoD**: Loss of Decoupling
- **LSQ**: Load-Store Queue
- **OoO**: Out-of-Order
- **PC**: Program Counter
- **PE**: Processing Element
- **RAW**: Read After Write
- **RTL**: Register Transfer Level
- **SSA**: Single Static Assignment
- **SCC**: Strongly Connected Component
- **WAR**: Write After Read
- **WAW**: Write After Write

Chapter 1

Introduction

Digital computer systems consist of layers of abstractions from electrons to algorithms. More often than not abstraction has a cost in terms of chip area or performance, and also more often than not that cost is justified. A stable Instruction Set Architecture (ISA), virtual memory, or precise exceptions are examples of abstractions fundamental to today's general-purpose computer systems. However, in some scenarios it makes sense to break some of these abstractions and use specialized hardware—bringing the algorithm closer to the electrons as illustrated in Figure 1.1. Some computer systems have strict latency or throughput requirements that cannot be met with general-purpose CPUs or GPUs; other systems operate at such a large scale that it makes economic sense to use specialized hardware. Fundamental limits to transistor technology scaling and the resulting slowdown of CPU and GPU performance improvements mean that specialized hardware becomes increasingly more popular and with it the importance of the hardware design process [108].

HLS significantly simplifies the hardware design process by providing an automated path from an algorithmic description to Register Transfer Level (RTL). Program annotations can guide the HLS tool to find a desired optimization point in a multi-dimensional space of area usage, throughput, latency, and circuit frequency . After decades of research [158] on improving the HLS quality, and as computer architects build more specialized accelerators, HLS has become a mature tool used in industry. For example, in 2021 Google described its use of HLS when designing a video coding accelerator for its YouTube video service [193]. The HLS benefits that they cite show why the technology will only become more important in the future as more accelerators are build:

- High productivity and code maintainability: compared to using RTL 5–10× less code to write, review, and maintain.
- Accelerated verification: using a higher abstraction layer means 7–8 orders of magnitude faster verification.
- Focusing engineering effort: leaving the cycle-by-cycle data path control logic to the HLS compiler, means more time spent on algorithm and (macro)architecture design.
- Design space exploration: HLS makes it easy to write parametrized designs, enabling automated tools to sweep through a large design space of area/throughput trade-offs.

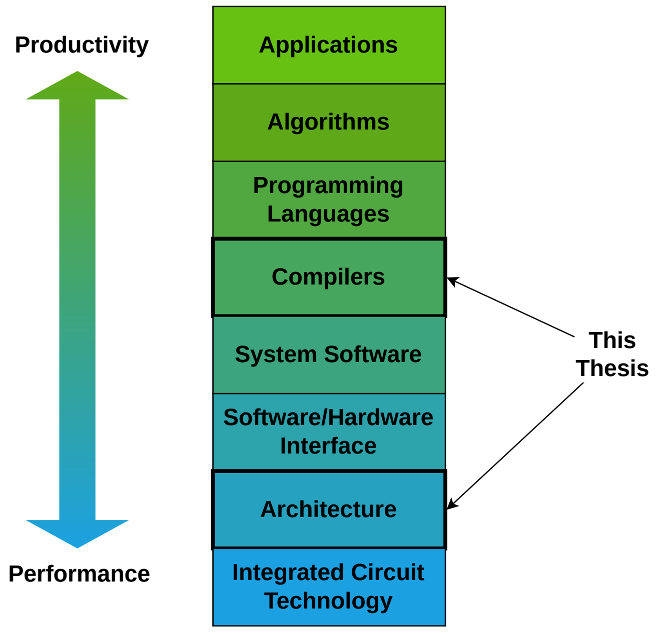


Figure 1.1: Getting the most performance out of our computer systems requires crossing the compute stack boundaries. This thesis primarily investigates the co-operation between compilers and computer architecture.

- Late feature flexibility: architectural adjustments late in the project to support late feature requests and address challenges exposed in the place and route state of the physical design are easier to do compared to RTL.

However, the success of HLS today is limited to only regular codes—codes where the compiler is able to statically analyze all operation latencies and dependencies. HLS of irregular codes is an active area of research, which we contribute to in this thesis.

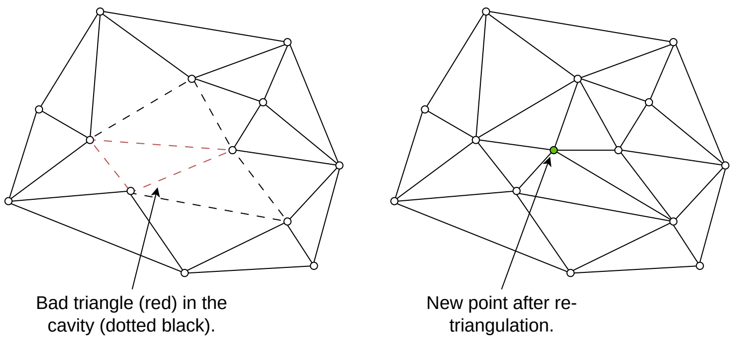
1.1 The Importance of Irregular Code

The terms “irregular code” and “regular code” have no precise definition in the literature, and they can mean slightly different things in different contexts. In this thesis, a given code is said to be irregular if it contains dynamic behavior that the compiler cannot predict statically, such that some code optimization (e.g., loop pipelining) cannot be performed.

Many interesting programs do contain dynamic, runtime-dependent behavior, which is unpredictable at compile time [138]. Popular domains with irregular codes are graph analytics (e.g., PageRank), sparse linear algebra (e.g., sparse matrix-vector multiply), error correction codes (e.g., Low-Density Parity-Check codes), or computational geometry (e.g., Delaunay Mesh Refinement). Typically, the order of items being processed in irregular codes is not known at compile time—it is dynamic. This is illustrated in Figure 1.2 showing one step in a Delaunay Mesh Refinement code. We can see, that the nodes being processed inside the

```
Mesh m = readInputMesh();
Worklist wl = {m.getBad()};
while (!wl.empty())
    Triangle t = wl.pop();
    Cavity c = getCavity(t);
    c.retriangulate();
    m.updateMesh(c);
    wl.push(c.getBad());
```

(a) Delaunay Mesh Refinement.


Bad triangle (red) in the cavity (dotted black).
New point after re-triangulation.

(b) Change in the mesh after one refinement.

Figure 1.2: An example mesh refinement code illustrating the dynamic behavior of irregular codes. The underlying mesh data structure changes dynamically, resulting in behavior that is unpredictable at compile time.

loop body cannot be predicted at compile time, because they depend on the calculations done inside the loop. In contrast, in a regular code, such as a dense matrix product, the order of items being processed can be analyzed to a much greater extent, allowing the compiler to parallelize the code. This crucial analysis information is missing in irregular codes, making the compiler's job much harder.

1.2 High-Level Synthesis of Irregular Code

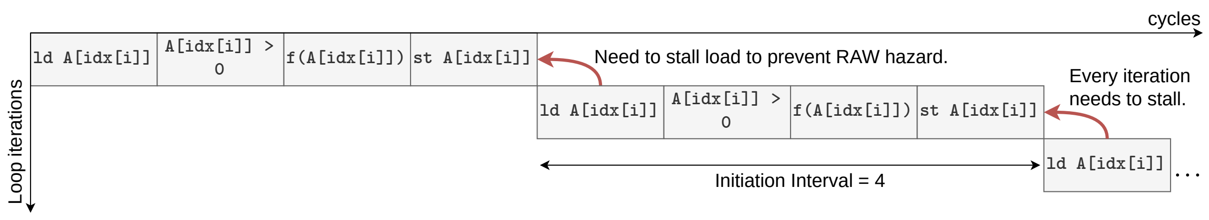
Why do existing HLS tools produce sub-optimal circuits for irregular codes, and how can we improve the state-of-the art with the work in this thesis? To motivate our technical contributions, it helps to identify the specific limitations of existing approaches to the HLS of irregular codes.

Most HLS tools today [120, 238, 38] follow the Finite State Machine with Datapath (FSMD) model of computation [105, 110] that requires the compiler to schedule datapath operations statically at compile time. Because of the reliance on static scheduling, FSMD HLS has to be conservative during scheduling and thus fails to synthesize efficient hardware for irregular codes with data-dependent behavior.

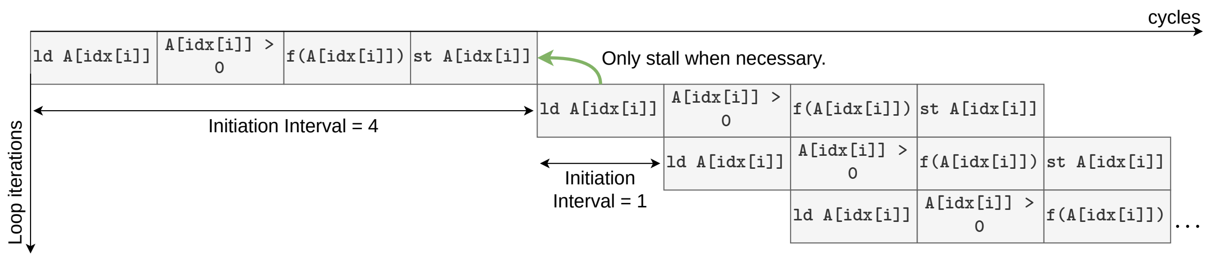
In the previous section, we gave a high-level intuition what irregular codes are and why they are difficult for the compiler to deal with. To illustrate the specific reasons why FSMD HLS struggles with such codes, it helps to have a small prototype of an irregular code, rather than an entire application. Consider the loop from Listing 1.1 for example as a representative of an irregular code.

Listing 1.1: Example of an irregular code (unpredictable memory addresses).

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



(a) Static HLS: one schedule needs to be followed for all iterations.



(b) Our work: the schedule can adapts to runtime conditions on every loop iteration.

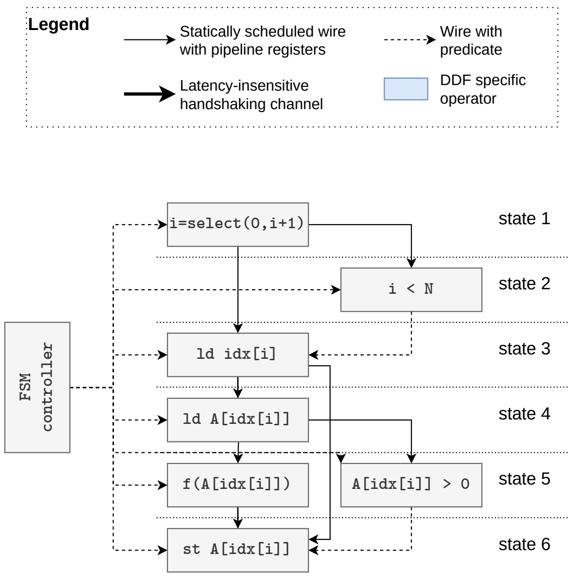
Figure 1.3: Pipelines achieved by current FSMD HLS tools (a) and our work (b) on the example loop from listing 1.1. Only the second iteration has a true data dependency, but FSMD HLS produces the same, sequential schedule for every iteration. Our work produces a schedule that adapts to runtime conditions.

This code contains multiple challenges that make static scheduling difficult:

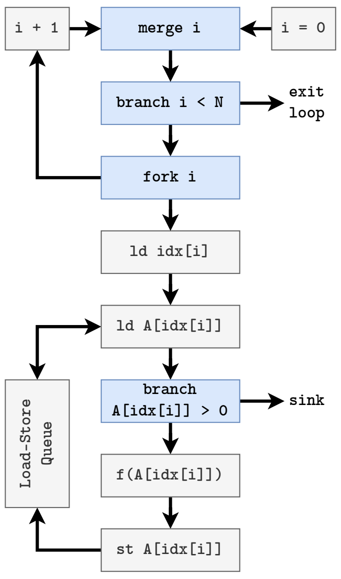
- Data-dependent memory accesses: the compiler cannot disambiguate individual accesses at compile time.
- Data-dependent control-flow: the compiler does not know how often the if-statement body will execute, and thus what the latency of the loop body will be.
- Variable-latency memory access: the compiler does not know what the memory access latency will be if a memory system non-deterministic latency is used (e.g., DRAM, or caches).

In FSMD HLS, the compiler assigns loop operations to states in an FSM. Figure 1.4a shows how our running example might be implemented by FSMD HLS. At runtime, only the operations corresponding to the current state in the Finite State Machine (FSM) controller are enabled. Pipeline parallelism can be achieved by merging multiple states into one and introducing pipeline registers to hold on to data that can only be consumed in later cycles. With perfect pipeline parallelism, we would start a new loop iteration on every clock cycle, ensuring perfect utilization of our hardware resources and maximum throughput—we say that the loop Initiation Interval (II) is 1, because one cycle passes between the start of subsequent loop iterations. However, before introducing any pipeline parallelism, the compiler needs to ensure that no inter-iteration dependencies would be violated.

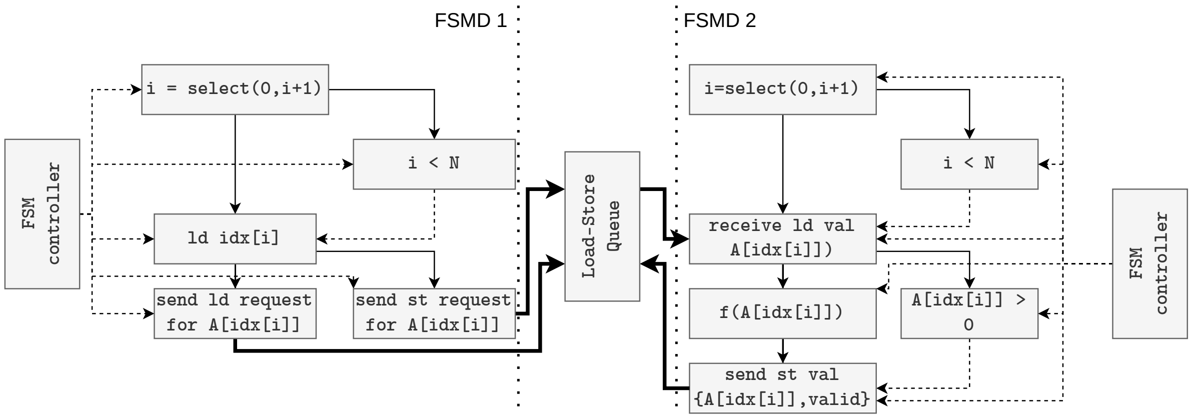
For example, in Figure 1.4a the compiler cannot merge states 4 and 5, because it cannot prove to itself statically that the load in state 4 will not read from the same address as some earlier store in state 6—there is an unpredictable RAW dependency. In result, FSMD HLS is forced to increase the loop II, sometimes completely sequentializing the loop pipeline, as illustrated in Figure 1.3a. The higher the loop II, the more cycles need to pass between the start of subsequent loop iterations, lowering the ultimate throughput of the computation.



(a) Static HLS approach using a single FSMD.



(b) DDF HLS approach with latency insensitivity introduced around every operator.



(c) Our approach: multiple FSMDs connected with latency-insensitive channels. Dynamic behavior is introduced selectively. An LSQ enables dynamic, speculative, out-of-order memory accesses.

Figure 1.4: An overview of HLS implementation strategies for the loop from Listing 1.1. Static HLS (a) produces a sequential pipeline because it cannot guarantee that FSM states can execute in pipeline parallel fashion without violating dependencies. DDF HLS is able to adapt to runtime conditions, but uses more area and cycle time than necessary to schedule everything dynamically. Our work (c) extends the static HLS FSMD model to support dynamic behavior by decoupling the original code into multiple FSMDs, introducing dynamic scheduling only where it is beneficial.

Unpredictable memory addresses are not the only source of irregularity. Unpredictable control flow can cause similar problems. Consider the loop in Listing 1.2 for example.

Listing 1.2: Another example of an irregular code (control-dependent reduction).

```
for (int i = 0; i < N; ++i)
    if (condition)
        reduction += f(reduction);
```

Here, we have a clear inter-iteration dependency on the `reduction` variable—assuming that the `f()` function takes more than one clock cycle, the compiler cannot pipeline the loop with an II of 1. The static schedule needs to allocate enough cycles in the loop to calculate the `f(reduction)` value before the next iterations starts, because it might need the result of this calculation. Crucially, this schedule is rigid and has to be repeated for every loop iteration. However, if the *if*-condition is rarely satisfied, then such a schedule would waste quite a few cycles allocated to calculating `f(A[idx[i]])`, work which might not even be needed at runtime.

The above problems are the same challenges that were burdening compiler writers for Very Large Instruction Word (VLIW) machines a few decades ago [140, 86]. In VLIW machines, as in FSMD HLS, the goal of discovering ILP lies entirely on the shoulders of the compiler, with the difference that in VLIW the compiler decides which instructions can execute together by packing them in a single instruction word, instead of merging FSM states. At a high level, the problem of scheduling irregular code in VLIW and FSMD HLS is essentially the same. In both cases, the compiler has limited information about the dependencies and control flow in the code and has to assume a worst-case scenario, sequentializing most of the execution.

Ultimately, the problem of scheduling VLIW machines for general purpose code, which contains many of the irregularities that we highlight here, was recognized to be intractable. Today, VLIW machines are mainly used in domains that contain regular codes [147], such as signal processing or machine learning. Most general-purpose processors are not VLIW machines; they are RISC machines that contain many hardware structures to enable dynamic, superscalar, out-of-order execution. Similarly, an efficient HLS of irregular codes calls for additional hardware structures to guide circuit execution at runtime, at which time the data dependencies and control-flow become known.

1.2.1 Dynamic Dataflow High-Level Synthesis

Dynamic Dataflow (DDF) HLS forgoes static scheduling completely, leaving the job of scheduling to *latency-insensitive handshaking logic* that determines the execution of operators at runtime [126, 74]. Latency-insensitive channels allow for communication between a sender and receiver without having to rely on their cycle behavior [87]. Each operator in a dataflow circuit has a wrapper with handshaking logic that stalls the operator until all inputs have arrived and the output port has space to accept new data. Figure 1.5 shows a simple handshaking protocol.

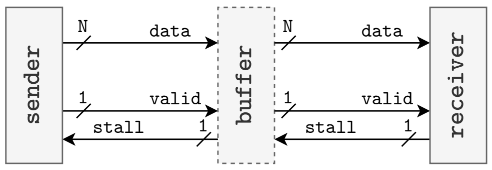


Figure 1.5: A latency-insensitive handshaking communication protocol with optional buffering. The `stall` signal implements *backpressure* that can cause upstream components to stall if some downstream component is not able to accept new data.

In recent years, DDF HLS has presented impressive throughput improvements on irregular codes compared to FSMD-based HLS [126]. However, the quality of results of DDF HLS in terms of area usage and critical path is still far away from manually written designs. This is because current DDF HLS implementations use dynamic scheduling throughout the entire circuit. In practice, irregular codes only have a few sources of irregularity and it is desirable to enable selective dynamic scheduling to achieve lower critical paths and area usage.

FSMD HLS is able to achieve better area usage due to the fact that static scheduling enables an efficient sharing of hardware resources—since the compiler knows when each operator will execute, it can share functional units without using additional wires and logic to do so. Many recent works try to move the DDF HLS quality of results closer to FSMD HLS by finding parts of the circuits where handshaking logic can be pruned [239, 50, 47, 50] or by providing a facility to dynamically share resources [128]. However, the improvements offered by these works are limited, with the resulting DDF HLS circuits still achieving significantly lower quality of results than is possible. This is because many parts of a dataflow circuit cannot be pruned without breaking the DDF computational model. For example, loops in DDF HLS are implemented by routing iterator value tokens through a series of combinatorial logic operators (see Figure 1.4b). To sustain their throughput advantage, DDF circuits must complete this routing in a single clock cycle, otherwise, a loop II of 1 is not possible. Because of this critical path, even the simple looping structure from Listing 1.1 already has a critical path overhead in the DDF HLS model.

We believe that a compiler combining static and dynamic scheduling should use the FSMD compute model by default, and only introduce selective dynamic scheduling where needed. There should be no performance overhead when synthesizing circuits for regular codes. The situation where an HLS uses a different compiler depending on if the code has some dynamic behavior is untenable, not least because most programmers are not familiar with optimizing compiler theory and are unable to say if a compiler can successfully analyze a given code at compile time.

1.2.2 Compiler-Hardware Co-Design in High-Level Synthesis

In this thesis, we propose to enable selective dynamic behavior in the FSMD based compute model. Figure 1.4c shows how selective dynamic scheduling is introduced for our running example from Listing 1.1. Most of the operators in our circuit are still scheduled statically, allowing for greater reuse of functional units. And most importantly, the critical path of the circuit is not increased by our approach.

HLS compilers instantiate parameterizable IP blocks to implement the compiled code, e.g., the compiler might parameterize an LSQ based on the memory access pattern. As we will argue throughout the chapters, to make HLS work well on irregular code, this compiler-hardware interface can be exploited to a much greater extend than before. On the one hand, new IP blocks have to be created that deal better with the unpredictability of irregular codes. On the other hand, compiler analysis can be used to aggressively specialize the instantiated IP blocks to the input codes, improving important metrics like circuit area and critical path.

1.3 Thesis Statement & Contributions

Compiler-hardware co-design enables a more efficient HLS of irregular codes. Instead of choosing between two extremes—everything scheduled either statically or dynamically—the compiler should automatically choose where to introduce dynamic scheduling in an FSMD-based circuit based on compile-time information. The foundation of our approach is automatic program slicing, which enables a decoupled execution of the program, and which, together with a latency-insensitive communication protocol, is a way to introduce selective dynamic behavior. Using the foundation of decoupled execution, we propose speculative, dynamic out-of-order memory interfaces in FSMD HLS, which the compiler can automatically specialize for each program to achieve improved throughput, critical path, and area efficiency. We also show that decoupled execution and a close compiler-hardware co-design enables new sources of parallelism by proposing dynamic loop fusion.

We make the following research contributions:

- In **Chapter 3**, we introduce selective dynamic scheduling into FSMD HLS. We show that classical static scheduling algorithms can be extended to recursively decouple an FSMD into multiple components, such that each component can be scheduled without having to over-approximate terms in the modulo scheduling equations. We show that that by connecting individual FSMD components using latency-insensitive channels we can effectively introduce the same dynamic behavior as is enabled in the DDF HLS model. This work was published in [213].
- In **Chapter 4**, we focus on the problem of dynamically scheduling memory accesses. We show that by using decoupled execution, FSMD HLS can be extended with a Load Store Queue (LSQ) to disambiguate memory accesses at runtime, and that the LSQ can be significantly optimized for a given code using information from a compiler analysis. This work was published in [212] and [217].

- In **Chapter 5**, we introduce speculative memory requests into FSMD HLS. We show that dynamic memory scheduling depends on the ability of the address generating instructions to be decoupled from of the rest of the program, and we show how the compiler can check if such decoupling is possible. Our speculation work restores address generation decoupling in a class of codes that could not benefit from an LSQ without our speculation. We also discuss how our compiler speculation support can be used in the domain of CPU and GPU prefetchers, and Coarse Grained Reconfigurable Arrays (CGRAs). This work was published in [215].
- In **Chapter 6**, we introduce the novel concept of dynamic loop fusion, a compiler-hardware co-design that enables the fusion of loops at runtime, even if they contain unpredictable memory dependencies. Such fusion introduces a new source of parallelism not exploited before, and is enabled by a new program-order schedule that can be parallelized and constructed efficiently in hardware; and by exploiting the fact that most irregular memory accesses form monotonically non-decreasing functions. This work was published in [216].

Before describing our technical contributions, in **Chapter 2** we give some background about and survey related work in the area of compilers, parallel programming, hardware acceleration, and HLS.

Chapter 2

Background & Related Work

This chapter provides general background on optimizing compilers, hardware design, FPGAs, HLS, and FPGA programming models. The technical chapters that follow later include background and related work that is closer to our technical contributions. The intention behind taking a broader perspective in this chapter is to give the reader more context behind the motivation of our thesis.

Readers familiar with the basics of compiler design or computer architecture can skip those sections; we start with the basics because not every person familiar with compilers is familiar with computer architecture, and vice versa. However, we only describe concepts that are crucial to our later technical contributions.

2.1 Compiler Preliminaries

We start with introducing essential compiler terminology, which we use extensively in our technical chapters.

2.1.1 Static Single Assignment Form

All compiler analyses and transformations proposed in this thesis operate on code in Static Single Assignment (SSA) form [194]. We use the LLVM compiler framework [142] as an implementation vehicle. We assume that the program consists of a single, fully inlined function—since in this thesis we are generating hardware, this is a typical and valid assumption.

SSA code implicitly encodes data dependencies through so called *def-use chains*. For instance, consider the following translation from C on the left to SSA form on the right:

```c
int x = input() + 1;
int y = sin(x) + cos(x);
```

```
%0 = call input
%1 = %0 + 1
%2 = call sin, %1
%3 = call cos, %1
%4 = add %2, %3
```

By convention, SSA register names start with a `%`[1]. A given SSA register can only be defined (or assigned) once. This single assignment policy makes many common compiler optimizations and analyses trivial, which is important for the scalability of production compilers. For example, in our example above it is trivial to obtain all the data dependencies of the `%4 = add` instruction by walking up the def-use chain of the operands (e.g., starting at operand `%2` we obtain the dependencies `%1` and `%0`). We use such def-use chain walks extensively in this thesis.

2.1.2 Control Flow in Static Single Assignment Form

Branch instructions cause code in SSA from to be divided into *basic blocks*. A basic block is a sequence of instructions terminated by a branch instruction, i.e., a branch instruction can only appear at the end of a basic block. An unconditional branch specifies the successor basic block to jump to; a conditional branch can specify multiple successor blocks out of which one will be selected at runtime depending on the branch condition value. A basic block with just one instruction is possible, with the instruction necessarily being a branch.

By specifying that a basic block is a graph node and that a block branch instruction encodes a graph edge from the contained basic block to a successor block we obtain the Control Flow Graph (CFG)—a graph representation of the program control flow. Figure 2.1 shows an example CFG for a simple C code reduction loop. In LLVM IR, distinct control flow constructs like loops, *if*-conditions, breaks, and continues loose meaning, all taking the form of branches. An additional control flow analysis is typically used to rediscover loop constructs [194].

Values which depend on control flow are represented using ϕ-nodes. For example, our Figure 2.1 example uses "`next_sum = phi [if_sum, ifTrueBB] [sum, bodyBB]`" to choose between the sum value coming out of the *if*-condition body block and the not changed sum value. A ϕ-node in $block_j$ takes a list of N $(value_i, block_i)$ pairs, where $N > 1, 0 \leq i < N$, and returns $value_i$ iff $block_i$ was the immediate predecessor of $block_j$ on the current CFG path. By the "current CFG path" we mean that this selection occurs dynamically, at runtime. The actual implementation of the control-flow selection is implementation dependent—for a CPU, the compiler would typically emit register conditional move instructions; in the context of hardware synthesis, a multiplexer unit is typically used, which selects between alternative values depending on a predicate value.

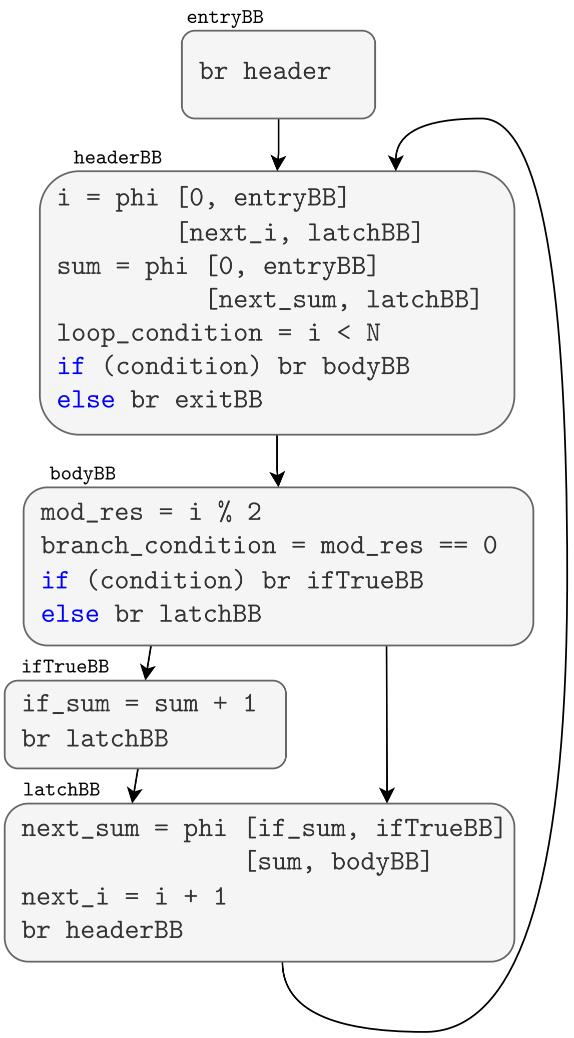
ϕ-nodes implementing loop recurrences cause cycles in the def-use chain, e.g., the `i`, `next_i` def-use chain forms a cycle in Figure 2.1. All values which are needed to calculate a given recurrence belong to the same Strongly Connected Component (SCC) in the def-use graph. Similarly, loop backedges cause cycles in the CFG, e.g., the `latchBB` to `headerBB` edge. Many graph operations, like topological ordering, are not possible on graphs with cycles, even

1. For readability, we deviate from this convention in the rest of this thesis. We will also use an infix notation for binary operators, instead of the standard Polish notation, and we will avoid instructions mnemonics; e.g., we will write "`add %1, %2`" as "`some_name_1 + some_name_2`"

```
int sum = 0;
for (int i = 0; i < N; ++i)
    if (i % 2 == 0)
        sum += 1;
```

```
entryBB:
    br header
headerBB:
    i = phi [0, entryBB]
            [next_i, latchBB]
    sum = phi [0, entryBB]
              [next_sum, latchBB]
    loop_condition = i < N
    if (condition) br bodyBB
    else br exitBB
bodyBB:
    mod_res = i % 2
    branch_condition = mod_res == 0
    if (condition) br ifTrueBB
    else br latchBB
ifTrueBB:
    if_sum = sum + 1
    br latchBB
latchBB:
    next_sum = phi [if_sum, ifTrueBB]
                   [sum, bodyBB]
    next_i = i + 1
    br headerBB
```

(a) C code and its SSA form.



(b) CFG.

Figure 2.1: Example of a CFG obtained from the SSA form of the original C code.

though they might be desirable in optimization passes. A common technique to sidestep this issue is to decouple the CFG or def-use chain into subcomponents. For example, by removing loop backedges from a CFG sub-region representing a loop we can obtain a topological ordering of the basic blocks for that loop since the remaining blocks form a Directed Acyclic Graph (DAG). Then separately, at the level whole loops, we can obtain a topological ordering of the loop forest.

Other SSA Control Flow Representations

Using ϕ-nodes is not the only approach for implementing control flow. **Gated SSA (GSA)** uses three different nodes to represent control flow [174, 109]:

1. The $r \leftarrow \mu(init, caried)$ node replaces ϕ instructions in loop headers and represents loop carried dependencies whose $init$ value comes from the loop predecessor and subsequent $carried$ values come from inside the loop.
2. The $r \leftarrow \eta(p, v)$ instruction is used to gate the use of v until the p predicate holds. Its intended use is to signal that values calculated in loops (e.g., a reduction value) are ready to use outside the loop.
3. The $r \leftarrow \gamma((p_0, v_0), (p_1, v_1), ...)$ selects a value v_i if the predicate p_i is true—it transforms control flow into data flow in ordinary basic block structures generated from *if*-conditions, similar to *if*-conversion [3].

As of today, the GSA form is not implemented in any commercial compiler, with maybe the exception of the μ instruction (e.g., the LLVM IR has `select` instructions and passes to turn some ϕ nodes into `select`'s). In practice, it is easy to check in which of the three GSA node categories a given ϕ-node falls.

It is also possible to represent control flow without any SSA pseudo-instructions. For example, the **MLIR** compiler framework [143] represents basic blocks as functions with arguments. Control flow is implemented by branching to a successor block and providing a list of arguments to carry forward, making the correspondence between SSA and continuation-passing style more explicit [131, 25]. In addition, basic blocks can be generalized into typed regions, like loops or *if*-statements, that optionally return values, thus providing the same type of functionality as the original GSA proposal [174] without the need for additional analyses to rediscover these structures.

In addition to simplifying the SSA representation compared to LLVM, MLIR also offers powerful abstractions for user-defined intermediate representations that allow for more precise problem domain modeling. This approach is especially promising for the hardware design domain, where tasks like pipelining, timing optimizations, placement, or verification benefit from a task-specific representation [91].

The HLS tools that we use in this thesis use LLVM, not MLIR, and we will use LLVM terminology in the rest of the thesis. New compilers increasingly choose to use MLIR [81], and porting efforts are underway to leverage MLIR in existing projects [222].

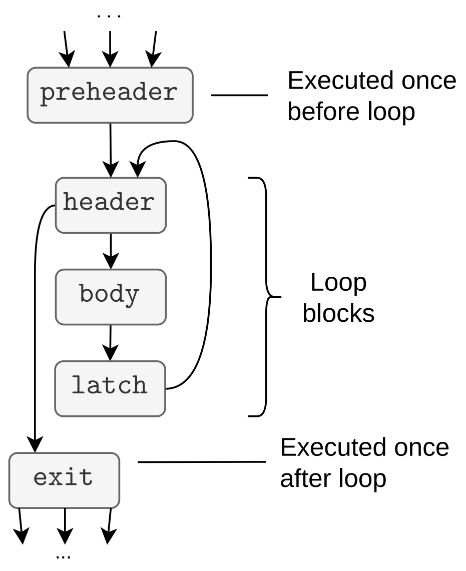


Figure 2.2: Our canonical loop representation. The body and latch blocks are allowed to be the same block.

2.1.3 Loop Terminology

We use a standard loop definition. In a CFG, a loop is a maximal SCC with at least one internal edge, such that there exists one entry block, called the loop *header* block, that dominates all other blocks in the SCC—all incoming edges outside of the SCC to the inside of the SCC point to this header entry block [106]. A basic block A dominates another block B if every CFG path from the function entry to the function exit that reaches block B goes through block A first—block A is guaranteed to have executed on the current CFG path if block B executes.

Canonical Loop Form

To simplify our compiler analyses and transformations, we use a canonical loop representation. Our loop header can only be reached by a loop *preheader*. Our loops have a single backedge going from the loop *latch* block to the loop header block—loops with `continue` or `break`, which otherwise would have multiple backedges, are transformed by inserting new basic blocks that jump to a single loop latch. Also, we have a single loop exit block, which is reached from the loop header. Figure 2.2 shows this canonical loop representation, which closely follows the LLVM loop simplify canonical form [92].

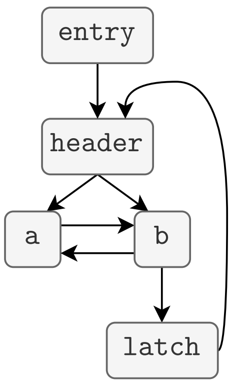
Irreducible Control Flow

We restrict most our analysis' and transformations to reducible control flow. Some of our transformation algorithms (e.g., in Chapter 5) would not work on codes with irreducible control flow. In practice, this is not a big restriction since most code has reducible control flow, and irreducible control flow can be transformed to have reducible control flow [18, 182]. For completeness, we define irreducible control flow.

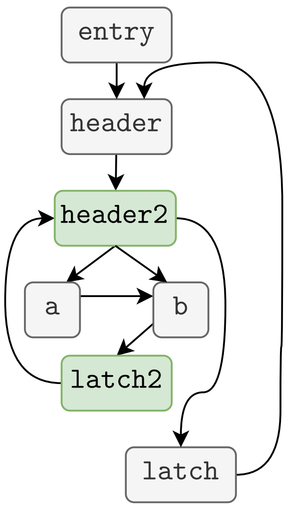
```
for (i = 0; i < N; ++i) {
    int j = 0;
    if (i % 2 )
        goto label_a;
    else
        goto label_b;

    label_a:
        j++;
        goto label_b;
    label_b:
        if (j < N)
            goto label_a;
}
```

(a) C code with irreducible control flow.



(b) The corresponding irreducible CFG.

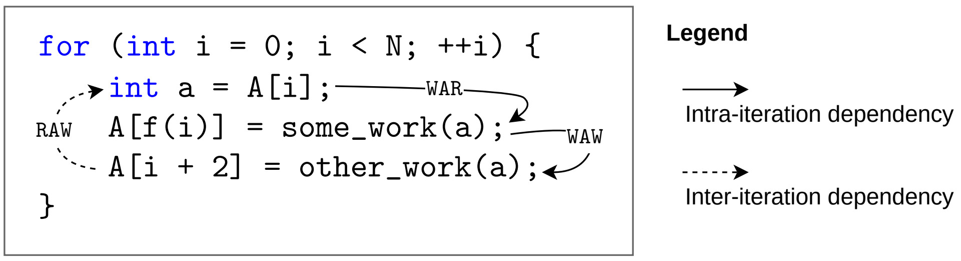


(c) A transformed CFG with reducible control flow.

Figure 2.3: Example of a code and CFG with irreducible control flow. Every irreducible CFG can be transformed to a reducible CFG by inserting new basic blocks [18].

A maximal SCC with multiple entry blocks is not a loop. For example, the CFG in Figure 2.3 has the SCC consisting of blocks $\{a, b\}$. This is not a loop since both blocks a and b are entry blocks. On the other hand, the SCC $\{header, a, b, latch\}$ is a loop, since only the $header$ is an entry block. A CFG is said to be *irreducible* if it contains a cycle (an SCC with at least one internal edge) that is not a loop, like the a, b cycle in Figure 2.3. Irreducible control flow in C-like languages can only arise when a `goto` with a backward label is used.

A common misconception is that accelerators like GPUs or FPGAs fundamentally cannot deal with irreducible control flow. This is not true. Clearly the algorithms that implement predicated control flow in GPUs [148, 78] could be extended to deal with non-loop cycles. Similarly, an FSM generation algorithm that implements control flow in FPGA accelerators could deal with arbitrary `goto` [137]. It is more precise to say that popular accelerator programming models do not accept `goto` keywords (and thus irreducible CFGs) out of pragmatic reasons—the virtual non-existence of `goto` in computationally intensive loop-based codes makes the additional code needed to handle them in the compiler not worth it. In practice, codes for high-performance computers do not use `goto`. Furthermore, any irreducible CFG can be transformed into a reducible CFG by inserting new basic blocks to turn non-loop cycles into loops [18, 182].



```
for (int i = 0; i < N; ++i) {
  int a = A[i];
  A[f(i)] = some_work(a);
  A[i + 2] = other_work(a);
}
```

Figure 2.4: An example of Read After Write (RAW), Write After Read (WAR), and Write After Write (WAW) dependencies. The arrows go from a dependency source to a dependency destination (also called sink). We distinguish between intra-iteration and inter-iteration dependencies. Inter-iteration dependencies traverse a loop backedge on the path from dependency source to dependency destination. (Not all dependencies are highlighted.)

2.1.4 Dependency Graphs

The SSA def-use chain implicitly encodes what we call scalar (or register) dependencies. In practice, this representation is not enough in optimizing compilers, because it does not include memory dependencies. The execution model of C-like languages guarantees a certain program order of memory loads and stores. To enable parallelism, the compiler can relax this ordering if it can prove that changing the order of memory operations does not change the meaning of the program. To prove the correctness of such reordering, the compiler needs to reason about memory dependencies.

Figure 2.4 shows an example of the type of memory dependencies considered by the compiler. SSA form, by definition, only contains scalar (Read After Write) RAW dependencies—Write AFter Write (WAW) and Write After Read (WAR) scalar dependencies are not possible in SSA. However, when we are dealing with dependencies through memory, WAR and WAW dependencies are possible and have to be considered by the compiler.

Dependency Distance

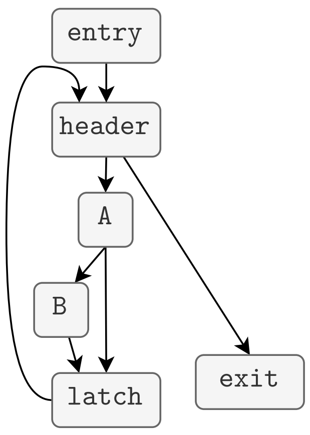
With memory dependencies contained in loops it is useful to talk about the dependency distance—how many iterations pass between the dependency source definition and the destination consumption. Scalar dependencies in SSA form can only have a dependency distance of 0 (used in the same loop iteration) or 1 (used in the next iteration). Memory dependencies, where the memory location is a function of the loop iterators and constant program parameters,[2] can be analyzed to determine the dependency distance [79, 16, 80, 184]. For example, the RAW dependency highlighted in Figure 2.4 has a dependency distance of 2.

If the compiler cannot reason about any of address expressions in a given memory dependency, then we say that the dependency distance is unknown. The compiler has to over-approximate unknown dependency distances to 1. For example, if the compiler cannot analyze the `f(i)` address expression in Figure 2.4, then it has to assume that the `f(i)` address might be used a store or load in the in the next iteration.

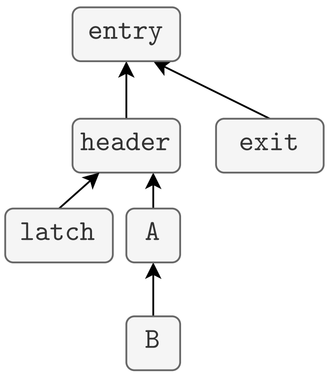
2. A constant parameter is a value that does not change during the loop execution; its value does not have to be known at compile time.

```c
// Header block
for (i = 0; i < N; ++i) {
    // A block
    if (i % 2 == 0) {
        // B block
        work();
    }
}
```

(a) C code.



(b) CFG.



(c) CDG.

Figure 2.5: Control dependency example. Block *B* is control dependent on the *A* block and on the *header* block. Given a Control Flow Graph (CFG), we can generate a Control Dependency Graph (CDG) to quickly query control dependency relations.

Data Dependency Graph

By combining the SSA def-use chain with a memory dependency analysis, we can construct a Data Dependency Graph (DDG). In a DDG, nodes represent SSA values (including load and store instructions) and typed edges represent different types of dependencies (scalar, memory) together with dependency information (RAW, WAR, WAW, dependency distance).

Note that a Data Flow Graph (DFG) is not the same as a DDG. A DFG is the graph that arises from the SSA def-use chain; a DFG does not include memory dependencies, while a DDG does.

Loop Initiation Interval

A DDG can be used to calculate the loop II—a number dictating how many cycles need to pass between the start of subsequent loop iterations. This is achieved by looking at the cycles (the SCCs) in the loop DDG, and dividing the latency of the instructions in the cycle by dependency distance of cycle result. This calculation will be described in more detail in Chapter 3.

Control Dependency

Given a CFG, we can determine if a basic block Y is control dependent on another block X. Control dependencies are intuitive to understand—the branch in block X determines if block Y will execute—but the formal definition is a bit more elusive. We use the definition based on the post-dominance relation proposed by Ferrante *et al.* [84].

First, we define the post-dominance relation. Given a CFG, a basic block Y post-dominates block X iff every CFG path from X to the CFG exit block goes through Y (if a CFG has a single exit node, then the post-dominance relation is equivalent to the dominance relation in the reversed CFG, where the direction of every CFG edge is reversed). For example, in Figure 2.5 the exit node post-dominates all the other nodes; the latch node post-dominates blocks A and B. Every basic block trivially post-dominates itself (strict dominance and post-dominance provide irreflexive versions of the definitions).

We now give the definition of control dependency. It helps to have the example of blocks B and A from Figure 2.5 in mind when reading the definition. Basic block Y is control dependent on block $X, X \neq Y$ iff:

1. Block Y does not post-dominate block X.
2. There exists a CFG path P from X to Y such that for all $Z \in P, Z \neq X, Z \neq Y$ Y post-dominates Z.

In our Figure 2.5 example, B is control dependent on block A.

Control Dependency Graph

Altough control dependencies can be calculated "on the fly" given a CFG, in practice it is common to pre-calculate all program control dependencies and store them in a graph data structure, called the Control Dependency Graph (CDG), for quick querying. Nodes in a CDG represent basic blocks and directed (A, B) edges encode that block A is an immediate control dependency source of block B.[3] Out of the set $\mathbb{A}$ of all block A control dependencies, a block B_{imm} is the immediate control dependency block of A iff B_{imm} is control dependent on all blocks in $\mathbb{A} \setminus B_{imm}$. By definition, a block can only have one immediate control dependency. For example, in Figure 2.5 block A is the immediate control dependency source of block B.

We use the CDG especially in Chapter 5 to calculate source blocks of control dependencies.

Program Dependency Graph

A Program Dependency Graph (PDG) is a graph that combines a DDG and CDG [84]. It can be implemented by adding a control dependency type edge to the set of existing DDG edge types. Representing both data and control dependencies in a single graphs makes expressing many compiler analyses and transformations easier, including the ones in this thesis. In practical implementations, including ours, a PDG is an abstraction layer on top of a DDG and CDG, which are stored separately.

3. Some papers reverse the direction of edges such that the CDG forms a tree with the program entry block as a root. The direction of edges does not matter as long as it is used consistently.

2.2 Computer Architecture

In this section, we give a brief overview of the different approaches to architect a programmable computer, motivating the need for ASICs and FPGAs as devices that trade off programmability for increased performance and energy efficiency. We divide computer architectures into temporal architectures—instructions are stored in memory, fetched, decoded, executed, and the results are committed to memory—and spatial architectures—the program is implemented by a set of Processing Elements (PEs) that communicate in a producer-consumer fashion. Temporal architectures are globally controlled, e.g., by using the Program Counter (PC). Spatial architectures have distributed control, where the execution of each PE is only determined by its inputs.

On paper, the global control and the fetch/decode/execute cycle through memory of temporal architectures seem like a clear Instruction Level Parallelism (ILP) overhead (often called "the on Neumann bottleneck" [17]). In practice, and for good reasons, no practical computer system today is purely temporal or spatial. The goal of computer architects is to find the right balance between the different approaches such that the resulting system meets certain criteria. For example, superscalar CPUs, Very Large Instruction Word (VLIW) CPUs, and GPUs contain multiple functional units to which instructions must to be mapped, either at runtime or compile time. Spatial architectures, like Coarse Grained Reconfigurable Architectures (CGRAs) or FPGAs, need to temporally share functional units in order to implement programs that would otherwise require more functional units than the machine provides, and often have localized data and code stores to save communication bandwidth. Nonetheless, it is still useful to use the temporal and spatial taxonomy when characterizing modern computer systems because the two approaches tackle the von Neumann bottleneck in different ways.

2.2.1 Temporal Architectures

CPUs and GPUs amortize the von Neumann overhead by, among other techniques, using vector execution, following in the footsteps of vector machines like the Cray-1 used in supercomputers [82]. A vector instruction can be used to apply the same operation to multiple scalar elements. In GPUs, vector execution is combined with coalesced memory accesses over very wide memory busses to further alleviate the von Neumann bottleneck. Recently, both CPUs [1] and GPUs [172] also provide matrix execution engines, which further alleviate the fetch/decode/execute bottleneck by increasing the number of operations performed per accessed memory byte.

From a programmer's point of view, CPUs are globally controlled by a PC, enabling debuggability and precise exceptions [107]. These are essential features for general purpose computing and operating systems. In many of the actual high-performance Out-of-Order (OoO) CPU implementations, great effort is exerted to relax global control and enable more ILP. Multiple instructions within a given instruction window are fetched, decoded, and executed in advance in dataflow order—as soon as their dependencies are satisfied and a suitable functional unit is available, even before previous instructions in program order have executed. Memory dependencies are tracked dynamically in a Load-Store Queue (LSQ) structure. Speculatively executing instructions without waiting for control-flow decisions can further increase the ILP, with sophisticated branch predictors deciding which control flow code path to fetch. Executed instructions are moved to a hardware structure responsible to commit the instructions in the original program order; mis-speculated instructions are detected, causing the core to revert to a previous checkpoint. In GPUs, the ordering of instructions is only guaranteed at a coarser-grained level, allowing GPU schedulers to keep the vector datapath occupied with later instructions, if instructions earlier in program order stall. In our thesis, we will adapt some of the techniques used to increase ILP in CPUs and GPUs to the context of HLS.

The Memory Wall

One issue that vector processing, and the recently added matrix processing, do not solve in temporal architectures is the contention to a single memory interface. Memory throughput has been steadily increasing over the decades, but not nearly at the same speed as the computational throughput [157]. Caches and prefetching are used to alleviate this bottleneck, with many research efforts in this direction still ongoing [163, 9, 59, 42, 104, 162]. The scalability of the register file has also been classically addressed with techniques like architectural registers, register renaming, hierarchical register files, or functional unit queues (allocation stations) [195]. The problem is that the memory technology implementing register files has a limited number of read/write ports, which, together with the dataflow tagging mechanism used to broadcast functional unit results, become the bottleneck as the number of functional units is increased in superscalar CPUs.

Multi-threaded GPUs face a similar issue in that a single core in a GPU might be executing tens of threads in a temporal fashion, requiring the GPU to hold the contexts of all threads in the register file. As a result, register files in GPUs are even larger, reaching hundreds of kilobytes and consuming tens of Watts [95]. Many research efforts try to minimize the need for such large register files, e.g., by sharing registers between threads whose lifetimes do not intersect [122], or by compressing the register file contents [144]. On the other hand, the vector (called warp by NVIDIA; wavefront by AMD; sub-group by the others) execution model allows for easy multi-porting of the register file, making many of the CPU techniques to make port sharing efficient redundant.

Decoupled Access/Execute Architectures

The Decoupled Access/Execute (DAE) technique is a popular approach to the memory wall problem. The idea behind DAE is to let the memory access instructions run ahead of the main program, thus alleviating any memory latency issues [205]. The idea is general and is used in many computational models: it is used in specialized FPGA accelerators generated from HLS [42, 41, 51, 54, 44, 89, 212, 216]; in CGRAs [166, 188, 187, 83, 111, 180, 233, 171]; and in CPU/GPU prefetchers [60, 167, 104, 9, 6, 59, 190]. The common denominator of all these works is that they rely on either the programmer or the compiler to decouple address-generating instructions from the rest of the program into an Address Generation Unit (AGU). The AGU sends load and store requests to the Data Unit (DU), while the DU sends load values to and receives store values from the Compute Unit (CU). All communication is FIFO based and ideally the AGU to DU communication is feed-forward (one-directional), allowing the address streams from the AGU to run ahead w.r.t the CU.

We use the DAE technique throughout the thesis, developing compiler analysis and transformation passes to automatically construct DAE architectures, and contributing a new DAE speculation technique in Chapter 5. In chapters 3 and 4, we refer to the DU as an LSQ; in chapters 5 and 6 we use the general DU term.

Static vs. Dynamic Scheduling in CPUs

We mentioned that a CPU can have multiple functional units that can all execute concurrently, increasing the ILP. Instructions can be mapped to the functional units either statically by the compiler (VLIW CPU) [86], or dynamically at runtime by a hardware scheduler (OoO CPU) [195]. A VLIW instruction encodes how each of the multiple functional units in the machine should execute. The contrast between these two approaches is the same as between Finite State Machine with Datapath (FSMD) HLS and Dynamic Dataflow (DDF) HLS, an observation made by many researchers working in the field. Not surprisingly, VLIW cores are successful in domains where dependencies can be determined at compile time (e.g., Digital Signal Processing (DSP), many AI computations), while OoO cores dominate general purpose computing, which contains more irregular code [147]. The notable effort to bring VLIW to general purpose computing in the form of the Intel Itanium architecture has failed, because, as extensively discussed in this thesis, a compiler cannot efficiently schedule code without having full information about its dependencies [234].

2.2.2 Spatial Architectures

The von Neumann bottleneck of temporal architectures was anticipated and described several decades ago and research quickly began on computer architectures with more distributed control that would allow for more parallelism. The idea was to lay down computation spatially and stream data directly between PEs, rather than stream instructions through a shared fetch, decode, and execute pipeline.

Dataflow computers researched in the 1970s and 1980s were arguably the first spatial architectures [11, 66, 102]. PEs in spatial dataflow architectures consume inputs and produce outputs for other PEs following a producer-consumer communication pattern that avoids the overhead of shared memory and control. In such an execution model, the machine configuration closely matches the DFG of the code that is being executed. Altough many efforts were made to commercialize these ideas in the form of a dataflow architecture, e.g., [177, 35, 208, 71], their success in the market was limited due to the difficulty to efficiently compile codes with complex control flow and data structures [32]. Also, since architectural state in a pure dataflow machine is distributed among all participating PEs, providing precise exceptions and a friendly debugging experience was also challenging—a common saying was that the machines provided *too much* parallelism—which meant that the machines were only used in specialized domains like signal processing. However, many of the ideas developed for dataflow architectures have eventually found their way into OoO CPU microarchitectures.

Coarse-Grained Reconfigurable Arrays

CGRA accelerators are a modern incarnation of spatial dataflow architectures [171, 179, 171, 113, 233, 166, 179, 65]. A typical CGRA consists of a grid of reconfigurable PEs interconnected via a network on chip. They closely follow the original dataflow execution model proposed in the 1970s [66], but their role as an accelerator liberates them from the requirements of general purpose computing, like precise exceptions. Additionally, decades of research of mapping sequential code to a spatial grid of PEs has improved the quality of results of CGRA compilers [156]. Today, several commercially successful CGRAs are available: from mW edge processing [75] to HPC and DSP acceleration [165, 94, 55].

Such design point flexibility of CGRAs is due to the the many design choices that can be made: one can customize the communication network and PE architecture, have shared PEs or not, make reconfiguration latency trade-offs, etc. Based on the design choices, a CGRA can be grouped into one of four categories [232]: systolic (statically scheduled with non-shared PEs), shared-systolic (statically scheduled with shared PEs), tagged-dataflow (dynamically scheduled with shared PEs), and ordered-dataflow (dynamically scheduled with non-shared PEs). A shared PE is time-multiplexed among different stages of the executed program. The scheduling and mapping problem in CGRAs is similar to scheduling and binding in HLS, with many similarities in the algorithms that solve these problems. Accordingly, some of the ideas developed in this thesis are directly applicable to CGRAs, which we highlight throughout the chapters.

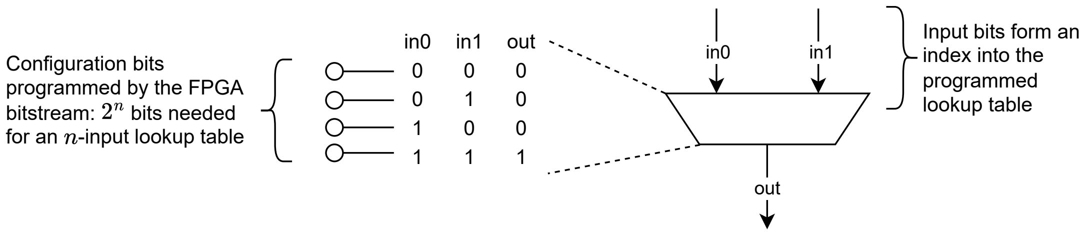

Configuration bits programmed by the FPGA bitstream: 2^n bits needed for an n-input lookup table
in0 in1 out
0 0 0
0 1 0
1 0 0
1 1 1
in0
in1
out
Input bits form an index into the programmed lookup table

Figure 2.6: An illustration of a simple FPGA lookup table capable of implementing any binary logic function. Lookup tables found in commercial FPGAs typically have 4, 6, or 8 input bits, and include additional registers whose value can be selected to drive the `out` wire instead of the lookup table result, e.g., reference [114] describes the lookup table used in Altera FPGA devices.

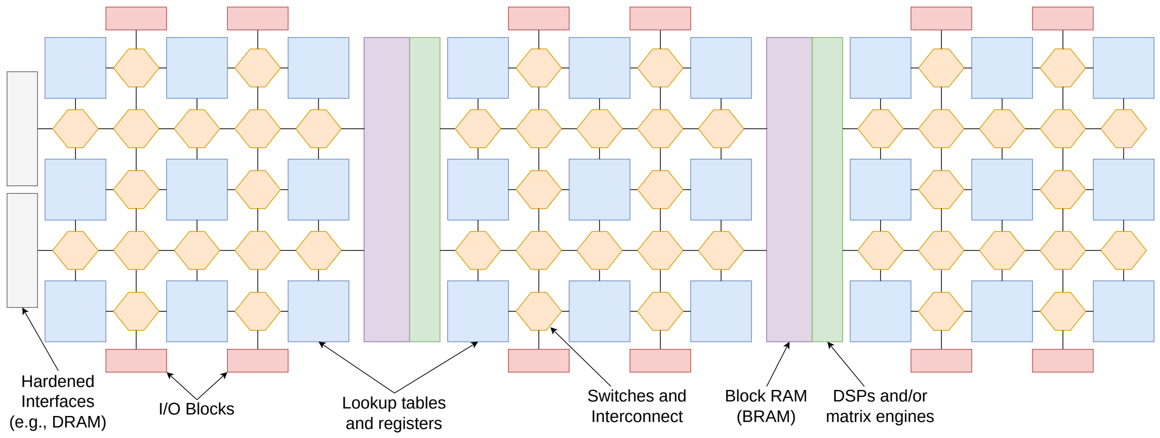

Hardened Interfaces (e.g., DRAM)
I/O Blocks
Lookup tables and registers
Switches and Interconnect
Block RAM (BRAM)
DSPs and/or matrix engines

Figure 2.7: Modern FPGAs consist of heterogeneous compute and storage structures. Diagram adapted from [130].

Field-Programmable Gate Arrays

Traditionally, FPGAs consisted of a sea of configurable logic blocks connected via programmable switches [31]. The logic blocks are typically made up of a lookup table and a register, and can be configured to implement any logical operation.[4] Figure 2.6 shows an example lookup table whose configuration is set to implement a logical AND function. Their configurability meant that FPGAs were used as fast simulators for other chips under development. Their high-bandwidth and low latency IO, and sub-word bit manipulation capability has also made them popular for networking and various sensing applications [159, 130].

In the last decade, as interest in FPGAs for data center and AI application has risen. FPGA vendors added hardened ALU structures, often called DSP blocks, to their high-end FPGA devices. With a DSP block, a multiplication (and similar) operation can be executed faster and more efficiently compared to implementing the same operation using lookup tables. Recent FPGAs also contain hardened matrix multiplication units. Today, an FPGA is a sea of *heterogeneous* compute and memory structures, as illustrated in Figure 2.7.

4. Interestingly, the AVX512 ISA designers were allegedly inspired by this functionality when creating the `VPTERNLOGD` ternary logic instruction [90].

2.2.3 Digital Circuits

In this brief subsection, we describe common digital circuits that we use throughout the thesis.

Shift Register

A shift register is a very efficient hardware storage structure, especially on FPGAs. An example shift register is shown in Figure 2.8. The short wires between the registers enable dense packing and a short delay, increasing area efficiency and maximal frequency of the circuit. However, the number of registers needs to be kept small for the shift register to be able to be packed densely on the FPGA chip (modern FPGAs can handle shift registers a few kilobits in size). Multiple tab points and conditional shifting further decrease the possible shift register size, since more area needs to be allocated to implement these features. Nonetheless, if a given function can be implemented in shift registers, this is usually the most efficient approach on FPGAs. We use shift registers in Chapter 4 to make our LSQ more efficient.

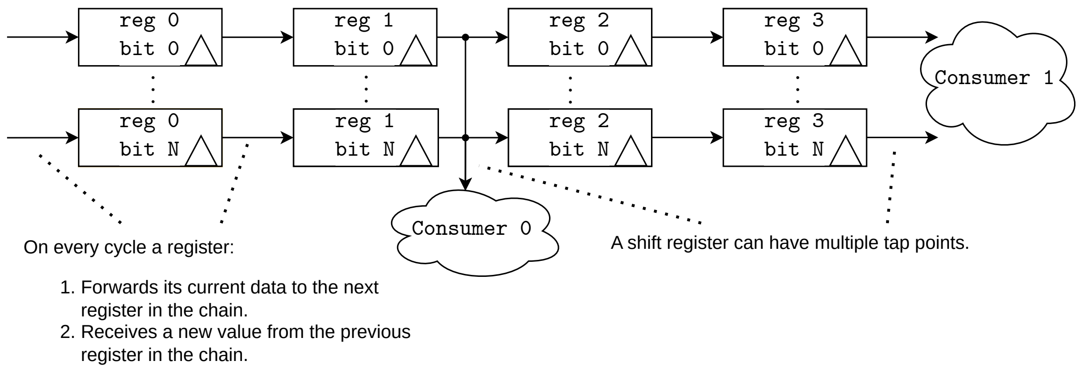
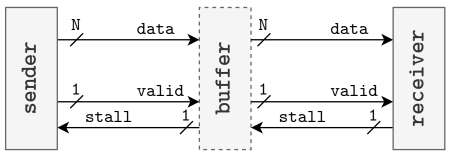

reg 0
bit 0
reg 1
bit 0
reg 2
bit 0
reg 3
bit 0
Consumer 1
reg 0
bit N
reg 1
bit N
reg 2
bit N
reg 3
bit N
Consumer 0
On every cycle a register:
1. Forwards its current data to the next register in the chain.
2. Receives a new value from the previous register in the chain.
A shift register can have multiple tap points.

Figure 2.8: An example 4 element N-bit shift register with two tap points. More advanced shift register architectures can contain more tap points, condition shifting, and multiplexers before the tap points.

Latency-Insensitive Channel

In shift registers, the communication between subsequent registers is done using wires. A wire that is driven (written to) on a cycle n delivers its value to the destination on the same cycle n. In any clock cycle after n, the value is lost. Hence, communicating with wires requires careful cycle level reasoning between the sender and receiver, a requirement that becomes increasingly more difficult to satisfy as the complexity of the design grows. Latency-insensitive channels [87, 39] solve this issue by decoupling the communication protocol from the cycle level behavior of the sender and receiver. As Figure 2.9 demonstrates, by using additional wires and storage structures, the sending and receiving of data becomes governed by the dataflow at runtime. We use latency-insensitive channels throughout the thesis.



Example Protocol:

`valid ∧ ¬stall`
The sender provides valid data, and the receiver accepts it.

`¬valid`
The sender does not provide valid data.

`valid ∧ stall`
The sender provides valid data, but the receiver does not accept it.

Figure 2.9: An example latency-insensitive channel implementation. The communication between the sender and receiver does not depend on their cycle behavior, but on the readiness of the receiver and the validity of data. More sophisticated protocols are used in practice to decrease the critical path of the logic.

We use latency-insensitive channels throughout the thesis to implement communication between separate modulo scheduling instances. We use the SYCL heterogeneous programming model in this thesis, where the latency-insensitive channel construct is provided in the form of **SYCL pipes**. Most HLS programming models provide similar constructs.

FPGA Block RAM

Most FPGAs contain on-chip embedded memories, typically refereed to as BRAMs. A single BRAM cell typically has a capacity of 5–20 kilobytes and can serve two memory requests per cycle (at most one of the requests can be a store). The load requests have to be typically pipelined, with the load address arriving 1–3 cycles before the load response is served, otherwise the circuit critical path may be impacted. A high-end FPGA can contain several thousands of BRAMs, resulting in tens of megabytes of on-chip storage capacity. The interesting thing about BRAMs is that, since each out of the thousands of BRAM cells has separate ports, they can provide tremendous memory throughput if the accelerated algorithm lends itself to trivial memory partitioning.[5] Such memory partitioning of on-chip BRAMs is one of the most important steps in high-performance HLS designs. We make extensive use of BRAMs in our LSQ described in Chapter 4.

2.3 Spatial Computing Programming Model

One of the reasons why early dataflow computers [11, 66, 102] did not find widespread adoption was their programming difficulty. As noted in the previous section, and studied by others in the literature [32], dataflow architectures contain so much implicit parallelism that it is difficult for the programmer to reason about machine state. Programming languages that gave explicit control over the machine parallelism failed to gain adoption [123]. Most software

5. For example, assume that a design reads one 64-bit value from a BRAM cell per cycle. A 250MHz design that uses 1000 such BRAM cells results in a memory bandwidth of 2 terabytes.

programmers preferred to reason about their program sequentially and let the compiler or machine discover parallelism for them. This automatic approach continues to this day, with CGRA and FPGA compilers using sophisticated transformations to map a fundamentally sequential user program to the underlying parallel architecture.

On the other hand, hardware programmers have been dealing with parallelism, concurrency, and explicit spatial architectures for decades, a fact reflected in the programming languages used in hardware design. Traditional Hardware Description Languages (HDLs) like Verilog, System Verilog, and VHDL, are structural languages that describe the state and state transition of every wire and register on every clock cycle. Parallelism is implicit with every operation executing on every clock cycle unless otherwise stated, and any concurrency issues have to be managed manually.

Today, advances in making spatial programming easier to use are made from two communities. The HLS community tries to automatically discover parallelism from a sequential program, while the HDL community tries to incorporate features from software languages to make the already parallel HDLs more productive (these new HDL languages are often called High-Level Hardware Description Languages (HHDLs)). We believe that both approaches are needed, since HLS and HDLs serve two slightly different goals. Altough both ultimately program a spatial architecture, HLS targets loop-heavy programs with a high compute intensity, while HDLs are typically used to generate more complex control programs with very little feed-forward dataflow.

Our thesis is that the compiler algorithms used in HLS can be extended to also support some form of irregular control flow. Before describing the HLS background, we mention some notable HHDL efforts.

2.3.1 High-Level Hardware Description Languages

HHDLs add features from high-level software languages, like advanced type systems, polymorphism, or higher order functions, to make HDL programmers more productive. Most, if not all, HDDLs compile to Verilog or VHDL to ensure compatibility with existing IP and EDA tools. Importantly, they do not change the underlying computational model and they do not apply automatic parallelization. The programmer has explicit control over the hardware resources and has to apply techniques, like pipelining or operator sharing, manually.

HHDLs can be embedded in an existing programming language, or they can form a completely new language. For example, Chisel extends the Scala language with hardware specific libraries to generate hardware [203]. Cλlash [15] and Lava [27] do the same, but use Haskell and focus more on composing generators to produce hardware.

The Guarded Atomic Actions Computational Model of Bluespec

A notable HHDL is Bluespec [169, 30], because, in additional to providing productivity features from software languages, it offers more comprehensive semantics through the Guarded Atomic Actions (GAA) computational model, which makes reasoning about concurrency easier. In the GAA computational model, the programmer defines state elements, which in the case of hardware are registers, wires, FIFOs, etc., and atomic actions, which describe state transitions. The actions are guarded by side-effect free boolean expressions. The Bluespec compiler automatically derives a scheduler, which at runtime enables the maximum number of actions, and implements operator sharing, all while guaranteeing a “one action at a time” semantics. Additionally, the language provides a number of annotation options about the execution priority of actions to guide the compiler to the desired schedule.

The GAA approach provides a powerful abstraction to reason about concurrency and shared structures that, in practice, incurs no area or performance costs, since the logic generated by the Bluespec compiler is almost always the same as the logic used by Verilog or VHDL programmers to guard against concurrency bugs and to share resources [7]. The GAA model is rooted in sound computer science theory—other names for GAAs are Rewrite Rules or Term Rewriting Systems [134]. The Bluespec implementers took direct inspiration from formal specification languages that are based on GAAs, e.g., Dijkstra’s Guarded Commands [69] or Lamport’s Temporal Logic of Actions [141]. Thus, it comes at no surprise that the GAA model makes the formal verification of Bluespec hardware easier [64, 164].

2.3.2 High-Level Synthesis

HLS, as opposed to HDLs, allows programmers to use familiar sequential languages, such as C, to generate a hardware design [155].[6] The programmer does not need to reason about low-level hardware structures like memory controllers, functional units, or bus protocols. The compiler automatically generates the required HDL code, which implements the algorithmic description written in C. Traditionally, this compilation process has been divided into three steps: allocation, scheduling, and binding [58, 38].

Allocation determines which hardware resources are used to implement the function (e.g., the number of lookup tables, or ALUs). This decision is tightly coupled with the area, circuit frequency, and power requirements since different types of resources have different delays, area usages, and power characteristics. The scheduling step decides at which clock cycle a given operation executes. Allocation influences scheduling in the sense that a larger area budget gives more opportunities for pipelining by allowing intermediate state to be stored in registers between clock cycles. The binding step is typically executed after the scheduling step. It binds program operations to the available hardware resources and decides if multiple operations should share a resource.

6. Research efforts in HLS of functional programs also exist, although these attempts produce less optimized circuits at the moment [245].

Out of the three steps, scheduling is the most complex and has the largest impact on performance. We have briefly described the difference between static and dynamic scheduling in the previous chapter, however, since the FSMD resource-constrained static scheduling problem is NP-hard, there are multiple different algorithms that can be used to find a static schedule [56]. In this thesis, we focus on the scheduling step when trying to make the HLS of irregular codes more efficient.

Academic HLS Approaches

Most, if not all, commercially successful HLS tools use the C or C++ programming language, inheriting their sequential semantics [120, 4, 38, 189, 36].[7] Recent work on HLS in academia investigated other language frontends and execution semantics HLS tools.

Kanagawa, developed at Microsoft, is a new HLS language with explicit threads and similar semantics to CUDA, which the authors call Wavefront Threading [181]. In addition to language constructs aimed at hardware design, like pipelined loops, Kanagawa also offers language constructs for hardware thread synchronization, like atomic operations and wait barriers.

Dahlia is another new HLS language attempt that focuses on generating hardware with predictable latency [168]. This is achieved by translating the problem of calculating circuit latencies as an affine type inference problem, with the idea that circuits with unpredictable latency will fail at the type check stage.

The language called Spatial focuses on providing abstractions based on parallel patterns—`for-each`, `reduce`, `map-reduce`, `stream`, etc. [135].[8] Such a language makes the Spatial compiler much easier to write, since parallelism is explicitly expressed by the programmer.

Another project in the same vein is AnyHLS [176], which uses the AnyDSL [145] framework to create an HLS DSL in the functional language Impala. AnyHLS produces pragma-annotated C code accepted by vendor tools. However, these pragmas are not directly exposed to the programmer. Rather transformations like loop unrolling are first-class citizens implemented as libraries of partially evaluated functions that can be returned by other functions and composed together. Users can build on these primitives to provide highly abstract, domain-specific building blocks for FPGA accelerators.

Many academic works use a functional-style language as the HLS front-end. The Lift [136] and Aetherling [73] projects have shown that it might also be beneficial to use a functional IR inside the compiler. Representing different abstraction layers in the compiler as functions—from an algorithmic description to low-level hardware implementation—together with formally defined lowering rewrite rules results in an elegant compiler design that is much easier to

7. Some of the commercial HLS tools grew out of academic projects [4, 38].

8. The work on mapping parallel patterns to hardware done by the team behind the Spatial language led to a later development of several academic chips based on the idea [188, 200], which still later evolved into the chips design company SambaNova [187].

understand and extend. Later work has shown that a functional IR can also elegantly model complex memory hierarchies composed of off-chip asynchronous memories, and on-chip multi-ported, multi-banked memories typically needed to achieve high performance on FPGAs [202].

Another thread of work on HLS is the idea of separating the algorithmic description from the specific optimizations, taking inspiration from scheduling languages like Halide and TVM [191, 43]. In such an approach, the domain programmer writes sequential code that expresses the algorithmic intent, without worrying about performance. Then, in a separate step, a schedule of optimizations is built that transforms the original code into a desired state. Such a separation of concerns allows for separate languages to be used to express the optimization schedule [206], and enables the algorithm design and the optimization stage to be divided between two programmers with a clear separation of concerns [22]. As concrete examples of applying this idea to HLS we can cite HeteroCL [139], which decouples hardware customizations from the algorithm, and Allo, which also adds support for large design by allowing the optimization schedule to transform interfaces between kernels in a hierarchical design [40].

The SYCL C++ Programming Model

In this thesis, we stick with the C++ sequential execution model. We implement our contributions as compiler passes that transform the LLVM IR in the Intel SYCL C++ compiler [120] and as compiler-parametrized IP blocks, also programmed using SYCL C++. SYCL is an open parallel programming model standard developed by Khronos [133]. Compared to OpenCL, it provides a higher abstraction layer by leveraging C++ templates and by reducing the amount of host-device “glue code” that was typically necessary in OpenCL.

SYCL allows to create hierarchical hardware designs composed of multiple kernels. Kernels can communicate using SYCL pipes (a successor specification to OpenCL channels)—these are the latency-insensitive channels we have described in Section 2.2.3. One of our contributions, described further in Chapter 3, is the realization that architectures with decoupled AGU, CU, and DU can be implemented in SYCL using separate kernels for the different units, and by using SYCL pipes to implement latency-insensitive communication between them. In Section 7.3, in the last chapter we briefly describe future work on a compilation strategy and intermediate representation that has the potential to make such decoupling easier from the compiler engineering perspective.

2.4 Conclusion

In this chapter, we have provided essential background in the operation of optimizing compilers, hardware design, and the high-level programming of FPGAs, including an overview of different HLS approaches. We have tried to cover a wide range of topics here, which means that our treatment of each topic is necessarily incomplete. The technical chapters that follow next provide a more detailed background on the problems at hand, and describe the related work more narrowly and thoroughly.

The overarching goal of this chapter was to motivate the use of spatial architectures, in our case in the form of FPGAs, as a way to achieve high performance and energy efficiency. The following chapters contribute to this goal by making HLS compilers for spatial architectures more efficient on irregular codes, significantly improving the state of the art HLS approaches described in the introduction chapter.

Chapter 3

Compiler Discovered Dynamic Scheduling

In this chapter, we introduce our methodology to selectively introduce dynamic scheduling into static HLS. As described in the introduction chapter, this is desirable because irregular codes typically only have a few sources of irregularity that benefit from dynamic scheduling, and scheduling the entire circuit dynamically introduces unnecessary area and critical path overheads. Additionally, if there is no dynamic behavior at all, and dynamic scheduling is not necessary, our compilation strategy will use the Finite State Machine with Datapath (FSMD) computational model exclusively, taking full advantage of static scheduling.

We propose a compiler algorithm to discover opportunities for dynamic scheduling in an FSMD circuit. Once an FSMD region is marked for dynamic scheduling, we show how a mechanized compiler transformation can recursively *decouple* the original FSMD into multiple FSMDs, such that each FSMD can be scheduled without having to over-approximate dependency distances or operations latencies. We further show that connecting the decoupled FSMDs using latency-insensitive channels to communicate data dependencies achieves the same dynamic behavior as a Dynamic Dataflow (DDF) circuit, but with the advantage of a lower area usage and higher circuit frequency. On a set of ten benchmarks, we show that our approach achieves on average an up to 3.7× and 3× speedup against state-of-the-art FSMD and DDF HLS tools, respectively.

3.1 Introduction

A major objective of HLS tools is loop pipelining. Loop pipelining is the process of starting new iterations of a loop while previous iterations have not yet finished. The number of cycles between the start of subsequent loop iterations is called the loop 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, with an II of one being the ideal case where every functional unit in the loop performs useful work on every clock cycle.

3.1.1 Scheduling in FSMD HLS

State-of-the-art FSMD HLS uses modulo scheduling to map operations to clock cycles at compile time [196, 37, 173]. To calculate the minimum II of a loop that still guarantees that data dependencies are honored, modulo scheduling goes over all recurrences (inter-iteration dependencies) in the loop Data Dependency Graph (DDG) and calculates their $delay$ (the number of cycles needed to traverse the whole recurrence path), and their dependency $distance$ (the number of iterations between the definition of a recurrence value and its use). The final loop II cannot be lower than the recurrence constrained II, which is calculated as the maximum over all recurrences in the DDG[1]:

$$minRecurrenceII = max_i \left\lceil \frac{delay_i}{distance_i} \right\rceil . \tag{3.1}$$

There may be other factors that increase the final loop II, like area or critical path constraints, but these issues are completely orthogonal to the problem under study here.

Crucially, static scheduling has to arrive at *one* II for a loop that needs to accommodate all control-flow paths through the loop. For example, in the example from Figure 3.1a there is a recurrence for `x`. Even if the `x > 100` condition would be satisfied only half of the iterations, modulo scheduling needs to allocate cycles for the operations in the `if` body and will produce the inferior schedule in Figure 3.1c. In practice, FSMD HLS applies *if*-conversion to control-dependent operations like the one above—the branch always executes at runtime but the result might be discarded depending on control flow.

3.1.2 Scheduling in DDF HLS

DDF HLS uses dataflow scheduling to trigger the execution of operations based on the availability of data, similar to the principles of first dataflow computers [10]. This allows the II of a loop to naturally adapt to runtime conditions. For the example code in Figure 3.1a, DDF HLS would produce the ideal schedule from Figure 3.1d. However, it would do so at the expense of dynamically scheduling the whole circuit, even if only one part of it exhibits dynamic behavior. Such dataflow circuits often use several times more resources and have a significant critical path overhead compared to FSMD HLS [239].

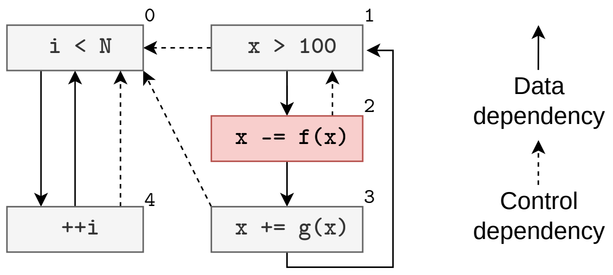
3.1.3 Combining Static and Dynamic Scheduling

There is a need to systematically and intelligently combine static and dynamic HLS scheduling, which is the problem that we study in this chapter. An initital attempt to achieve this was made by Cheng *et al.* [47, 50]. They extended a DDF HLS tool with a methodology, which lets programmers manually identify static islands in their otherwise dynamically scheduled circuit. Later, the authors provided formal, automatable guidelines on where static islands can be beneficial in a DDF circuit. Static islands can be scheduled statically on the inside and

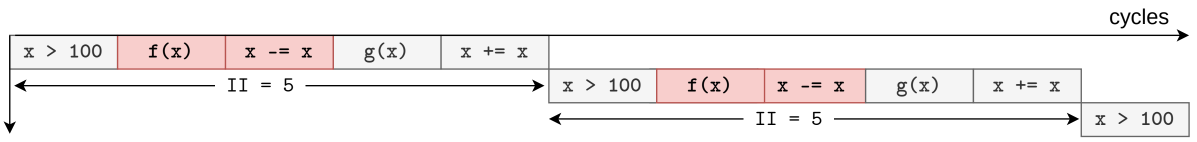
1. A DDG recurrence forms a Strongly Connected Component (SCC).

```
for (i = 0; i < N; ++i) {
    if (x > 100)
        x -= f(x);
    x += g(x);
}
```

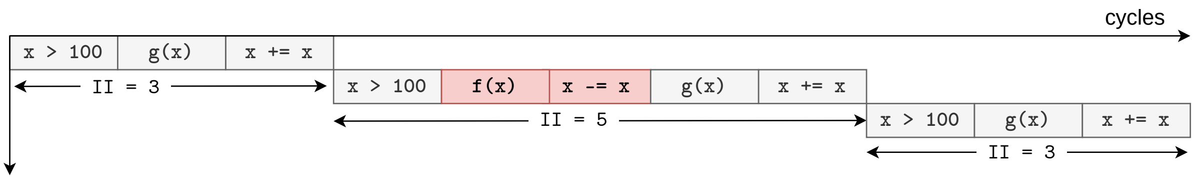
(a) Motivating code.



(b) Control and data dependency graph.



(c) A static schedule: a new iteration started every 5 cycles for all iterations.



(d) An ideal schedule: `x = x - f(x)` is not executed if not required.

Figure 3.1: A motivating example of code with an inter-iteration control-dependent data dependency (a). Current FSMD HLS needs to create a worst case schedule (c). We propose to enhance FSMD HLS with analysis and transformation passes which enable the dynamic schedule in (d).

are wrapped in interfacing logic to communicate results with the dynamic part of the circuit. The performance and resource usage of such a hybrid approach is promising, but the circuit critical path is still bottlenecked by the dynamic part. This is due to the way looping is implemented in current DDF HLS tools, which we alluded to in Chapter 1. Furthermore, some of the restrictions on what can be marked as a static island are prohibitive, e.g., a memory interface cannot be shared between static islands, nor between the dynamic and static parts of the circuit [50].

The approach we describe in this chapter has no such restrictions. We propose to introduce dynamic scheduling into FSMD HLS using only constructs available in FSMD HLS. Our method is directly informed by the scheduling algorithm used in FSMD, and can be directly integrated with existing HLS tools. We also show in our evaluation that our approach produces circuits that use strictly fewer resources and achieve better frequencies compared to the work of Cheng *et al.* [47, 50], because we introduce dynamic behavior only when it has a chance of improving throughput.

3.2 What to Schedule Dynamically

We now describe how to find basic blocks, memory operations, or whole loops that can benefit from dynamic scheduling. The question of how to actually schedule these code fragments dynamically, while keeping the rest of the circuit on a static schedule, is described in the next section.

3.2.1 Marking Basic Blocks for Dynamic Scheduling

Modulo scheduling [196, 37, 173] arrives at a minimum recurrence-constrained loop II by going over all SCCs in the DDG using Equation 3.1. Since modulo scheduling has to arrive at a single II, it has to necessarily over-approximate the recurrence-constrained II if there are control-dependent paths through the DDG with a lower $delay$, or if the dependency $distance$ is variable or unknown due to control-flow or unpredictable memory accesses. Our key idea is to selectively decouple parts of the SCC, such that each decoupled SCC instance can be scheduled without having to over-approximate unknown $delay$ or $distance$ terms.

Algorithm 3.1 describes our analysis for marking basic blocks for dynamic scheduling. We enumerate all possible control-flow paths through each DDG SCC and calculate the II of that path. For each SCC path with an $II > 1$, we collect DDG nodes that are control-dependent on anything else but the loop header (every instruction inside a loop body is trivially control-dependent on the loop header). For every collected DDG node, we obtain all other DDG nodes from the same basic block and calculate their contribution to the II of the currently considered SCC path. Specifically, we check if without the collected nodes the path $delay$ decreases or the dependence $distance$ increases, which would result in a lower loop II. If true, we mark that block for dynamic scheduling and collect all instructions in the block that are part of the currently considered SCC path. The block could contain instructions that are not part of the currently evaluated SCC in which case they will not be marked (they could still be marked when evaluating a different SCC). One could set a threshold for the $delay$ decrease or dependence $distance$ increase (e.g. to avoid dynamic scheduling overhead if the II improvement is small), however, such fine tuning will be implementation dependent.

The $GetControlDependencySrc(BB)$ call in Algorithm 3.1 returns the closest parent of BB in the Control Dependency Graph (CDG), ignoring loop headers. We use a standard technique to build the CDG out of the Control Flow Graph (CFG) dominance relationships [84]. The $CalculateII(path)$ call returns the sum of the operation latencies associated with the operations in the path.

Algorithm 3.1 Marking Basic Blocks for Dynamic Scheduling

```
1:  Input: DDG, CDG, CFG
2:  Output: List of Basic Blocks BBs
3:
4:  for $SCC \in DDG$ do
5:      for every control flow $Path \in SCC$ do
6:          if $CalculateII(Path) = 1$ then
7:              continue
8:          for $Node \in Path$ do
9:              $BB \leftarrow BasicBlock(Node)$
10:             $NodesBB \leftarrow GetDDGNodesForBlock(BB)$
11:             ▷ set difference to get path without DDG nodes originating from the $BB$ block
12:             $PrunedPath \leftarrow Path \setminus NodesBB$
13:             $C_1 \leftarrow GetControlDependencySrc(BB) \neq LoopHeader$
14:             $C_2 \leftarrow CalculateII(PrunedPath) < CalculateII(Path)$
15:             if $C_1$ and $C_2$ then
16:                 mark $BB$ for dynamic scheduling
```

Example

Consider the DDG in fig. 3.1b with two SCCs: $(0 \rightarrow 4)$ and $(1 \rightarrow 2 \rightarrow 3)$. $(0 \rightarrow 4)$ has a trivial II of 1, so it is not marked. The second SCC has an II of 5, so we check if it contains any control-dependent nodes. The blocks containing nodes 1,3 are control-dependent on the loop header block, so they are ignored. The block containing node 2, however, is control-dependent on a non-loop-header block, so it is marked for dynamic scheduling.

3.2.2 Marking Memory Operations for Dynamic Scheduling

A pair of memory operations, where at least one of the operations is a store, form memory-dependency edges in the DDG [84, 132]. Modulo scheduling treats these edges in the same way as it treats scalar dependencies. The only difference is that the dependence $distance$ between memory operations can be unknown, for example, if the access pattern is data-dependent or the compiler does not employ a strong enough alias analysis [49, 161]. If the dependency distance is known, we employ the same strategy as for marking basic blocks, namely, we check if there is a control flow path through the DDG with a higher dependency $distance$, and if yes, we check if it is control dependent on anything but a loop header. If the dependency distance between dependent memory operations is unknown, we immediately mark them for dynamic scheduling. In such cases, we say that the memory dependency cannot be disambiguated at compile time.

For any marked pair of memory operations, we also mark all other memory operations that use the same base pointer for dynamic scheduling. This is because a mix of dynamically scheduled stores and statically scheduled loads or stores might break the original program order of memory operations.

3.2.3 Marking Entire Loops for Dynamic Scheduling

So far, we have collected instructions for dynamic scheduling for cases where static scheduling has to over-approximate operation $delay$ and operation dependency $distance$ terms in Equation 3.1. In addition to dynamically scheduling such operations inside a single loop, our methodology extends to whole loops. The first scenario is a nested loop that is control-dependent on anything else than its parent loop header, and where the parent loop is not perfectly pipelined because of a dependency in the nested loop. That is, the decoupling of the inner loop should improve the average II of the outer loop.

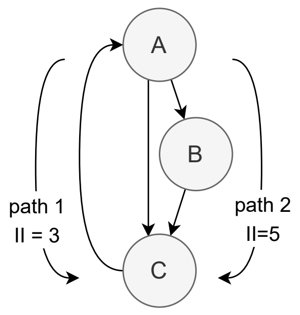
The second opportunity for decoupling whole loops is a scenario with multiple sibling loops, i.e., loops at the same level of nesting. If a loop L_1 has a sibling loop L_2, then we check if it is legal to start the second loop before the first one has finished. We mark L_2 for dynamic scheduling if:

1. There are no data dependencies between L_1 and L_2 calculated by a recurrence, and with a source in L_1 and destination in L_2. In other words, if the dependency destination in L_2 needs to wait for the whole L_1 to finish, then there is no benefit to decoupling L_2.
2. There are no memory dependencies between L_1 and L_2 such that the address expressions in L_1 and L_2 cannot be disambiguated at compile time.

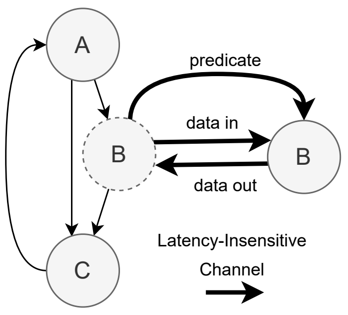
Compile time memory disambiguation across loops is often not possible. For example, in the polyhedral model [23] it would require proving that the L_1 and L_2 polyhedra do not overlap at all, which is not possible in non-affine loops. Connecting memory operations in L_1 and L_2 loops to a Load Store Queue (LSQ) to be disambiguated at runtime would be of little benefit because the L_2 loop would have to wait for all allocations in L_1 to finish—an LSQ uses the equivalent of a program counter to discover operation ordering at runtime, which results in the problem that the L_2 loop cannot know its program counter until the L_1 has finished. We will relax this requirement in Chapter 6 by introducing a new program order schedule optimized for hardware that, as opposed to the program counter, allows for parallel execution of loops.

3.3 Achieving Selective Dynamic Scheduling

This section presents our main contribution: a method for introducing dynamically scheduled code regions in modulo-scheduled HLS via latency-insensitive channels.



(a) Statically scheduled loop with a worst case II.



(b) Decoupled unpredictable control-dependent data dependency.

Figure 3.2: Our main idea in this chapter: control-flow paths with a higher recurrence-constrained II are decoupled into separate FSMDs, with data dependencies communicated via latency-insensitive channels. A recurrence through registers is decoupled into a predicated PE. Figure 3.3 shows how recurrences through memory are handled.

3.3.1 Dynamically Scheduled Basic Blocks

Basic blocks marked for dynamic scheduling are transformed into predicated Processing Elements (PEs). Figure 3.2a shows a possible CFG for our motivating example code from Figure 3.1a. Figure 3.2b shows a high-level overview of how the marked block *B* would be decoupled by our transformation. All instructions collected in the marked block are moved from the original CFG to the predicated PE. We then collect the set of input and output data dependencies between the PE and the original CFG using a simple data flow algorithm: every SSA value used in the PE but defined in the original CFG is an input dependency from the original CFG to the PE, and vice versa for output dependency from the PE to the original CFG. All SSA values collected as input dependencies are replaced with latency-insensitive channel writes in the original CFG, and with reads in the PE. The dual is done for output dependencies. Finally, we insert a predicate channel write to the beginning of the decoupled block in the original CFG which will trigger our predicated PE whenever control transfers to that block during execution.

The PE is guaranteed not to access any memory directly. If a memory access inside a marked block was itself marked for dynamic scheduling, then it will be replaced by channel read or write, which we describe in Section 3.3.2. If the access was not marked, we keep it in the original CFG and communicate its operands as dependencies between the PE and the original CFG—a load used in the PE becomes an input dependency, a store operand defined in the PE becomes an output dependency. This is necessary, because accessing a memory from two concurrently executing PEs without any synchronization might introduce race conditions.

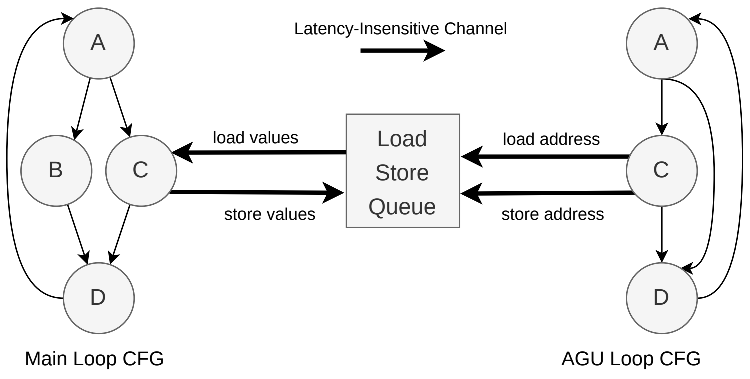


Figure 3.3: A recurrence through memory is decoupled into an LSQ and an AGU.

Effect of the Transformation

After removing all inter-iteration dependencies that have an unpredictable or variable dependency distance or operation delay, modulo scheduling will find that the Figure 3.1a reccurence $delay$ is now 3 and not 5. Whenever control transfers to the decoupled basic block, the original loop will trigger the predicated PE and communicate the required input dependencies. It will then continue its execution until it encounters an operation that is dependent on an output dependency from the decoupled block. If such a dependency is encountered, then the original loop is stalled until the required dependency is communicated from the PE. Thus, a variable II is achieved which naturally adapts to runtime conditions.

Communication Avoiding Optimizations

As presented so far, our transformation is local to a basic block and does not require updating SSA values in other blocks. This can change after *hoisting redundant channel communication* out of loops. A channel operations in the main CFG can be hoisted out before or after the loop if the value it is carrying is only used or defined in the predicated PE. For example, the code in fig 3.1a would not have any channel operations hoisted out, because the `x` value would be used in both the PE and the original CFG. If, however, the `x += g(x)` statement would be removed, then the channel operations supplying and receiving `x` could be hoisted out because the original CFG would not use its value in the communication sequence: $CFG \xrightarrow{x} PE \xrightarrow{x} CFG \xrightarrow{x} PE$. This can be checked using a standard dataflow analysis.

Dynamic scheduling of whole loops is achieved in the same fashion as for basic blocks, with the difference that the dependencies are calculated for the whole loop.

3.3.2 Dynamically Scheduled Memory Operations

Memory operations marked for dynamic scheduling require runtime memory disambiguation machinery, which we provide in the form of an LSQ. An LSQ can check for memory conflicts at runtime by comparing load and store addresses out-of-order with the actual memory accesses, and stall the datapath if a true data hazard is detected. One can easily plug any LSQ design into FSMD HLS, however, this is not enough to achieve ideal throughput.

For an LSQ to be most effective, it should be able to accept load and store requests ahead of store values. In DDF HLS, this happens naturally since the production of memory addresses is decoupled from the actual load and store operations. To achieve the same effect in FSMD HLS, the address generation should also be decoupled, similar to the principle of Decoupled Access/Execute (DAE) architectures [205]. Decoupled memory accesses have been studied before in the context of FPGAs, but only for prefetching and hiding variable latency memory accesses [42, 51]. We contribute the insight that this approach, together with an LSQ, can enable dynamically scheduled out-of-order loads in static HLS [217].

In Chapter 4, we describe our approach to automatically generate a DAE architecture in FSMD HLS, including an analysis to check if address decoupling is profitable in the first place, and an approach to let the compiler customize key parameters of the LSQ to achieve ideal throughput at minimal area cost and critical path. In Chapter 5, we introduce compiler support for speculation in DAE architectures to enable address decoupling on a much larger class of codes. In this section, we only describe the effect of using the DAE approach and our LSQ, without going into implementation details. In the evaluation of this chapter, we use an LSQ similar to previous work [125], without applying our optimizations from Chapter 4.

If the Chapter 4 analysis determines that decoupling of address generation is profitable, then we proceed to decouple the memory-generating instructions into a separate PE, which we call the Address Generation Unit (AGU). We copy the original loop CFG and delete from it all instructions not needed to generate the addresses (these can be easily obtained by walking the DDG). Channels for input and output dependencies between the new AGU and the original CFG are materialized similarly to Section 3.3.1. Regardless of whether the generation of memory addresses is decoupled or not, we insert the required channel calls to supply load and store requests to the LSQ and to supply to it and receive from it store and load values. Figure 3.3 shows the resulting communication pattern if the address generation is decoupled. Load and store instructions in block *C* have been replaced with latency-insensitive channel reads and writes from and to an LSQ, respectively. The addresses to the LSQ are supplied by a separate FSMD component, which contains only address-generating instructions. The generation of load and store addresses in this decoupled component is control-flow equivalent to the consumption and generation of load and store values in the original CFG.

3.3.3 Composability of Transformations

The presented transformations for introducing dynamically scheduled basic blocks, loops, and memory operations are composable. A decoupled loop can have a number of its own basic blocks decoupled, and the basic blocks can include dynamically scheduled memory operations. The problem of LSQ request ordering across decoupled code regions is solved by the design of our LSQ. Our LSQ 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 function of the tag inside the LSQ is described in Chapter 4, but suffice here to say that it in effect produces a data dependency chain between memory stores and other memory operations. This tag dependency chain is picked up through our input and output dependency collection, and as a result is communicated between decoupled code regions according to runtime control flow, naturally taking care of the correct order of LSQ requests.

3.4 Evaluation

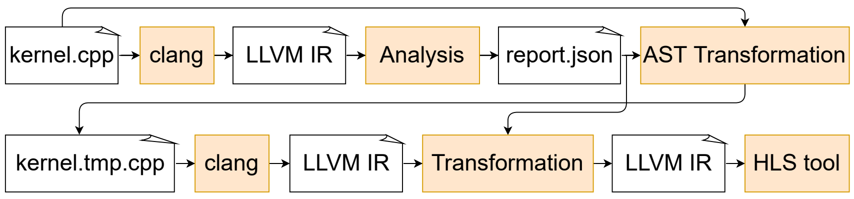


Figure 3.4: Our tool flow. The AST transformation consists of creating kernel copies, inserting LSQ kernels and creating latency-insensitive channels, which are later used by the transformation operating on the LLVM IR.

We implemented our compiler analysis and transformations in the LLVM framework [142] and integrated them with the Intel HLS compiler (version 2023.1.0). Figure 3.4 shows an overview of our tool flow. Our implementation and this evaluation are publicly available[2].

In this section, we evaluate our approach of selectively introducing dynamic scheduling in FSMD HLS against three other HLS scheduling approaches.

- **SS**: Pure statically scheduled FSMD HLS using the Intel HLS compiler [120].
- **DDF**: Pure DDF HLS using the academic Dynamatic compiler [126].
- **DASS**: A methodology to introduce selective static scheduling into DDF HLS [47] (we have described the limitations of this approach in Section 3.1.3).

We try to answer the question if our selective dynamic scheduling approach achieves a better area-delay product compared to the above approaches.

3.4.1 Implementation

We use Intel HLS [120] as a representative of FSMD HLS in this work. Intel HLS uses the SYCL programming model [118] defined by the Khronos Group and intended as a successor to OpenCL that offers a higher abstraction layer. In our implementation a SYCL kernel can be through of as a single FSMD instance; a SYCL pipe is an implementation of a latency-insensitive channel. Each SYCL kernel has its own static schedule, and kernels can communicate with each other via SYCL pipes. In the context of our implementation work, SYCL is advantageous over OpenCL because SYCL pipes are implemented as types, not kernel arguments, making them easier to use in compiler transformations. To the best of our knowledge, there is no prior work that shows that dynamic scheduling can be implemented using off-the-shelf FSMD HLS tools as we do here.

2. `https://github.com/robertszafa/elastic-sycl-hls`

3.4.2 Evaluation Methodology

Dynamatic is based on Xilinx tools, while our approach is implemented on top of Intel HLS, which makes a direct comparison in terms of absolute area usage difficult. Because of that, we compare the *normalized* area usage of Dynamatic, DASS, and our approach against their respective static HLS baseline. The register and LUT usage overhead are combined using geometric mean into a single area overhead. For Dynamatic and DASS we used the post-synthesis report from Vivado 2020.2 for the Xilinx xc7k160tfbg484; our approach used Quartus 19.2.0 post-synthesis reports for the Altera 10AX115S. Clock cycles were obtained using ModelSim.

We applied our approach to ten benchmarks from the HLS literature [47, 126], which were made publicly available by Cheng [45]. We describe these benchmarks in Table 3.1. Since some of the codes have data-dependent behavior, we report worst- and best-case performance for different input data distributions where applicable. In codes with unpredictable memory addresses, we use an LSQ adapted to our approach, while Dynamatic and DASS use the Dynamatic LSQ [125]. Any difference in the experiments due to the different LSQ designs is left out of the evaluation— we do not include the LSQ areas in the results, and the throughput of the two LSQs is the same. This means that the area overhead of our approach, DDF HLS, and DASS HLS is slightly larger than what we report in this Chapter. Chapter 4 focuses exclusively on dynamic memory scheduling in HLS and provides a more detailed area cost analysis of our LSQ. All codes evaluated in this chapter use on-chip SRAM, although our implementation also supports DRAM, which will also be evaluated in the Chapter 4.

We also include two codes that have no dynamic behavior in our evaluation. This is to highlight that our approach has no overhead when scheduling code with no dynamic behavior. This is because, fundamentally, our approach just extends existing FSMD HLS tools to support dynamic scheduling—if there is no dynamic behavior in the code, the single FSMD implementing it will not be decoupled. In contrast, the DDF HLS and DASS baselines, against which we compare, always have some dynamic behavior in their circuit, even on codes that can be fully scheduled statically.

3.4.3 Results

Figure 3.5 shows the area overhead and speedup of the DDF approach, the DASS methodology, and our approach over their respective static scheduling baselines. Table 3.2 at the end of this chapter features detailed results of all ten benchmarks, which we analyze in the next paragraphs.

Table 3.1: Benchmarks used in our evaluation. Benchmarks 1–4 have control-dependent inter-iteration data dependencies, 5–8 have unpredictable memory hazards, and 9–10 have no data-dependent behavior and can be optimally scheduled without our transformations.

Kernel	Description	Class
sparseMat	Calculates sparse matrix power.	Control-dependent inner loop.
tanh	*tanh* approximation using CORDIC [72].	Control-dependent recurrence.
filterSum	Array sum with a filter.	Control-dependent recurrence.
vecNorm	Vector normalization.	Control-dependent recurrence.
histogram	Counts number of occurrences per bin.	Unpredictable memory access.
sort	Simple bubble sort.	Unpredictable memory access.
getTanh	Applies *tanh* on an entire array.	Unpredictable memory access.
BNN	Small binarized neural network.	Unpredictable memory access.
covariance	Computes the covariance matrix.	No dynamic behavior.
gesummv	Scalar, vector and matrix multiplication.	No dynamic behavior.

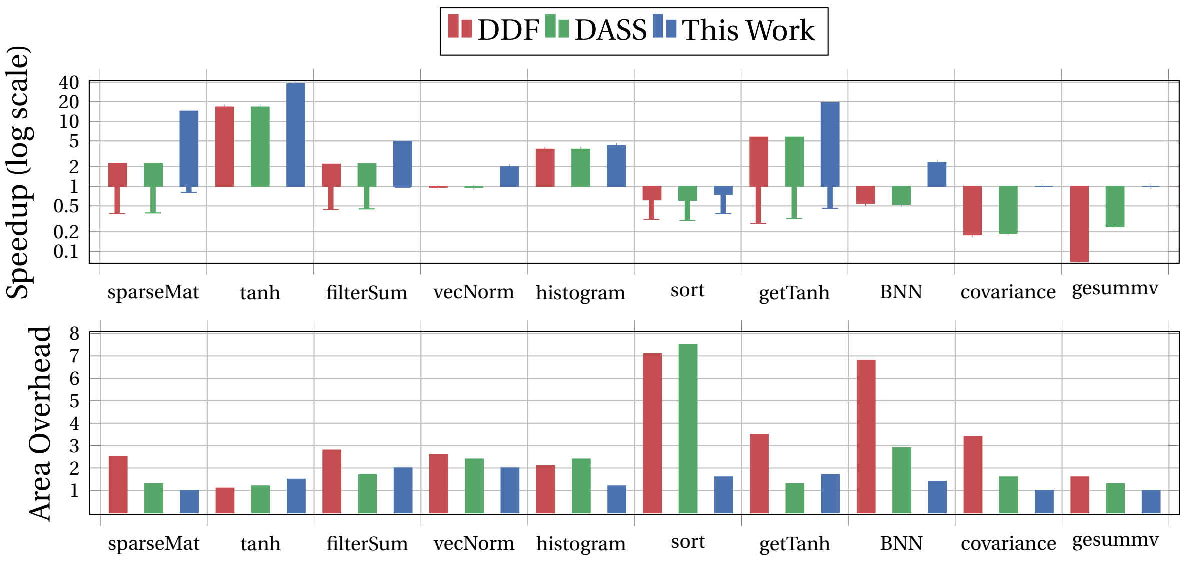


Figure 3.5: Speedup and area overhead of DDF HLS [126], DASS [47] and this work against their respective statically scheduled (SS) baselines. The range bars in the speedup plot represent the range of speedup as the data distribution changes. A speedup below 1 indicates a slowdown relative to static scheduling.

Area Usage

DDF HLS incurs area overhead for handshaking logic, and the missed opportunity for resource sharing: if two hardware components become decoupled via latency-insensitive channels, then the HLS tool cannot make as many latency assumptions as it could if the two components were following the same static schedule. On average, our approach increases area usage only by a factor of 1.3× over pure FSMD HLS, but for the DDF approach this overhead rises to 2.7×. The DASS approach improves the area overhead a bit compared to DDF, but at 1.8× overhead it still consumes more area than our approach.

In cases where modulo scheduling does not have to over-approximate dependency distances or operation latencies, our approach does not change the static hardware, which results in no resource usage increase, while DDF and DASS see a resource increase of 2.3× and 1.4×, respectively, for those benchmarks. The area overhead of DDF HLS and DASS in the *sort* and *BNN* benchmark is the highest. *BNN* consists of bit-level logic which we could get more aggressively optimized in the more mature FSMD HLS tools compared to the academic Dynamatic tool. *sort* on the other hand has a large number of basic blocks compared to instructions in them, resulting in a large ratio of dataflow components to functional units in the dataflow circuits, which use additional units to implement control-flow. We also note that DDF HLS and DASS use on average 1.6× and 1.1× more DDF HLSPs than static scheduling, while our approach does not increase DDF HLSP usage at all.

Critical Path

The biggest advantage of our approach is the higher frequency achievable by FSMD HLS compared to DDF HLS. On codes 1-4, which do not require an LSQ, our approach achieves on average 0.94× the frequency of FSMD HLS. In contrast, DDF HLS and DASS both see a 4× frequency drop because they introduce a new critical path to implement their loop constructs. On codes 5-8, the critical path is increased significantly by the LSQ for all three approaches. The next chapter shows how this LSQ overhead can be overcome with a closer compiler-hardware co-design. On codes without dynamic behavior, DDF and DASS see frequency drops, while our approach does not.

Throughput

We achieve the same or better throughput as SS, DDF, and DASS (lower cycle counts in Table 3.2 indicate higher throughput). We perform better than DDF HLS and DASS on *getTanh* and *BNN* because they involve nested, control-dependent loop nests, which current DDF HLS tools struggle with, because they cannot start executing the next loop in program order until the previous loop has finished. In contrast, FSMD HLS can overlap the execution of multiple loops in a loop nest—in this case, the only stalls in the loops will be from waiting for data dependencies from latency-insensitive channels that our approach has introduced. DASS performs better than Dynamatic on *getTanh* when the data distribution favors static scheduling, but it cannot achieve perfect pipelining when there are no data hazards, because it cannot start the next iteration of the outer loop until the inner loop has returned from its static island. On codes without any dynamic behavior DDF HLS and DASS incur non-trivial overheads, while our approach does not change the FSMD hardware.

Execution Time

Execution time is the product of the number of cycles and circuit frequency, and since we benefit from the high frequency of using the FSMD HLS approach, while also achieving the same (or higher) throughput as DDF, our approach performs better than DDF HLS and DASS. Across the ten benchmarks our approach is on average up to **3.7**× and **3**× faster than DDF HLS and DASS, respectively. The performance of our approach is also more stable and predictable across varying data distributions. This is visible in Figure 3.5, where the speedup range bars for DDF HLS and DASS dip below 1 more often (which means a *slowdown* over SS)—DDF HLS and DASS would actually be slower on code with data-dependent behavior, if the underlying data distribution agrees with the decision made by over-approximating terms during FSMD scheduling. Our approach is only slower than SS in the *bubbleSort* and *getTanh* benchmarks, and only when the data distribution completely favors static scheduling. This is because the frequency of our circuits for those codes is more than 3× lower than SS due to the critical path overhead of the LSQ (a shortcoming which is addressed in the next chapter).

3.4.4 Limitations

One limitation of our work is that on codes that require an LSQ, the speedup of our implementation is less than ideal, because we suffer the same frequency degradation as DDF HLS and DASS. Thus, a memory disambiguation method with no critical path overhead but with the same throughput as an LSQ is desired for our approach. We present work towards this goal in the next chapter.

Another limitation of our implementation is the fact that our II analysis could potentially diverge from II analysis of the Intel HLS compiler with which we integrate. The II calculation in Intel HLS is performed in the back-end of the compiler, which is closed source. We do not have access to the model of operation latencies used in Intel HLS, and we are also oblivious to optimizations like operator chaining [116]. However, this is not a fundamental limitation; it is only a symptom of performing experiments with a tool that is not fully open-source. Ideally, the analysis for finding opportunities for dynamic scheduling should use the same loop II scheduling implementation and latency model as the compiler back-end.

Similarly, in a production compiler, the implementation of our decoupling transformation should not rely on SYCL pipes and kernels, which are user-facing features. To this end, future work could investigate open-source HLS tools. A promising development is CIRCT [91]. CIRCT is based on the MLIR compiler infrastructure [143] and uses different dialects (intermediate representations) to represent hardware with different semantics. For example, there exist separate dialects for statically and dynamically scheduled circuits. We will explore the possibility of fully open source HLS compilers in Section 7.3 in the conclusion chapter.

3.5 Related Work

Carloni *et al.* formalized the theory of latency-insensitive design [39]: assuming that modules can be stalled, their communication protocol separates module communication from their cycle behavior. Later, Cortadella *et al.* proposed a simplified latency-insensitive protocol called SELF, which could be applied in synchronous circuits [57]. The SELF protocol, or modifications thereof, has since been used as a basis for dynamically scheduled circuits [129, 113, 219, 126, 224, 88]. Most commercial HLS tools provide a latency-insensitive channel construct, e.g., SYCL Pipes [133], or AMD Xilinx Streams [4].

Several HLS tools that automatically create dataflow circuits have been developed in academia [126, 219, 74, 229]. CASH [229] was one of the first C to hardware compilers that used dynamic scheduling. It differs from recent dynamic HLS in that it used asynchronous hardware. CASH used an explicit dataflow Intermediate Representation (IR) called Pegasus IR [34], a form of predicated Single Static Assignment (SSA) [174], which implicitly transforms control-flow into dataflow. Dynamatic by Josipovi *et al.* [126] is the most recent academic DDF HLS tool, introducing latency-insensitivity for every def-use SSA value pair that spans across two basic blocks. The resulting circuits perform well on irregular code, but, as we have argued in this chapter, they use more dynamic scheduling than required. Instead, we selectively introduce the minimum amount of dynamic scheduling to achieve the same throughput by using the DDG and CDG to connect selected dynamically scheduled producer and consumer pairs directly. In contrast, Dynamatic materializes dataflow constructs using the CFG [198] order, which means that a token needs to flow through all basic blocks between a producer and consumer (although recent work started to address this issue [76]).

Xu *et al.* [239] proposed to use Linear Temporal Logic (LTL) to prove that certain handshaking signals in a dataflow circuit will never be used or that they are equivalent to other signals at the time of use, allowing them to be removed without losing correctness. While this brings the resource usage of dataflow circuits closer to static HLS, there is still a significant gap between the two, both in terms of resource usage and achievable critical path. Furthermore, model checking of LTL formulas is notorious for its exponential complexity in the number of transitions in the system, e.g., [239] reports 80 min check time for a code with two matrix multiply loops. The authors use the abstraction technique to reduce the size of the state system, but it remains to be seen how this approach performs on more complex codes that result in a DDG with high connectivity (and thus more states that cannot be abstracted away). We propose to tackle the problem of resource usage by selectively introducing dynamic scheduling into static HLS. We argue that the number of dynamic regions in codes is much smaller than the number of static regions, making it more beneficial to selectively introduce dynamic behavior into static HLS than vice versa, as is evidenced by our evaluation and previous work [47]. We also argue that it is easier to find dynamic regions starting from FSMD HLS than to recover static information in a fully latency-insensitive system, since the algorithms that produce FSMD schedules (e.g., modulo scheduling) already provide enough information for this goal.

There is also a line of work that aims to improve the quality of results from static scheduling by using source-to-source transformations [67, 153]. For example, Liu *et al.* [153] applied polyhedral analysis to split the iteration space based on the minimum II that they can safely use, generating a separate loop for each split. Derrien *et al.* [67] proposed to improve loop pipelining on unpredictable codes by speculative execution realized via a source-to-source transformation. Source-level approaches are usually tightly coupled to the specific HLS tool that is being targeted. Our analysis and transformations are expressed at the Control and Data Dependency Graph level and informed by the fundamental limitations of scheduling FSMD circuits, making them independent of any particular HLS tool.

Our approach to dynamic scheduling resembles Decoupled Software Pipelining (DSWP) proposed by Ottoni *et al.* [175]. In DSWP, the DDG and CDG of a loop are partitioned into SCCs, which are decoupled into separate CPU threads communicating via FIFOs. Although aimed at multicore CPUs, the decoupling approach works especially well on FPGAs where FIFO communication is efficient, e.g., S. Cheng *et al.* used the DSWP principle to minimize stalls resulting from cache misses on reconfigurable accelerators [51]. Although similar in nature, our approach is fundamentally different because our goal is the selective introduction of dynamic scheduling and we perform the decoupling *inside an SCC*, while DSWP decouples *entire SCCs*. To illustrate the difference, consider the DDG and CDG in Figure 3.1b. DSWP would decouple the whole recurrence SCC $1 \rightarrow 2 \rightarrow 3$, while we would decouple only node 2.

3.6 Conclusion

In this chapter, we have presented an algorithm for identifying code regions that can benefit from dynamic scheduling in FSMD HLS and contributed a novel method for realizing dynamically scheduled basic blocks, loops, and out-of-order memory operations in FSMD HLS. Our main idea is to decouple parts of control-flow paths that increase the loop II into separate FSMD components connected via latency-insensitive channels. On ten irregular code benchmarks, our approach achieves an average 3.7× speedup over state-of the-art DDF HLS, while also using less area.

One key limitation of the work in this chapter is the fact that using an LSQ in our circuits to schedule memory operations dynamically significantly increases the circuit critical path, thus losing much of the performance edge over statically scheduled FSMD HLS. In the next chapter, we address this problem with a closer compiler-hardware co-design.

Table 3.2: Evaluation of our approach against pure FSMD HLS (SS), DDF HLS [126], and the DASS methodology [47]. Three sets of benchmarks: benchmarks 1–4 have control-dependent inter-iteration data dependencies, 5–8 have unpredictable memory hazards, and 9–10 have no data-dependent behavior and can be optimally scheduled without our transformations.

	Area ×SS			**DSPs ×SS**			**FMax (MHz)**				**Cycles (thousands)**				**Execution Time (μs)**			
	[126]	[47]	Us	[126]	[47]	Us	SS	[126]	[47]	Us	SS	[126]	[47]	Us	SS	[126]	[47]	Us
sparseMat	2.5	1.3	1	2	1	1	415	161	161	334	1.8–11	0.3–11	0.3–11	0.1–11	4.3–26.5	1.9–68.3	1.9–68.3	0.3–32.9
tanhDouble	1.1	1.2	1.5	1	1	1	371	161	161	372	38	1	1	1	102.4	6.2	6.2	2.7
filterSum	2.8	1.7	2	1	1	1	425	185	189	411	5	1–5	1–5	1–5	11.8	5.4–27	5.3–26.5	2.4–12.2
vecNorm	2.6	2.4	2	4	1.4	1	374	185	201	379	12	6.1	6.7	6.1	32.1	33	33.3	16.1
hmean	**1.9**	**1.5**	**1.4**	**1.7**	**1.1**	**1**	**1**	**0.43**	**0.44**	**0.94**	**1**	**0.14–0.34**	**0.14–0.34**	**0.09–0.34**	**1**	**0.33–0.78**	**0.33–0.78**	**0.12–0.36**
histogram	2.1	2.4	1.2	1	1	1	356	146	146	168	9	1	1	1	25.3	6.8	6.8	6
sort	7.1	7.5	1.6	1	1	1	447	139	136	168	20	10–20	10–20	10–20	44.7	71.9-143.9	73.5-147.1	59.5-119
getTanh	3.5	1.3	1.7	2	1	1	368	119	119	161	44-56	2.5–66	2.5–56	1–56	120–152	21–554.6	21–470.6	6.2–331.3
BNN	6.8	2.9	1.4	3	1	1	447	124	119	174	60	30	30	10	134.2	241.9	252.1	57.5
hmean	**4.4**	**2.8**	**1.4**	**1.6**	**1**	**1**	**1**	**0.33**	**0.32**	**0.42**	**1**	**0.2–0.5**	**0.2–0.32**	**0.12–0.18**	**1**	**0.6–1.54**	**0.62–1.5**	**0.29–0.88**
covariance	3.4	1.6	1	1.8	1.8	1	434	86	100	434	68	72.9	84	68	156.7	847.7	840	156.7
gesummv	1.6	1.3	1	2.2	1.7	1	410	113	163	410	65.8	262	68.8	65.8	160.5	2318.6	674.5	160.5
hmean	**2.3**	**1.4**	**1**	**1.4**	**1.3**	**1**	**1**	**0.23**	**0.3**	**1**	**1**	**2.07**	**1.13**	**1**	**1**	**8.84**	**4.75**	**1**
all hmean	**2.7**	**1.8**	**1.3**	**1.6**	**1.1**	**1**	**1**	**0.3**	**0.35**	**0.74**	**1**	**0.39-0.71**	**0.32-0.5**	**0.22-0.39**	**1**	**1.21-2.2**	**0.99-1.77**	**0.33-0.68**

Chapter 4

Dynamic Out-of-Order Memory Scheduling

In this chapter, we describe how dynamic out-of-order memory operations can be efficiently implemented in Finite State Machine with Datapath (FSMD) HLS. As we saw in the previous chapter, most sources of irregularity are memory operations with unpredictable memory addresses. To make the execution of such irregular codes efficient, HLS tools must support dynamically scheduled memory operations. The most common approach in this situation is to use a Load-Store Queue (LSQ), which can disambiguate data hazards at circuit runtime. We have already used an LSQ in the previous chapter, where we focused on the idea of combining dynamic and static scheduling in general. In this chapter, we describe our LSQ design in more detail and analyze the performance impact of our design choices.

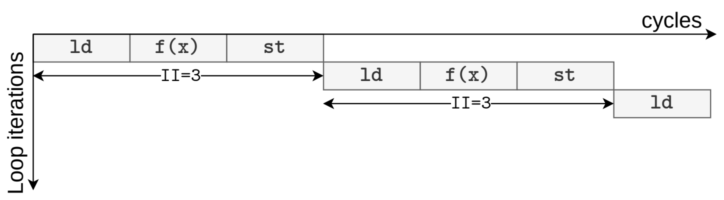
In previous work, the increased throughput from using an LSQ typically came at the price of lower clock frequency and higher resource usage compared to circuits without an LSQ. The lower frequency often nullifies any throughput improvements over static scheduling, while the resource usage becomes prohibitively expensive with large queue sizes. We show how our compiler can automatically parameterize crucial LSQ parameters to significantly reduce the overhead of using an LSQ in HLS.

We show that in order to take advantage of an LSQ in FSMD HLS, the LSQ address generation needs to run ahead w.r.t. the rest of the circuit. We propose to achieve such run-ahead behavior by using a Decoupled Access/Execute (DAE) architecture, and we show how the compiler can automatically generate DAE architectures in HLS. We also introduce the concept of speculative LSQ memory requests to preserve address decoupling in the face of Loss of |Decoupling (LoD) events due to control dependencies, a capability not present in previous LSQs for HLS. On a set of benchmarks with data hazards, our approach achieves an average speedup of 11× against static HLS and 5× against dynamic HLS that uses a state of the art LSQ from previous work. Our LSQ also uses several times fewer resources, scaling to queues with hundreds of entries.

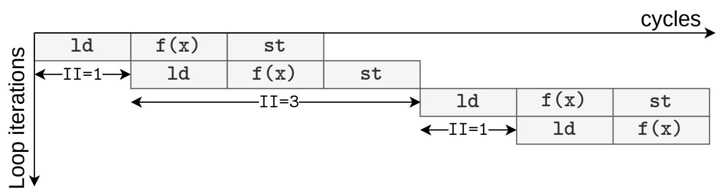
```
// idx = 0, 1, 1, 2, 2, ...
for (int i = 0; i < N; ++i) {
  int x = data[idx[i]];
  data[idx[i]] = f(x);
}
```

RAW data hazard

(a) Motivating source code with a RAW 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.

Figure 4.1: 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).

4.1 Introduction

As described in the previous chapter, one of the first steps in scheduling an FSMD circuit is determining the minimum number of cycles between the start of subsequent loop iterations, such that data dependencies across iterations are honored. For example, modulo scheduling uses Equation 3.1 to find the maximum recurrence-constrained Initiation Interval (II) across all recurrences for a given loop. Equation 3.1 uses the dependency $distance$ term, which is the number of iterations between the definition of a recurrence value and its use. FSMD HLS compilers rely on an accurate memory dependency analysis to discover the dependency $distance$ of a Data Dependency Graph (DDG) recurrence through memory. Work on memory dependency analysis, such as the polyhedral model, are directly applicable in this case [161, 153, 49, 185, 246]. However, there is a large class of codes where the calculation of the dependency distance is fundamentally impossible due to limited compile time information. Take the code in Figure 4.1 as an example. The code contains data-dependent memory reads and writes that form a recurrence in the DDG. For such codes, the dependency distance cannot be obtained and has to be conservatively set to one, i.e., it is assumed that every iteration needs to wait for all previous iterations to finish, eliminating any possibility for loop pipelining, resulting in the pipeline from Figure 4.1b.

The alternative to the FSMD scheduling approach is to schedule memory accesses dynamically at runtime, when the unpredictable memory accesses become known. We have briefly described our approach to introducing dynamic memory scheduling into FSMD HLS in Section 3.3.2. Our approach, like most other approaches [125, 236], uses an LSQ. The context of HLS and the nature of distributed on-chip memories on FPGAs allows to use separate LSQs for separate base addresses. Such distributed smaller LSQs are typically more efficient than one large monolithic LSQ.

The LSQ allows out-of-order loads by checking store address histories—loads can execute before all previous stores in program order have committed, as long as no data hazard is violated. Such out-of-order dynamic load execution is essential for pipelining codes with unpredictable memory accesses. For our example code from Figure 4.1a, HLS with an LSQ can achieve the ideal schedule in Figure 4.1c. However, all previous LSQs intended for HLS, including the one we used in the previous chapter, incur non-trivial resource and critical path overheads, often nullifying any throughout advantage [126, 48].

In this chapter, to unleash the full potential of HLS circuits using dynamic memory scheduling, we propose an LSQ with a faster critical path, lower area overhead, and better scalability than previous work (Section 4.4). This is achieved by a closer compiler-hardware co-design, allowing us to parametrize our LSQ exactly to the specific code for which we HLS is performed (Section 4.6). We also give more detail about how dynamically scheduled memory operations can be enabled in FSMD HLS in the first place, irrespective of what LSQ is used; and we provide a brief motivation and intuition for speculative LSQ memory requests (Section 4.5).

4.2 Background & Related Work

There are two main approaches to enable 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 [236].

4.2.1 Value-Based Memory Disambiguation

Thielmann *et al.* investigated the use of load speculation in reconfigurable hardware [220]. 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.* [62] 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. Misprediction triggers a squash and replay action, which adds overhead.

The benefit of value-based disambiguation is that it can pipeline loops where the store operation is control-dependent on a load [220]. The disadvantage is that squash-and-replay is prohibitively expensive. Budiu *el al.*, who developed one of the first dynamically scheduled HLS compiler [33, 229], noted that “implementing a generic prediction scheme (be it branch prediction or value prediction) in a dataflow model is hindered by the difficulty of building a mechanism for squashing the computation on the wrong paths" [32]. We address this fundamental issue in the next chapter by proposing a compiler transformation that can disambiguate memory accesses on speculated paths with no requirement for squash-and-replays, i.e., with no misspeculation cost. In this chapter, we show how our LSQ can support such speculative memory requests.

4.2.2 Address-Based Memory Disambiguation

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 an LSQ. Most LSQs aimed at HLS have a similar operating principle as LSQs used in out-of-order CPUs [236]. For example, the Dynamatic LSQ [125] has a single store queue buffer which holds stores in-flight to memory, together with metadata needed to recover program order. Dependent loads check this structure for aliasing using the memory address and other metadata, deciding if a load is safe to perform, if a store value can be forwarded, or if the load has to wait. It is this single-cycle Content Addressable Memory (CAM) access that maps poorly to FPGA technology, resulting in a high critical path and area usage [151].

Our LSQ design described in this chapter is fundamentally different. We recognize that LSQs for HLS do not have to be as general as CPU LSQs since we can use compiler analysis to specialize an LSQ exactly for a given program. We propose to break up the single store queue CAM into two separate shift-register-based queues, one holding just store address requests and the other store commits. Compiler analysis allows us to size the shift registers exactly. Instead of the single-cycle CAM access in Dynamatic, we spread our memory disambiguation checks into multiple pipeline stages for an improved critical path and resource usage. This is enabled by using a DAE architecture, which allows for the generation of LSQ memory requests to run ahead w.r.t. load value consumption, easing the latency requirement on the LSQ. Another major difference is our support for speculative address requests, enabled by having separate store request and commit queues, which paves the way for efficient speculative memory requests described in the next chapter. Our LSQ approach can be seen as a generalization of shift-registers-based approaches to pipelining of loops with statically analyzable dependency distances [2], e.g., sparse matrix-vector multiply accelerators [121].

4.2.3 Program-Order Representation

The central question in LSQ designs for spatial computing is how to recover program order of memory requests without a Program Counter (PC) that can be used in CPU LSQs. Josipovi *et al.* proposed to allocate LSQ addresses from a single basic block in parallel and sequentialize the execution of basic blocks [125]. 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 LSQ does not rely on the sequential execution of basic blocks. Instead, we recover program order by tagging each memory request with a unique integer representing the state of memory at that time. Our tags are similar to the work by Elakhras *et al.* [77] who addressed the sequentialized block request problem of the Dynamatic LSQ by introducing virtual data dependencies between blocks with LSQ accesses. However, in addition to ordering the request of addresses, we also use the actual tag values for disambiguation inside the LSQ.

4.3 The Memory Disambiguation Problem

We define an LSQ memory request as an $(address, tag)$ tuple. The tag is an integer indicating the state of memory expected by the request—it is used to recover the program order sequence of memory operations, similarly to how the PC is used in CPU LSQs. 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 i-th store, with the state at $i = 0$ representing the initial memory state.

The inputs to our LSQ are the following streams: a sequence of load requests; a sequence of store requests; a sequence of store values where each $stValue_i$ corresponds to $storeRequest_i$. We require that requests and values within streams retain program order. The LSQ outputs a sequence of load values, which correspond to the sequence of previously made load requests. The tag of a load request indicates which memory state is expected by the load; the tag of a store request represents the new memory state after the store.

Problem Definition

Given any pair of $loadRequest_i$ and $storeRequest_k$, if the two conditions hold:

$$\begin{gathered} loadRequest_i.address = storeRequest_k.address, \\ loadRequest_i.tag \geq storeRequest_k.tag, \end{gathered} \tag{4.1}$$

then $loadRequest_i$ cannot be served before observing the side-effect of $storeRequest_k$.

Finally, we define a store commit as an $(address, value)$ tuple. Our LSQ holds a sequence of store commits internally, representing values in-flight to memory. Store commits can be used to forward stored values directly to aliasing loads. Note the omission of program ordering information from the store commits. In previous LSQs, in the case when a load aliases multiple store commits, the forwarding logic had to pick the youngest store commit. In our case, this would require adding a tag field to the store commit tuple, and finding a store commit with the maximum tag value. We avoid the need for this logic by keeping store commits ordered, and by ensuring that the store commits do not contain stores that, in program order, come after a load that has not yet been served.

It is the responsibility of the compiler to instrument the Address Generation Unit (AGU) with code that generates the above information.

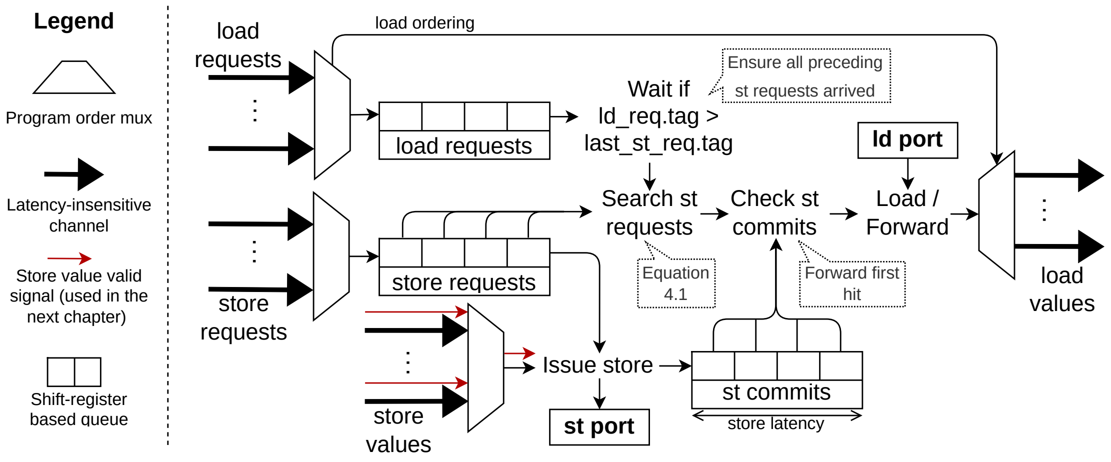


Figure 4.2: Our shift-register-based LSQ design. Load requests are checked for aliasing over multiple pipeline stages. The store commits and store requests queues are sized automatically by our compiler by analyzing the code where the LSQ is used. Store values carry an optional *valid* bit, allowing the LSQ to support speculative memory requests, which we exploit in Chapter 5.

4.4 LSQ Design

We now present the design of our LSQ, showing how loads and stores are executed. We also show how our LSQ can support speculative memory requests, a feature that we exploit in the next chapter.

4.4.1 LSQ Overview

Figure 4.2 shows an overview of our LSQ, protecting a memory with one load and one store port. Load/store request queues and the store commit queue are implemented as shift registers, obeying FIFO order. Shift-register-based queues have a lower critical path compared to circular buffer based queues used in previous LSQ designs for HLS [125, 112]. Our store queue is broken up into two separate queues: one for requests and one for commits. Store-forwarding is the only latency-critical logic in our LSQ, and decoupling it allows the rest of the LSQ to be pipelined (such decompositions have been proposed before for CPU LSQs [20], but never for LSQs used in HLS). We implement store-to-load-forwarding using a store commit queue, which holds store address, value pairs for the duration between store issue and memory commit—the store commit queue size must be equal to the maximum store latency.

Our LSQ accepts one load request per cycle for every available load port to memory. Multiple load request sequences can be served in parallel as long as the number of sequences is not greater than the number of load ports. If there are more load request sequences than available load ports to memory, then the sequences are multiplexed according to program order (as is the case in Figure 4.2). Multiple store request sequences, and their corresponding store value sequences, are always multiplexed in program order, regardless of the amount of memory store ports available. Using a single store port protects us against Write After Write (WAW) hazards by construction, since the store requests/values arrive in program order to the LSQ.

4.4.2 Load Execution

A given $loadRequest_i$ at the head of the load request queue compares its tag to the tag of the last accepted store request, and waits if its tag is higher than the store tag. This tag check ensures that all store requests coming before $loadRequest_i$ in program order have arrived to the LSQ. Next, $loadRequest_i$ checks all store requests in the store request queue for conflicts using Eq 4.1. This involves comparing the load address against the address of all requests in the store requests queue—this is the most expensive operation in our LSQ. In previous LSQs, this check needed to be done in a single clock cycle, significantly increasing the circuit critical path [125]. we do not have such a restriction. Because the LSQ is used in a DAE architecture, where the production of load requests runs well ahead of their consumption, and because our store request queue is decoupled from the store commit queue, we have the cycle budget to pipeline this check and avoid an increase in the circuit critical path.

If the load finds no conflicts in the store requests queue, then the store commit queue is checked next. At this point, the store commit queue is guaranteed to hold only stores that come before $loadRequest_i$ in program order. In the commit queue, we check from the youngest to the oldest store and forward the first (i.e., youngest) store value that matches the $loadRequest_i$ address. This check is not pipelined, to decrease the latency between the production of a store value and its forwarding to a dependent load. If there are no hits in the commit queue, then we can safely issue the load request to the memory system. Eventually, a load value (either forwarded or loaded from memory) is returned to the LSQ client via a non-blocking, latency-insensitive channel.

4.4.3 Store Execution

A given $storeRequest_j$ at the head of the store request queue waits for its corresponding store value to arrive. On the arrival of the awaited store value, a store is immediately issued to memory and a $stCommit_j$, holding the store address and store value, is shifted into the store commit queue. The corresponding $storeRequest_j$ is shifted away from the store request queue. A store can only be in the store request or store commit stage, but never both. The store commit queue is sized such that it holds on to the store value until it is guaranteed to have been committed to memory.

4.4.4 Speculation Support

Our LSQ can support speculative store requests by extending each store value with a valid bit. Valid store values are handled without change. Invalid store values are not stored to memory and are not shifted into the store commit queue, but they still cause the corresponding store request to be shifted away from the store requests queue. This mechanism allows to speculatively allocate store addresses to the LSQ with no requirement for replays because *a misspeculated store request is never actually committed.* The next chapter shows how the compiler can create speculative requests, enabling more memory parallelism in codes that could previously not benefit from an LSQ.

4.4.5 Considerations for Memory Technology

Our LSQ design can be used to protect both on-chip memory (e.g., BRAM on FPGAs) and off-chip memory (e.g., DRAM) from data hazards. Our LSQ can use multiple load ports in parallel. Multiple store ports cannot be exploited by our design—to automatically protect WAW hazards, we multiplex multiple store sequences onto one store port.

To support multi-cycle memory (e.g., DRAM) we grow the size of the 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 added latency.

4.5 Compiler Generated DAE Architecture

In this section, we show how an FSMD HLS compiler can use our LSQ, and how to enable dynamically scheduled out-of-order loads in FSMD HLS by automatically generating a DAE architecture. Figure 3.3 from the previous chapter shows an abstract view of a DAE architecture, consisting of: an AGU, which sends memory requests to our LSQ; an LSQ, which executes loads and stores; and a Compute Unit (CU), which implements the computation part of the circuit, consuming load values and producing store values.

4.5.1 LSQ Placement

We use the compiler analysis described in Section 3.2.2 to find memory base addresses with data hazards. Each selected base address uses its own LSQ (in stark contrast to a CPU LSQ, which is shared by the entire core, putting pressure on the queue sizes). All memory operations using a selected base address are transformed into read/writes from/to latency-insensitive channels connected to an LSQ. The channels to an LSQ can be reused across basic blocks if they are guaranteed not to execute in the same clock cycle, similar to how FPGA block RAM ports can be shared—this information can be directly obtained from the schedule of the FSMD component that uses these channels.

Our LSQ design uses integer tags to recover program order of memory operations. Each address generating unit has a tag corresponding to a single LSQ, which is initialized to zero at the start of the circuit execution. Store requests increment the tag before using it, while load requests use the tag directly. This creates a data dependency between a store request $storeRequest_j$ and any other LSQ request that follow $storeRequest_j$ in program order, thus ensuring the correct order of the store request sequence.

4.5.2 Generating a DAE Architecture

The throughput of circuits using an LSQ depends on the number of addresses that can be disambiguated ahead of their actual memory operation execution—this number is often referred to as the *out-of-order address window*. In a statically scheduled pipeline, the out-of-order address window can be at most one—address generation and memory access proceed in lockstep. In Dynamic Dataflow (DDF) circuits, the generation of memory addresses is naturally decoupled from the memory operation and allows for much larger out-of-order address windows, essentially only limited by the buffer sizes of the latency-insensitive channels. To achieve the same effect in FSMD HLS, we decouple the generation of memory addresses into a separate static pipeline, similar to the principle of DAE architectures [205, 103, 42]. The AGU will contain only address generating instructions and will run ahead w.r.t. the CU, increasing the out-of-order address window in our LSQ.

Our compiler implements a DAE architecture automatically using the following steps:

1. **AGU:** Starting with the original code, for each memory operation to be decoupled, we change it to a `send_ld_request/send_st_request` function that sends memory address, tag pairs to the **LSQ**.
2. **CU:** Dually, the CU starts with the original code, but each memory operation to be decoupled is changed to a `consume_value/produce_value` function that receive/send values to/from the DU.
3. **Dead Code Elimination (DCE):** We run a standard DCE pass in the CU to remove the now unnecessary address generation code. In the AGU, we delete all side effect instructions that are not part of the address generation def-use chains, and then also run a standard DCE pass. We also use a control-flow simplification pass that removes empty blocks potentially created by DCE.

4.5.3 The Loss of Decoupling Problem

In some cases, address generation decoupling cannot result in the run-ahead of address requests. Such LoD events [103] arises when the address generation for a given base address depends on values loaded from the same base address, i.e., a load value from an array is used to generate a load/store address to the same array.

Definition 4.1 (General Loss of Decoupling)**.** 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 LoD 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; or there is a path from a branch instruction k to j, such that the basic block containing i is control-dependent on the basic block containing k, and k is not a loop branch.

Address decoupling is not possible in such cases, because the address requests and their memory operations need to be in effect synchronized. This is not a drawback of using a DAE architecture and FSMD HLS since a fully dynamically scheduled DDF circuit would also have to synchronize the two sequences.

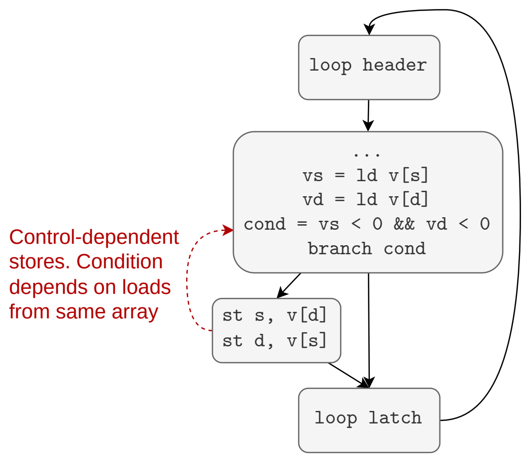
The loss of decoupling resulting from control dependencies can be solved with speculation, which our LSQ supports. In this chapter, we will only give an informal intuition of how such speculation can be implemented in a DAE architecture. In the next chapter, we will study the LoD problem in more detail and provide a formal mechanism for general compiler support for speculation in DAE architectures. By using speculative memory requests in the next chapter, we will only have to consider direct data dependencies, ignoring control dependencies in definition 4.1 and allowing us to maintain a high out-of-order address window, even in cases where a fully dynamic HLS compiler would suffer from loss of decoupling.

4.5.4 Intuition for Speculative Memory Requests

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. Consider the code in Figure 4.3a as an example. Here, the execution of the stores to `v` is control dependent on the *if*-condition, which itself uses values loaded from `v`. Under the execution model of both DDF HLS [126] and our DAE, there is no possibility for out-of-order memory operations in this code—the Figure 4.3a code has a LoD event due to the control dependency. We propose the concept of *speculative address requests* to relax this restriction.

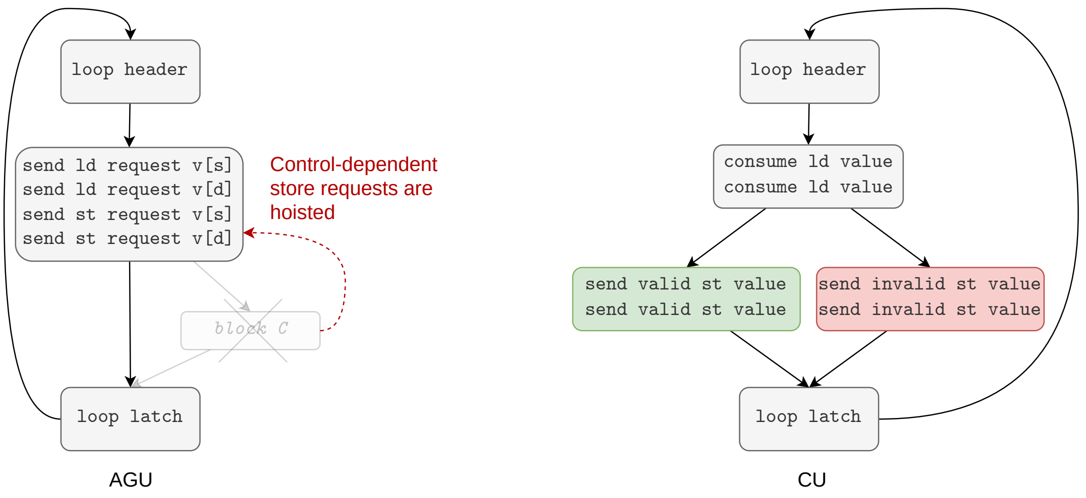
Consider the code in Figure 4.3a again. Although the store execution is control-dependent, the store addresses have no data dependency on values loaded from `v`. We can hoist the address instructions out of the *if*-condition in the Control Flow Graph (CFG) of the AGU, as illustrated in Figure 4.3c. As a result, store address allocations will be produced without having to evaluate the *if*-condition.

```
for (int j = 0; j < num_edges; ++j) {
    int s = e[j].src;
    int d = e[j].dst;
    if (v[s] < 0 && v[d] < 0) {
        v[s] = d;
        v[d] = s;
    }
}
```



(a) Maximal matching code.

(b) Maximal matching CFG.



(c) Our transformation: speculative address requests in the AGU, and invalidated store value writes on misspeculation in the CU.

Figure 4.3: Speculative store address requests in the maximal matching graph code.

Store requests sent to the LSQ, but later not used, are said to be *misspeculated*. Misspeculations are handled in the CU CFG by inserting invalid LSQ store value writes on CFG paths containing misspeculations, as illustrated in Figure 4.3c. An invalid LSQ store has the *valid* bit set to 0 and will result in the deallocation of the misspeculated address allocation in the LSQ (Section 4.4.2 describes the LSQ support). Handling misspeculated loads is trivial, since a load does not have side effects (at least in our execution model) and the loaded value can simply be discarded.

This compiler speculation approach can achieve a high degree of out-of-order loads on codes such as in Figure 4.3a, without having to suffer the cost of expensive misspeculation replays common in load-value-based speculation approaches. In the next chapter, we will show how the compiler can guarantee that order of store values generated in the CU (valid or invalidated) matches the order of the store requests generated in a speculative AGU.

4.6 Compiler LSQ Parametrization

Apart from the optimizations aimed at improving the LSQ critical path described in the previous section, we also let the compiler parameterize the LSQ queue sizes to achieve ideal throughput for a given code, while using the least amount of area possible.

As mentioned in the previous section, the size of the store commit queue should be equal to the maximum possible store latency. The HLS tool is aware of the target memory system, so this information is available directly to the compiler.

The optimal sizing of the LSQ store request queue requires more work, because it depends on the target loop II. Assume a target II of 1, and a loop datapath as presented in our motivating example in Figure 3.1a. 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 requests to cover all store addresses for the $[N, N+L]$ iteration range. Thus, the optimal store request queue size in this case is equal to the maximum latency between a dependent load and a store, call this $maxLoadToStoreDelay$ (for most codes, this is equal to the recurrence constrained II obtained using Equation 3.1). The optimal size will increase if there are multiple stores in the loop datapath; call the number of stores $numStoresInLoop$. All of the above information is static, allowing us to find an optimal store request queue size at compile time:

$$storeQueueSize = \left\lceil \frac{maxLoadToStoreDelay}{targetII} \times numStoresInLoop \right\rceil \tag{4.2}$$

Table 4.1 in the evaluation section shows how the resource usage and critical path of our LSQ scales with the size of the store request queue, showing that by using lower queue sizes we are able to save significant circuit area.

4.7 Deadlock Freedom

The original DAE paper [205] discussed that deadlocks are impossible to arise in their CPU-based implementation if a given instruction interleaving of the access and execute threads is followed. In this section, we extend their discussion to our specific context of a DAE architecture realized on a spatial compute platform. We prove that our compiler-generated DAE architecture cannot result in deadlock by proving that none of the blocking channel operations in the AGU and CU FSMD will ever block forever. In the next chapter, we extend this proof to include speculative memory requests.

In this discussion, we treat the LSQ as a black box, assuming that it itself can never deadlock if the AGU does not deadlock—the LSQ will always eventually accept a given load request, store request, or store value; and for every accepted load request, it will always eventually return the corresponding load value. Specifically, given a $loadRequest_i$ with $tag = k$, the

LSQ will return a $ldValue_i$, if $storeRequest_j$ with a $tag = l, l \geq k$ has arrived to the LSQ; and given a $stValue_j$, the LSQ is guaranteed to free the store queue entry corresponding to $storeRequest_j$. These LSQ properties can be formalized using linear temporal logic [19], but this is beyond the scope of this discussion (the proof would be similar to proofs for CPU LSQs).

Figure 3.3 shows the communication pattern between AGU, CU, and LSQ components after our transformations. First, notice that there is a duality between the channels in the CU and AGU. In both components, the same memory instruction is replaced with either a channel read or a channel write call to the same channel; each channel operation in the AGU has a counterpart in the CU. This gives us two tuples of channel operations, where the first element belongs to the address generation kernel and the second to the compute kernel:

$(loadRequest_i, ldValue_i)$,
$(storeRequest_j, stValue_j)$.

These tuples have two properties which we use in the proof, and which are guaranteed by our design:

Property 1: Every channel in the AGU has equivalent control flow to its counterpart channel in the CU.
Property 2: The channel in the CU will block until its counterpart channel in the AGU has finished writing. Thus, a loop iteration *i* of the CU cannot finish before the iteration *i* of the AGU has finished.

Theorem 4.1 (DAE Deadlock Freedom)**.** *None of the blocking channel operations introduced by our Section 4.5.2 transformation and depicted in Figure 3.3 can result in a deadlock of the DAE architecture.*

Proof. There three cases where deadlock might arise, making theorem 4.1 false. Each case involves a component waiting forever to read from an empty channel, or waiting forever to write to a full channel. We show that none of the three cases can arise in our DAE architecture. We prove theorem 4.1 for specific points in time, i.e., for specific load and store requests in a sequence of load and store requests. Extending this to the entire request sequence using induction is trivial and is left out of the proof.

Case 1 The $loadRequest_i$ channel write and the $ldValue_i$ channel read cannot result in deadlock because of property 1, i.e., for each load request write in the AGU, there will be a load value read in the CU. Our LSQ guarantees that for a given $loadRequest_i$ with $tag = k$, $ldValue_i$ will be returned to the compute kernel if $storeRequest_j$ with $tag = l$, $l \geq k$ has been accepted by the LSQ. But property 2 tells us that since $loadRequest_i$ with $tag = k$ has arrived, $storeRequest_j$ with $tag = l = k$ must have already arrived.

Case 2 The $storeRequest_j$ channel write could deadlock if the LSQ cannot accept new store requests. The only scenario where this is possible is if the store queue is full. However, by our earlier assumption of no deadlocks inside the LSQ, our store queue is guaranteed to always eventually have available space. Assume for the sake of argument that the store queue is full and the AGU tries to send $storeRequest_j$. Since our ordering of stores follows sequential program order, it is guaranteed that all $storeRequest_k$ for $0 \leq k < j$ have already arrived to the LSQ. Property 1 tells us that if a $storeRequest_{j-1}$ has been accepted into the LSQ, $stValue_{j-1}$ is guaranteed to be supplied as well. Once a $stValue_{j-1}$ has been accepted into the LSQ, the store queue entry corresponding to $storeRequest_{j-1}$ will eventually be freed. Thus, it is guaranteed that the store queue will eventually have a free entry for the $storeRequest_j$ to be accepted, and the $storeRequest_j$ channel will not result in deadlock.

Case 3 The $stValue_j$ channel write could result in deadlock if the LSQ cannot accept new store values. The LSQ will not accept a $stValue_j$ only if the corresponding $storeRequest_j$ has not been previously accepted. But this is not possible because of property 2—if a $stValue_j$ channel write call was made, then a channel write call with $storeRequest_j$ must have been made in the AGU. □

4.8 Evaluation

In this section, we show that the close compiler-hardware co-design of our LSQ results in circuits with a lower critical path and lower area usage than previous LSQs for HLS. We also show that our speculative memory requests preserve address decoupling in codes where previous work suffers from LoD events, resulting in a significant throughput improvement of our LSQ over previous work on such codes. We show this by evaluating our LSQ against the DDF HLS compiler Dynamatic [126] that uses a state-of-the-art LSQ [125]. we also compare against two commercial FSMD HLS compilers (Intel HLS [120] and Vivado HLS [238]), to show that dynamically scheduled memory significantly improves circuit performance on irregular codes. We also discuss how our LSQ design scales with the size of its store allocation queue. Our implementation and this evaluation are available as a public artifact [211].

4.8.1 Methodology

We extend our implementation described in Section 3.4.1 with the generic LSQ template, and compiler analysis' and transformations described in this chapter. We automatically find data hazards in the input code, decouple the address generation into separate FSMD components (separate SYCL kernels), and connect memory requests to an LSQ specialized to the input code. The LSQ specialization involves instantiating a C++ template with parameters obtained from the compiler analysis'.

We evaluated our work against the DDF HLS tool Dynamatic using a research artifact from their most recent paper [77]. Cycle counts were obtained using ModelSim and are compared directly between all tools. Dynamatic uses Vivado for synthesis, while we use Intel tools, making a direct comparison of area and circuit frequency in absolute terms difficult. Instead, we again compare the normalized frequency, execution time, 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 codes using on-chip BRAM; DRAM accesses are not supported in Dynamatic.

We applied our approach to benchmarks with data hazards used in previous work [47, 126]. The first four benchmarks are the same as in the previous chapter and do not require speculation (described in table 3.1). These additional three benchmark codes also do not need speculation to preserve address decoupling.

- *vecTrans* applies a polynomial expression on elements of a sparse array.
- *spmv* is a sparse matrix-vector multiply.
- *chaosNCG* is a function from a chaos engine with data-dependent loads and stores.

The remaining benchmarks have control-dependent stores, making our speculative address allocation approach applicable:

- *histogramIf* is similar to *histogram*, but the store is control dependent on the load value.
- *matching* is the code example from Figure 4.3a.
- *floydWarsh* finds shortest paths in a weighted digraph.
- *sort* sorts a list of integers using a bitonic merge network.

We report worst- and best-case performance, which depends on the true number of data hazards in the input data distribution. We automatically choose our store allocation queue size according to sec. 4.6. For Dynamatic, we manually choose the smallest queue size that enables perfect pipelining in the case of no data hazards, following their manual approach described in [151].

4.8.2 Results

Speedup

Figure 4.4 shows that our approach achieves a higher speedup than Dynamatic when comparing each tool to their respective static HLS baseline. On most codes, the higher speedup is due to the higher frequency achievable by our LSQ. On some codes (e.g. *chaosNCG*), we also achieve a better throughput than Dynamatic, because we can support the required large store queue size and Dynamatic cannot. The maximum store queue size that we could use in Dynamatic was 32 (Table 4.1).

Table 4.3 at the end of this chapter shows detailed benchmark results. On average, designs with our LSQ achieve 90% of the frequency achieved by Intel HLS, whereas Dynamatic LSQ designs achieve a frequency of 35% compared to Vivado. Dynamatic sees a higher throughput overhead when the data distribution favors static scheduling (more iterations with a true data hazard), resulting in an average 1.4× more cycles to finish than the Vivado designs,

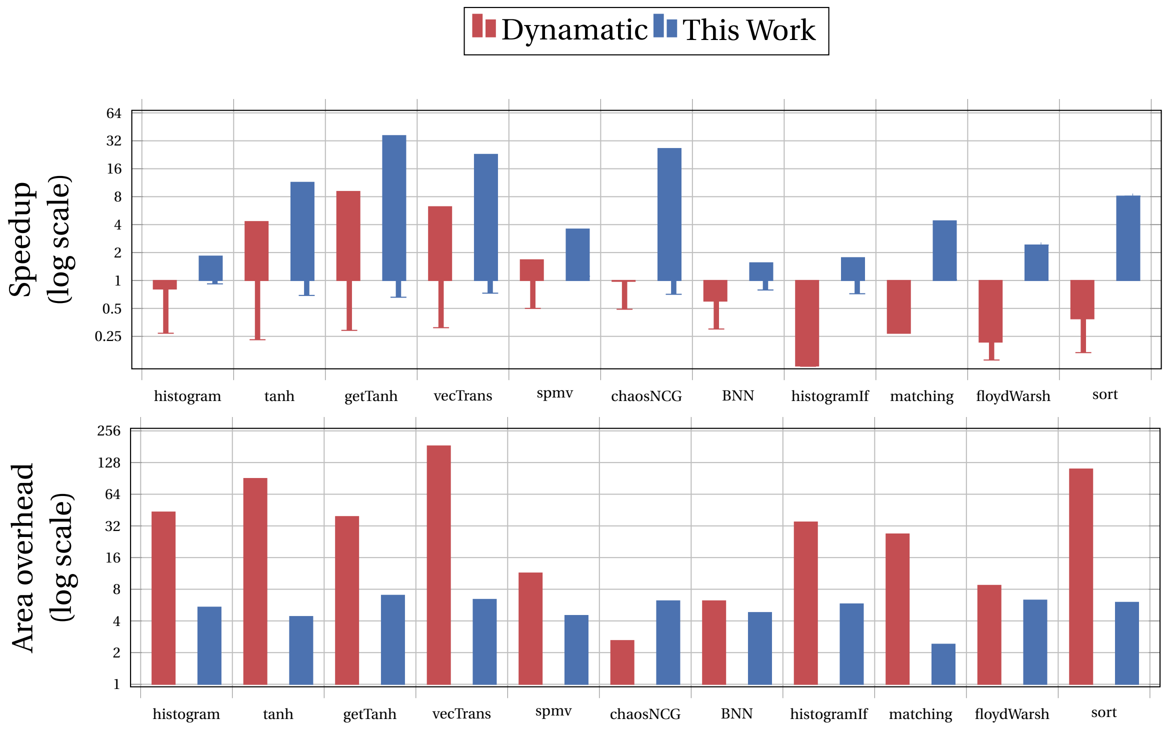


Figure 4.4: Speedup and area overhead of our work and Dynamatic [77] compared to their static HLS baselines (Intel HLS and Vivado, respectively). The range bars represent the speedup range, with a value below 1 indicating a slowdown.

rising to 3.67× more execution time due to their lower frequency. On average, our approach has no overhead in execution time compared to its Intel HLS baseline. The slight overhead in the number of cycles for some codes is only for data distributions that repeatedly read and write to the same memory location, which is a highly unlikely scenario.

The last four codes benefit from our speculation scheme, allowing us to preserve address decoupling and enabling speedups over our static HLS baseline. The dataflow circuits produced by Dynamatic suffer a LoD problem on these codes and do not result in any throughput improvements compared to a static pipeline.

Area Overhead

In addition to a better speedup, our LSQ also has a lower area overhead than Dynamatic. On average, we see a 4.9× area overhead compared to 12.3× for Dynamatic, and that is despite the fact that for several codes we use larger LSQ queue size.

Store Queue Scaling

Table 4.1 shows how the frequency and area usage changes with the size of our store allocation queue. Previous LSQ designs targeting FPGAs are notorious for their poor scalability [236, 151]. Our LSQ scales better, allowing for store queues with hundreds of entries. Even though larger store queues still degrade the achievable circuit frequency, the degradation is sub-linear and is more than compensated by the increased potential throughput compared to statically scheduled memory accesses. For example, for a 256 entry store queue, the circuit frequency drops by 2×, but the potential throughput increases by 256× in the best case.

Table 4.1: Scalability of our store request queue compared to the store queue in Dynamatic [77] on the histogram benchmark.

Queue Size	**Freq (MHz)**				**Area (Slices / ALMs)**			
	Dyn	×	Ours	×	Dyn	×	Ours	×
No LSQ	379	**1**	379	**1**	129	**1**	1814	**1**
2	173	**0.46**	338	**0.89**	409	**3.2**	9155	**5**
4	178	**0.47**	337	**0.89**	684	**5.3**	9305	**5.1**
8	163	**0.43**	331	**0.87**	1554	**12**	9847	**5.4**
16	155	**0.41**	313	**0.83**	5582	**43**	10705	**5.9**
32	92	**0.24**	271	**0.72**	22580	**175**	12509	**6.9**
64	-	-	274	**0.72**	-	-	14140	**7.8**
128	-	-	258	**0.68**	-	-	23623	**13**
256	-	-	195	**0.51**	-	-	39598	**22**

4.8.3 Using Off-Chip DRAM

Table 4.2 at the end of this chapter shows the speedups over static Intel 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 dual-channel DDR4 memory. On average, using our LSQ results in an 4–10× speedup over Intel HLS. The store commit queue, needed to cover the maximum store latency to DRAM, has a cache-like effect which is more noticeable in DRAM codes, compared to codes using BRAM. As a result, our LSQ still offers a significant speedup even if most of the iterations have a true data hazard.

Circuits with DRAM connections use more resources, making the area overhead of our LSQ smaller (1.4× for DRAM vs. 3.4× for BRAM). For some codes, using our LSQ results in virtually no resource increase. This is because the Intel HLS baseline uses more costly bursting DRAM load-store units, while we use simpler, pipelined units.

The DRAM benchmarks achieve on average a 7–10× lower throughput than the BRAM codes. Our DRAM load-store units do not take advantage of DRAM coalescing that amortizes the large off-chip memory latency in typical HLS designs by coalescing multiple memory requests into one wide request that uses the whole DRAM width. For example, instead of sending a store request to the DRAM memory controller for every 32 bit value, we could accumu-

Table 4.2: Performance comparison of our LSQ against Intel HLS when protecting off-chip DRAM.

Kernel	Exec. Time (μs)			Freq. (MHz)			Area (ALMs)		
	I	O	O/I	I	O	O/I	I	O	O/I
histogram	363	43.7–61.3	**0.12–0.17**	273	272	**1**	19832	19647	**1**
tanh	564	36.9–150	**0.08–0.27**	281	205	**0.73**	27365	35559	**1.3**
getTanh	396	35.8–122	**0.09–0.31**	281	235	**0.84**	26018	29051	**1.1**
BNN	4167	336–636	**0.08–0.15**	264	241	**0.91**	10916	17459	**1.6**
vecTrans	441	40.6–182	**0.09–0.37**	305	241	**0.79**	20217	22814	**1.1**
spmv	158	40.8–63.5	**0.26–0.34**	287	256	**0.89**	7826	18313	**2.3**
chaosNCG	687	63.3–502	**0.09–0.54**	270	170	**0.63**	21190	37314	**1.8**
histogramIf	362	34.2–61.8	**0.09–0.17**	274	248	**0.91**	19903	20950	**1.1**
matching	496	53.5–175	**0.11–0.35**	289	227	**0.79**	8655	19907	**2.3**
floydWarsh	300	59.9–98.4	**0.21–0.33**	257	250	**0.97**	31280	32173	**1**
sort	319	33.9–53.3	**0.11–0.17**	270	241	**0.89**	12587	24781	**2**
hmean			**0.1–0.25**			**0.84**			**1.4**

I—Intel HLS O—Our work

late 16 such values into one request to use the entire 512 bit DRAM channel width. It is unlikely that DRAM coalescing could be used effectively in an LSQ, because the memory access pattern of codes using LSQs is seldom contiguous. The issue of memory coalescing will be explored further in Chapter 6.

We challenge this limitation in Chapter 6 by proposing an alternative memory disambiguation method to the LSQ that is able to take advantage of such coalesced DRAM requests, in addition to having other properties helpful in increasing memory parallelism in HLS.

4.9 Conclusion

In this chapter, we have presented a novel, shift-register-based 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. We have also presented how a compiler can automatically generate a DAE architecture, which together with our LSQ enables dynamically scheduled, out-of-order memory loads in FSMD HLS. We have shown that the DAE approach relaxes the latency requirement in our LSQ, allowing us to use more pipeline stages for a better critical path. Additionally, we have shown that the close compiler-hardware co-design of various parts of the LSQ decreases the amount of resources used by our LSQ. Our LSQ design achieves a higher frequency and lower area overhead compared to previous LSQs used in HLS, resulting in an average speedup of 11× compared to static HLS and 5× compared to dynamic HLS. Our LSQ scales to queues with hundreds of entries, and can protect both on-chip and off-chip memory.

One limitation of our DAE approach is the fact that decoupling the address generating instructions from the rest of the program is not always possible due to dependencies between the AGU and the CU that cause synchronization (Section 4.5.3). In this chapter, we have shown that speculative memory requests generated in the AGU can be an effective way to mitigate most LoD problems. We have shown how our LSQ can support such speculation and we gave an informal intuition how the compiler can transform the AGU and CU CFGs to support speculation in a DAE architecture. However, from our discussion it is not clear that speculation support is general enough to apply to any CFG. We tackle this problem in the next chapter by proposing a general compiler support for control speculation in DAE architectures.

Table 4.3: A comparison of our work against Vivado, Dynamatic [77], and Intel HLS. All codes use on-chip BRAM.

Kernel	**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**	5.5	3–6	**0.55–1.1**	129	5582	**43.3**	1814	9847	**5.4**
tanh	14	1–19	13.1	1–17	304	96	330	297	46.1	10.7–198	**0.23–4.3**	39.8	3.5–57.3	**0.09–1.44**	245	22103	**90.2**	3803	16730	**4.4**
getTanh	68	2.5–79	56.2	1.1–59	266	89	377	346	263	28.1–888	**0.11–3.47**	149	4.1–224	**0.03–1.51**	572	22399	**39.2**	1825	12753	**7**
BNN	20	15–30	20.7	10.4–20.4	258	116	365	284	77.5	129–259	**1.67–3.34**	56.9	36.8–72	**0.65–1.26**	1214	7466	**6.2**	4214	20222	**4.8**
vecTrans	30	1.5–31	30.1	1.1–33	304	97	365	291	98.7	15.9–320	**0.16–3.24**	82.5	3.6–113	**0.04–1.38**	125	22997	**184**	1811	11672	**6.4**
spmv	2.3	0.8–2.7	3.6	0.8–2.7	263	152	328	280	8.7	5.2–17.6	**0.6–2.02**	10.9	3–9.8	**0.28–0.9**	494	5628	**11.4**	5255	23406	**4.5**
chaosNCG	72	37–74	74.3	2.1–77	308	155	335	246	234	239–477	**1.02–2.04**	222	8.4–313	**0.04–1.41**	779	2017	**2.6**	5274	32960	**6.2**
histogramIf	2	5–6	2.1	1-2.5	388	117	379	328	5.15	42.7-51.3	**8.29–8.3**	5.5	3.1–7.7	**0.57–1.4**	155	5395	**34.8**	1814	10452	**5.8**
matching	6	6–8	7.6	2–8.8	404	110	246	291	14.9	54.6–72.7	**3.67–4.9**	30.9	7–30.2	**0.23–0.98**	141	3778	**26.8**	7713	18310	**2.4**
floydWarsh	6.2	7–11	6.3	3.4	366	90	229	299	16.9	77.8–122	**4.59–7.2**	27.3	11.3	**0.42**	255	2226	**8.7**	807	5056	**6.3**
sort	3.1	2.6–6.1	9.6	1.5	300	97	248	305	10.4	26.9–62.8	**2.58–6**	38.8	4.8	**0.12**	51	5683	**111**	911	5424	**6**
hmean		**0.15–1.4**		**0.07–0.64**		**0.35**		**0.9**			**0.45–3.67**			**0.09–0.62**			**12.3**			**4.9**

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

Chapter 5

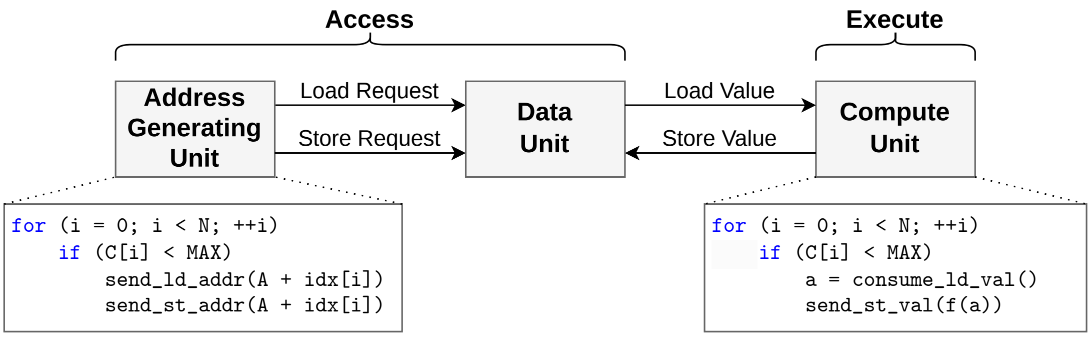
Compiler Support for Speculation in DAE Architectures

In this chapter, we focus on the speculation technique introduced in the previous chapter describing our Load-Store Queue (LSQ) design and automatic Decoupled/Access Execute (DAE) architecture generation. We propose general compiler support for speculation in DAE architectures. We make the claims made in Section 4.5.4 concrete by showing that our speculation approach works on arbitrary, reducible control flow and we prove that it preserves the sequential consistency of the original program.

DAE is a common technique used in CPU/GPU prefetchers and specialized accelerators to tackle the problem of memory and communication latencies in irregular codes. The technique relies on the compiler to separate memory address generation from the rest of the program, but such a separation is not always possible due to control and data dependencies between the access and execute slices, resulting in a Loss of Decoupling (LoD) event. We present compiler support for speculation in DAE architectures that preserves decoupling in the face of control dependencies.[1] We propose algorithms that implement speculative memory requests in the access slice and that kill mis-speculations in the execute slice without the need for costly recovery or synchronization.

Altough we focus explicitly on HLS generated accelerators in this thesis, the DAE principle is widely used in the computer architecture field. We show that our speculation compiler support, in addition to HLS accelerators, applies to a wide range of architectural work on CPU/GPU prefetchers and Coarse Grained Reconfigurable Arrays (CGRAs), citing concrete examples from the literature. Supporting DAE speculation makes these works applicable to a wider range of codes than before. In our evaluation on HLS accelerators that use our LSQ, we show that our speculation support allows us to preserve address decoupling and thus achieve a higher level of memory parallelism compared to using our LSQ without speculation.

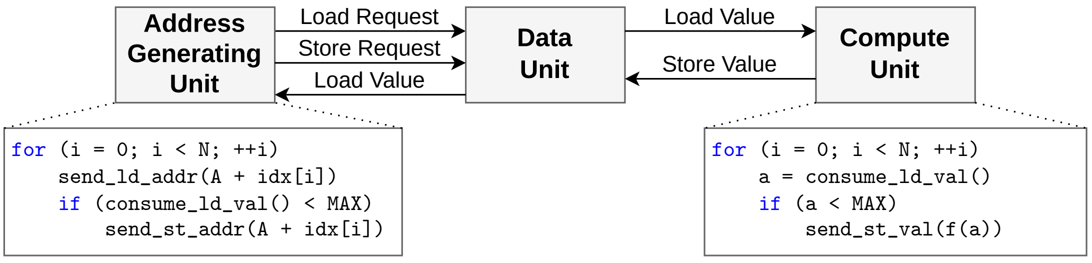
1. LoD due to data dependencies can still occur, but this is not as common as control dependencies.



```
for (i = 0; i < N; ++i)
    if (C[i] < MAX)
        send_ld_addr(A + idx[i])
        send_st_addr(A + idx[i])
```

```
for (i = 0; i < N; ++i)
    if (C[i] < MAX)
        a = consume_ld_val()
        send_st_val(f(a))
```

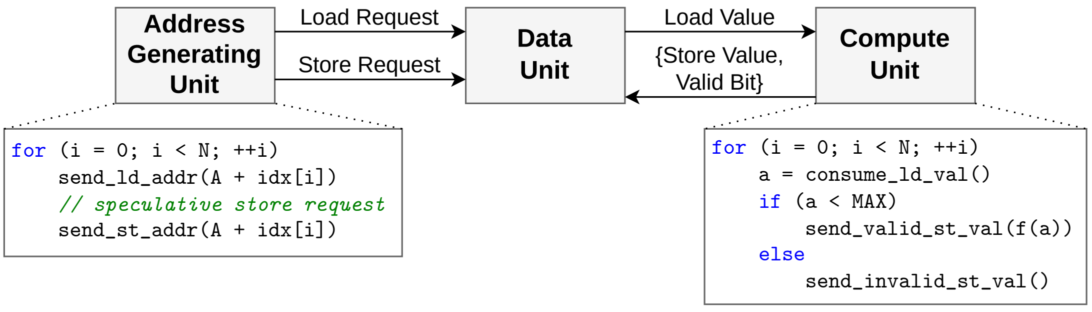
(a) An architecture with decoupled address generation, memory access, and compute.



```
for (i = 0; i < N; ++i)
    send_ld_addr(A + idx[i])
    if (consume_ld_val() < MAX)
        send_st_addr(A + idx[i])
```

```
for (i = 0; i < N; ++i)
    a = consume_ld_val()
    if (a < MAX)
        send_st_val(f(a))
```

(b) Loss-of-decoupling between the AGU and memory access due to a dependency on the memory value.



```
for (i = 0; i < N; ++i)
    send_ld_addr(A + idx[i])
    // speculative store request
    send_st_addr(A + idx[i])
```

```
for (i = 0; i < N; ++i)
    a = consume_ld_val()
    if (a < MAX)
        send_valid_st_val(f(a))
    else
        send_invalid_st_val()
```

(c) Our contribution in this chapter: compiler support for speculation removes loss-of-decoupling due to control dependencies.

Figure 5.1: A generic DAE architecture template. In this thesis, the AGU and CU are implemented as separate FSMD components in reconfigurable hardware; another implementation might use CPU/GPU threads, or separate PEs in a CGRA architecture.

5.1 Introduction

As alluded to in our thesis introduction chapter, irregular codes are common in domains like graph analytic and sparse linear algebra. They are characterized by data-dependent memory accesses and control flow, for example:

```
for (int i = 0; i < N; ++i)
    if (C[i] < MAX)
        A[idx[i]] = f(A[idx[i]]);
```

This code has unpredictable control flow that causes frequent branch mis-predictions on CPUs and thread divergence on GPUs. Because of these limitations and challenges with Moore's Law and Dennard performance scaling computer architects are interested in adding CPU/GPU support to accelerate such code patterns, or even to use accelerators specialized for a given algorithm [108].

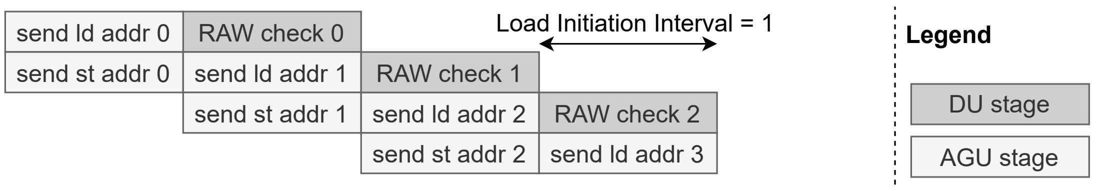
Many of the proposed architectures follow the decades-old idea of a DAE architecture shown in Figure 5.1. In DAE, memory accesses are *decoupled* from computation to avoid stalls resulting from unpredictable loads [205]. The Address Generation Unit (AGU) sends load and store requests to the Data Unit (DU), while the DU sends load values to and receives store values from the Compute Unit (CU). All communication is FIFO based and ideally the AGU to DU communication is feed-forward (one-directional), allowing the address streams from the AGU to run ahead w.r.t the CU. Figure 5.1a shows an example of such a DAE architecture implementing the earlier code snippet.

As the gap between memory speeds and compute has grown over the years, the importance of the DAE technique has only increased. DAE is a general technique applicable to many computational models: it is used in specialized FPGA accelerators generated from HLS [42, 41, 51, 54, 44, 89, 212, 216]; in CGRAs [166, 188, 187, 83, 111, 180, 233, 171]; and in CPU/GPU prefetchers [60, 167, 104, 9, 6, 59, 190]. For example, recently NVIDIA introduced hardware-accelerated asynchronous memory copies in the Ampere architecture and later extended the idea to the Tensor Memory Unit (TMA) in the Hopper architecture [6]. The CUDA programmer can provide a "copy descriptor" of a tensor to copy and the hardware will run ahead and generate the corresponding addresses in the TMA.

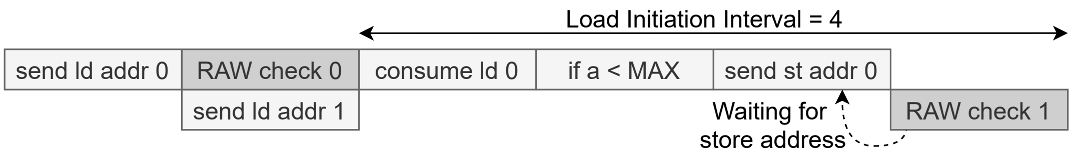
The common denominator of all these works is that they rely on either the programmer or the compiler to decouple address-generating instructions from the rest of the program. However, it has long been recognized that such a decoupling is not always possible [26, 223]. If any of the instructions generating an address for an array `A` depend on a value loaded from `A`, then there is a LoD event [103], a problem briefly described in Section 4.5.3 in the previous chapter. Access patterns such as `A[f(A[i])]` are rare, but control dependencies that involve loads from `A` are commonplace. For example, consider replacing `C[i]` with `A[i]` in our running example:

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

Here, there is a LoD, because the store to `A` is control-dependent on a branch that loads from `A`. Whereas before the load from `C` could be prefetched, now the AGU/DU communication is synchronized, because the AGU waits for `A` values from the DU before deciding if a store address should be generated (see Figure 5.1b). In turn, the load waits for the store address to ensure that there is no aliasing—the store address is needed for memory disambiguation. As



(a) Pipeline of decoupled address generation from figure 5.1a.



(b) Pipeline of non-decoupled address generation from figure 5.1b.

Figure 5.2: Comparison of a decoupled and non-decoupled address generation. Non-decoupled address generation results in a later arrival of the store address, which stalls the RAW check for the next load, lowering load throughput.

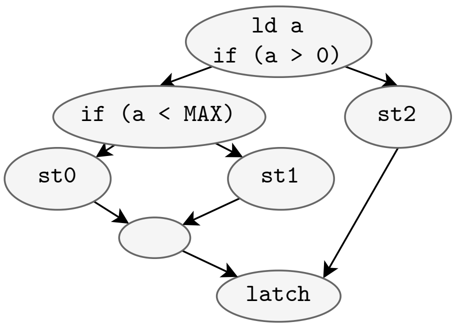
a result, the AGU cannot run ahead of the CU anymore, as illustrated in Figure 5.2 pipeline diagram. In the previous chapter, the code in Figure 4.3a and the last four codes in the evaluation section have this behavior, resulting in a stark performance degradation if speculation is not used (e.g., see the Dynamic Dataflow (DDF) HLS results in the Section 4.8).

One approach for restoring decoupling in these cases is control speculation. As shown in Figure 5.1c and briefly described in Section 4.5.4, we can hoist the store request out of the *if*-condition in the AGU (speculation), and later *poison* the store in the CU on mis-speculation (store invalidation). However, it is unclear how the compiler should coordinate the speculation and recovery transformations across two distinct control-flow graphs. While the example from Figure 5.1c is trivial, the task quickly becomes complicated with more speculated stores and nested control-flow, as we demonstrate in the Section 5.1.1. *The key challenge* here is to guarantee that the order of store requests sent from the AGU, matches the order of store values or kill signals sent from the CU on all control-flow paths.

General compiler support for speculated stores in DAE architectures is an open question that we tackle in this chapter, making the following contributions:

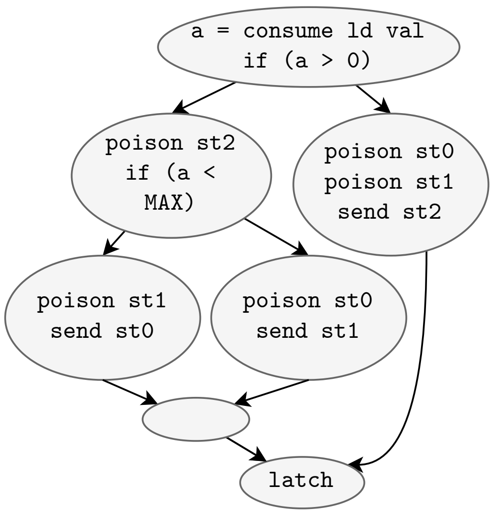
- We give a formal description of the fundamental reasons why address generation cannot always be decoupled from the rest of the program (Section 5.3).
- We describe compiler support for speculative memory in DAE architectures, effectively solving the LoD problem due to control dependencies. We propose an algorithm for introducing speculative memory requests in the AGU, and an algorithm for poisoning mis-speculations in the CU (Section 5.4).
- We prove that our speculation approach preserves the sequential consistency of the original program and does not introduce deadlocks (Section 5.5).
- We show that our work enables the use of DAE on a wider class of codes than before, with applications in CPU/GPU prefetchers, CGRAs, and accelerators.

```
for (i = 1; i < N - 1; ++i) {
    a = A[i];
    if (a > 0) {
        if (a < MAX1)
            A[i + 1] = a + 1;   // st0
        else
            A[i - 1] = a + 1;   // st1
    } else {
        A[i] = a + 1;           // st2
    }
}
```



(a) Code and CFG of a loop with three control-dependent stores causing a loss-of-decoupling.

```
// Assume AGU store request order was:
//     s2, s0, s1
for (i = 1; i < N - 1; ++i) {
    a = consume_ld_val();
    if (a > 0) {
        poison_st2()
        if (a < MAX1) {
            poison_st1();
            send_st0(a + 1);
        } else {
            poison_st0();
            send_st1(a + 1);
    } else {
        poison_st0();
        poison_st1();
        send_st2(a + 1);
    }
}
```



(b) Code and CFG of CU implementing an incorrect misspeculation strategy. Depending on control flow, the order of store values can be: (s_2, s_1, s_0), (s_2, s_0, s_1), (s_0, s_1, s_2), but only (s_2, s_0, s_1) is correct.

Figure 5.3: Poisoning speculated stores immediately when they become unreachable results in an ordering mismatch between AGU store requests and CU store values.

- We evaluate our DAE speculation approach on accelerators generated from HLS implementing codes from the graph and data analytics domain. We achieve an average 1.9× (up to 3×) speedup over the baseline HLS implementations. We show that our approach has no mis-speculation penalty and minimal code increase impact with an average accelerator area increase of < 5% (Section 5.7).

5.1.1 Motivating Example

Here, we briefly show why an obvious approach to speculation in DAE architectures is incorrect, motivating the contributions of this chapter.

The FIFO-based nature of DAE requires that the order of memory requests (speculative or not) generated in the AGU matches exactly the order of load/store values (poisoned or not) in the CU. The motivating example in Figure 5.1c contains just one speculative store and one path through the compute Control Flow Graph (CFG) where the speculation becomes unreachable, making the problem of ordering trivial in that case.

Consider the more complex code in Figure 5.3a with three stores s_0, s_1, and s_2. Speculating all store requests in the AGU might result in the store request order (s_2, s_0, s_1). In the CU, we need to guarantee the same order of corresponding store values (poisoned or not) on every possible control-flow path through the loop. Unfortunately, the obvious approach that worked for the trivial example in Figure 5.1c does not work here. If we poison values at points where the corresponding speculations become unreachable, as illustrated in Figure 5.3b, we end up with three possible orderings of store values depending on the CFG path in the CU, but only one of the orderings is correct. This is why any previous implementations of speculative stores in DAE architectures has only considered trivial triangle or diamond shaped CFGs [104], like the one in 5.1c. Generalized compiler support for store speculation that guarantees the correct order of poisoning is the *key challenge* that we solve in this chapter.

5.2 Architectural Support

In the previous chapter, we have described how our own LSQ for HLS can support speculative memory requests in Section 4.4, and we have show how a compiler can automatically generate a DAE architecture in Section 4.5. In this section, we describe the general architectural support needed to enable our approach to speculative memory requests in DAE architectures.

Our speculation technique requires architectural support for predicated stores and FIFOs to communicate between the AGU, DU, and CU. Store values communicated from the CU are tagged with a *poison bit* that, when set, causes the corresponding store request to be dropped in the DU without committing a store. We say that a store request gets killed (or poisoned) if its corresponding store value has the poison bit set. This is a lightweight form of speculation that does not require replays in the CU and does not result in out-of-bounds stores, because *mis-speculated stores are never committed.*

Predicated stores are cheap to support in hardware since the underlying memory protocol usually already uses a *valid* signal. Architectural FIFOs are also commonly added in works on CPU/GPU microarchitecture or can be relatively cheaply implemented in software; accelerators often use FIFOs as primitives. We discuss concrete examples of architectures that can benefit from our work in Section 5.6. Here, we give a general overview of how our work applies to these domains.

Predicated Stores in CPU/GPU Prefetchers

Academic works on DAE CPU/GPU prefetchers add architectural FIFO queues and extend the Instruction Set Architecture (ISA) with instructions for producing load/store addresses and consuming/producing store values [60, 167, 104, 9, 6, 59, 190]. The prefetcher from [104] supports predicated stores through a *store_inv* instruction, but the authors support speculation only on simple triangle or diamond control flow patterns, calling for future work on general speculation support.

Predicated Stores in Spatial Accelerators

Adding predicated stores to specialized spatial accelerators is trivial, and in many cases does not require any protocol changes, e.g., the commonly used AXI4 interface protocol [8] has a strobe signal to indicate which write data bytes are valid. We evaluate our work on accelerators generated from HLS, where we have complete control of the memory interfaces and we add predicated store support.

Speculative Loads

Speculative loads are relatively simple to support, because they are usually side-effect free in accelerators—a mis-speculated load can simply be discarded. A speculative load request could potentially have an out-of-bounds address, but this can be easily handled in the DU (e.g., by omitting out-of-bounds loads and instead returning dummy values).

5.3 Loss-of-Decoupling Analysis

As discussed in Section 4.5.3, LoD arises when the address generation for a given memory access depends on a load that cannot be trivially prefetched, causing the AGU, DU, and CU communication to be synchronized. By *non-trivially prefetched* we mean loads that have a RAW hazards, i.e., the DU needs to receive all previous store addresses in program order to perform memory disambiguation before executing the load. In Section 4.5.3, we have given a general definition of LoD, which did not distinguish between a control and data dependency. In this chapter, we are more precise and separate definitions for the two cases.

Given a set of address-generating instructions G, and a set of memory load instructions A using addresses generated by instructions in G, there is a loss of decoupling if:

Definition 5.1 (Loss of Decoupling due to a Data Dependency)**.** There exists a path in the def-use chain from $a \in A$ to $g \in G$. While encountering a ϕ-node on the def-use chain leading to g, we also trace the def-use paths of the terminator instructions T in the ϕ-node incoming basic blocks to see if any terminator instruction in T depends on any $a \in A$.

Definition 5.2 (Loss of Decoupling due to a Control Dependency)**.** There exists an instruction $g \in G$ that is control-dependent on a branch instruction b, and there is a path in the def-use chain from $a \in A$ to b.

Definition 5.3 (Loss of Decoupling Control Dependency Source)**.** If $g \in G$ experiences a LoD due to a Control Dependency on branch instruction b, then we can we call the basic block that contains b the *LoD control dependency source*. The LoD control dependency source need not be the immediate control dependency of g, and that g might have multiple LoD control dependencies.

Algorithm 5.1 Control-flow hoisting of AGU requests

```
1: Input: $srcBlocks$ list of blocks that are the source of a LoD control dependency
2: Output: $SpecReqMap$ {basic block: list of hoisted requests to this block}
3:
4: for $srcBB \in srcBlocks$ do
5:     for $fromBB \in reversePostOrder(srcBB)$ do        ▷ traverse DAG up to loop latch
6:         if $fromBB$ contains memory requests then
7:             hoist $fromBB$ requests to the end of $srcBB$
8:             add requests to $SpecReqMap[srcBB]$
```

Depending on the hardware context, the definition of the A set can be expanded or narrowed. For example, if the AGU is implemented in hardware with limited control flow support, then A could include all branch instructions. On the other hand, given an address generating instruction, we could limit A to only include loads from the same array for which the given address is generated—this could be useful if we only want to preserve decoupling for that array and do not care about losing decoupling for other arrays. Our speculation technique applies equally well to all these definitions.

An example of a LoD data dependency is the access `A[f(A[i])]`. Our speculation approach does not recover decoupling for such cases, but fortunately such code patterns are rare. An example of a more common LoD data dependency is the code pattern `if (A[i]) A[i++] = 1`. In this case, the def-use chain leading to the definition of the store address contains a ϕ-node defining the current value of `i`, which depends depends on a load from A. Such a pattern is sometimes found in algorithms that operate on dynamically growing data structures, e.g., queues or stacks. Our speculation technique does not work on such cases either, but this is not a large limitation, since performance oriented codes typically do not use dynamically growing structures, instead opting for implementations with bounded space requirements that can be allocated statically [241].

An example of LoD due to a control dependency is shown in Figure 5.1b. This case is much more common than a direct data dependency and is the focus of this chapter.

5.4 Compiler Support for Speculation

We now describe our dual compiler transformations that enable speculation in the AGU and poison mis-speculations in the CU.

5.4.1 Speculating Memory Requests

Algorithm 5.1 describes our approach to introducing speculation in the AGU. Given a LoD control dependency source block $srcBB$, we hoist all memory requests that are control dependent on $srcBB$ to the end of $srcBB$. There can be multiple blocks with memory requests that have a LoD control dependency on $srcBB$, which poses the question in which order should they be hoisted to $srcBB$. We use reverse post-order in Algorithm 5.1. Assuming reducible control flow, the CFG region from $srcBB$ to the loop latch is a DAG. The reverse post-order of a DAG is its topological order. Topological ordering gives us the following useful property.

CFG Topological Ordering Property

Given two distinct basic blocks A and B in a given loop, if $A \prec B$ in any path through the loop then $A \prec B$ in the topological ordering. Note that there can be multiple topological orderings for a DAG, but it does not matter which one is chosen in our algorithm.

Algorithm 5.1 traverses the CFG region from $srcBB$ to the end of its loop (or to the end of the function if $srcBB$ is not any loop). During the traversal, we ignore CFG edges leading to loop headers—we do not enter loops other than the innermost loop containing $srcBB$.

Hoisting Example

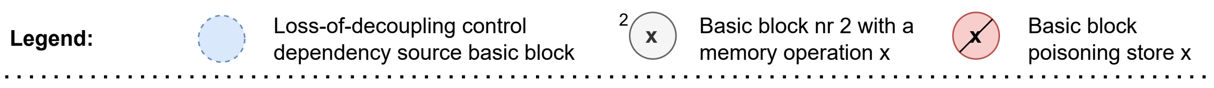
Consider the CFG from Figure 5.4a. There are three LoD control dependency source blocks (2, 3, 5) and five blocks with memory requests (blocks 2, 4, 5, 6, 7 with memory requests a, c, b, d, e, respectively). Assume that the blocks hold a single memory requests—multiple memory requests within the same block are treated in the same way by our algorithms. Figure 5.4c shows the topological order of the loop (block 1 is omitted for brevity). Algorithm 5.1 will hoist b, e to the end of block 2, and c, d, e to the end of block 3—the result is presented in Figure 5.4b. Note that the requests b and e were hoisted to both block 2 and 3, because they are reachable from both blocks. Nothing is hoisted to block 1 since it is not a LoD control dependency source.

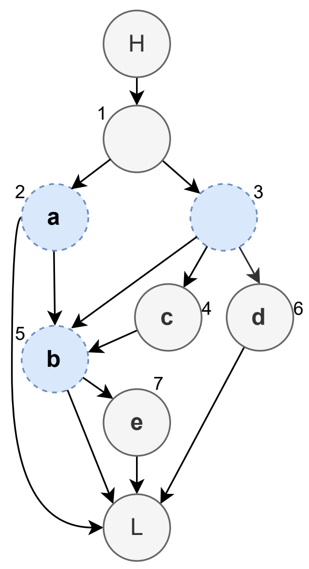
Nested LoD Control Dependencies

Block 5 in Figure 5.4b does not contain any speculative requests because it itself has a LoD control dependency on block 2 and 3. Algorithm 5.1 considers only LoD control dependency source blocks that are not themselves the destination of another LoD control dependency. Given a chain of nested LoD control dependencies, we only consider the chain head. For example, the Figure 5.4a CFG has two LoD control dependency chains: 2, 5 and 3, 5—Algorithm 5.1 considers only blocks 2 and 3.

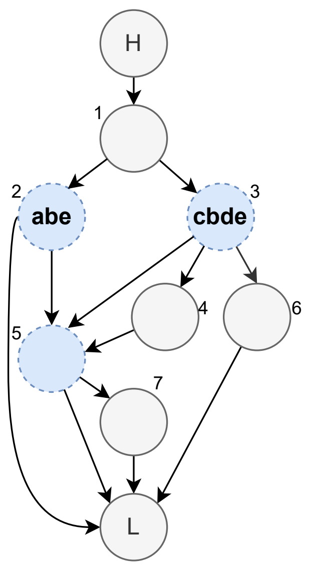
Why Topological Ordering in Algorithm 5.1?

Topological order is needed to make it possible to match the order of speculative requests made in the AGU with the order of values that will arrive from the CU on all its possible CFG paths. Consider the example of requests b and c in Figure 5.4a. We first want to hoist c to block 3 before hoisting b, because there exists a CFG path where c comes before b, but not vice versa. If b were hoisted before c, then the speculative requests order would be $b \prec c$, which would be impossible to match with values in the CU on the CFG path 3, 5, 7.

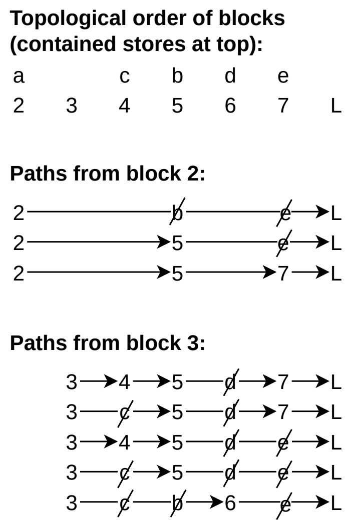

Legend:
Loss-of-decoupling control dependency source basic block
2 x
Basic block nr 2 with a memory operation x
x
Basic block poisoning store x



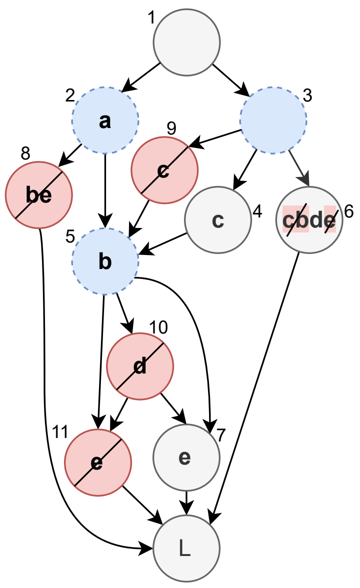
(a) Original CFG with memory operations: *a*, *b*, *c*, *d*, *e*.



(b) AGU CFG with speculative requests (algorithm 5.1).


Topological order of blocks (contained stores at top):
a c b d e
2 3 4 5 6 7 L
Paths from block 2:
2 → $\cancel{b}$ → $\cancel{e}$ → L
2 → 5 → $\cancel{e}$ → L
2 → 5 → 7 → L
Paths from block 3:
3 → 4 → 5 → $\cancel{d}$ → 7 → L
3 → $\cancel{c}$ → 5 → $\cancel{d}$ → 7 → L
3 → 4 → 5 → $\cancel{d}$ → $\cancel{e}$ → L
3 → $\cancel{c}$ → 5 → $\cancel{d}$ → $\cancel{e}$ → L
3 → $\cancel{c}$ → $\cancel{b}$ → 6 → $\cancel{e}$ → L

(c) Insertion of poison stores on CFG paths in the CU (algorithm 5.2).



(d) Final CU CFG with poison calls and blocks (algorithm 5.3). Block 6 sends two invalid values (poisoning stores c and b), then one valid value (for store d), then one invalid value (poisoning store e).

Figure 5.4: An example of introducing speculative memory requests in the AGU (section 5.4.1); and poisoned stores in the CU (section 5.4.2).

5.4.2 Poisoning Mis-speculated Stores on CFG Edges

Our strategy for killing misspeculations in the CU is to first map poison calls to CFG edges, and then to map poisoned CFG edges to poison store calls contained in basic blocks.

Algorithm 5.2 Mapping Poison Stores to CFG Edges in CU

```
Input: $SpecReqMap$ {basic block: list of requests hoisted to this block in Algorithm 5.1}

for $specBB, specRequests \in SpecReqMap$ do
    for $path \in allPathsToLoopLatch(specBB)$ do
        $trueBlocks \leftarrow \varnothing$                          ▷ set keeps insertion order
        for $r \in specRequests$ do
            $trueBB \leftarrow$ block where $r$ is true
            $trueBlocks.insert(trueBB)$
        for $edge \in path$ do
            for $trueBB \in trueBlocks$ do
                if $edge_{dst} = trueBB$ then
                    $trueBlocks.remove(trueBB)$
                    break                                       ▷ to the next edge
                if $trueBB$ not reachable from $edge_{dst}$ then    ▷ ignore loop backedges
                    poison $trueBB$ requests on $edge$               ▷ Algorithm 5.3
                    $trueBlocks.remove(trueBB)$
```

Algorithm 5.2 describes the first step. Given block $specBB$ that contains speculative memory requests $specRequests$, we consider each path in the DAG from the $specBB$ to the loop latch in the CU. We call the block where a $r \in specRequests$ becomes true the $trueBB$ (for example, the $trueBB$ for request b in Figure 5.4a is block 5). For each CFG path, we use the $trueBlocks$ list to keep track of which requests were already used or poisoned on the path—the list contains the $trueBB$ for each $r \in specRequests$.

Given an edge in the traversal, the edge is skipped if the next $trueBB \in trueBlocks$ is still reachable from $edge_{dst}$. This guarantees that the order of speculative requests in the AGU matches the order of values in the CU, i.e., a speculative request for a given $trueBB$ block is not poisoned immediately when $trueBB$ becomes unreachable if there is an earlier speculative request that can still be used.

Example of Mapping Poison Stores to CFG Edges:

Figure 5.4c shows which CFG edges are poisoned given the original CFG in Figure 5.4a and the AGU CFG in Figure 5.4b. For example, the path $3 \to 5 \to L$ will have: $poison(c)$ on the $3 \to 5$ edge; and $poison(d)$, $poison(e)$ on the $5 \to L$ edge (4th path from BB 2 in Figure 5.4c).

5.4.3 Mapping Poisoned CFG Edges to Basic Blocks

Algorithm 5.3 shows how poisoned CFG edges are mapped to actual poison calls placed in a concrete basic block. Given a poisoned request r on $edge$, there are three cases:

1. There exists a path from $trueBB$ to $edge_{dst}$. In this case, we cannot insert $poison(r)$ in $edge_{dst}$, because we would end up with a CFG path where the store is both true and poisoned. To avoid this, we create a new $poisonBB$ block on $edge$ and append $poison(r)$ it.

Algorithm 5.3 Poisoning Stores on Edges in CU

```
 1: Input: store request $r$; CFG $edge$; block $specBB$ where $r$ was speculated; block $trueBB$
    where $r$ is true
 2:
 3: $poisonBlockReuse \leftarrow \varnothing$                    ▷ preserve set across Algorithm calls
 4: if $edge_{dst}$ is reachable from $trueBB$ then
 5:     $poisonBB \leftarrow$ create new block on $edge$ or
 6:                     get from $poisonBlockReuse$ if exists
 7:     append $poison(r)$ to the end of $poisonBB$
 8:     $poisonBlockReuse.insert(poisonBB)$
 9: else if $specBB$ does not dominate $edge_{dst}$ then
10:     $poisonBB \leftarrow$ create new block on $edge$
11:     append $poison(r)$ to the end of $poisonBB$
12:                     ▷ create $\phi$ recursively on $specBB \rightarrow edge_{src}$ paths
13:     create $\phi(1, specBB)$ value in $edge_{src}$
14:     branch from $edge_{src}$ to $poisonBB$ on $\phi = 1$
15: else
16:     append $poison(r)$ to the start of $edge_{dst}$
```

2. There exists a path from the loop header to $edge_{dst}$ that does not contain $specBB$. In this case, we cannot insert $poison(r)$ in $edge_{dst}$, because we would end up with a CFG path where r was not speculated in the AGU, but was poisoned in the CU. To avoid this, we create a new block $poisonBB$ on the edge and append $poison(r)$ to it. We also add steering instructions to the path from $specBB$ to $poisonBB$ that will branch from $edge_{src}$ to $poisonBB$ only if $specBB$ was encountered on the current CFG path.
3. Otherwise, $poison(r)$ can safely be prepended to the start of $edge_{dst}$.

Algorithm 5.3 is executed only once per $(edge,\ r)$ tuple—a given request is poisoned at most once on a given edge. Also, poison blocks created in case 1 in Algorithm 5.3 can be reused to poison other requests.

Example of Mapping Poison Edges to Basic Blocks

Consider how the poisoned edges in Figure 5.4c are mapped to basic blocks in Figure 5.4d. This example exercises all three cases in Algorithm 5.3.

Case 1

Store c is poisoned on the $3 \rightarrow 5$ edge. Since there is a path from the true block of c (block 4) to the edge destination block (block 5), we create a new block on the $3 \rightarrow 5$ edge and append $poison(c)$ to it.

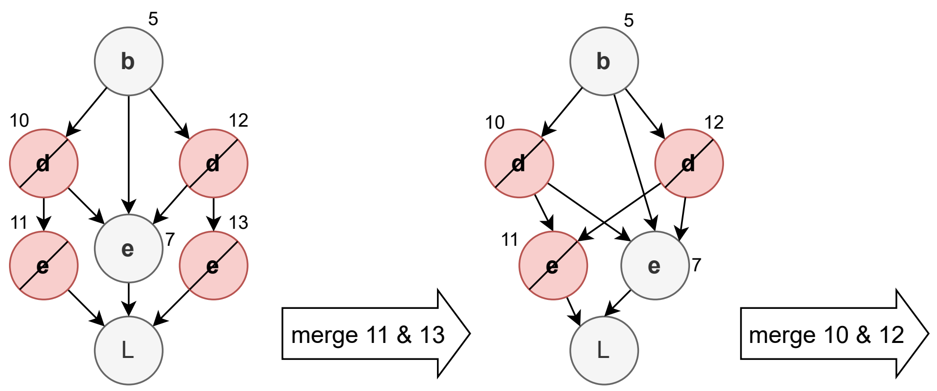


Figure 5.5: Basic blocks with the same list of poison stores and the same immediate successor can be merged in the CU.

Case 2

Store d is poisoned on both the $5 \to 7$ and $5 \to L$ edges. The $specBB$ for d is block 3. Since there exists the path $H \to 1 \to 2 \to 5$ that does not contain block 3, we create a new block on the $5 \to 7$ edge with the $poison(d)$ call. We add steering instructions to the $3 \to 5$ and $3 \to 4 \to 5$ paths that will cause block 5 to branch to the new poison block on the $5 \to 7$ edge only if block 5 was reached from a path containing block 3.

Case 3

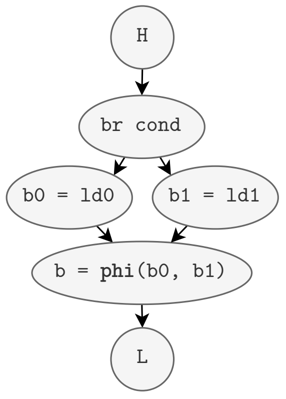
Store c is also poisoned on the $3 \to 6$ edge, but here it is safe to prepend $poison(c)$ to the start of block 6.

5.4.4 Merging Poison Blocks

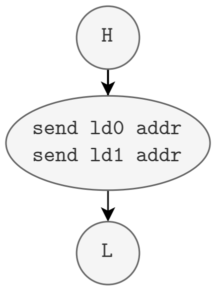
Case 1 and 2 of Algorithm 5.3 might create multiple poison blocks for the same store on different CFG edges. It is possible to merge two poison blocks into one if they contain the same list of poison stores and if they have the same list of immediate successors (when merging, we keep instructions from just one block). We apply such merging iteratively after algorithms 5.2 and 5.3. For example, Figure 5.5 contains a CFG sub-region of our running example from Figure 5.4. Algorithm 5.3 inserted poison blocks $10, 11, 12, 13$ to poison stores d and e. Block pairs $(11, 13)$ and $(10, 12)$ can be merged, because they contain equivalent poison calls and the same list of immediate successors.

5.4.5 Consumption of Speculative Loads

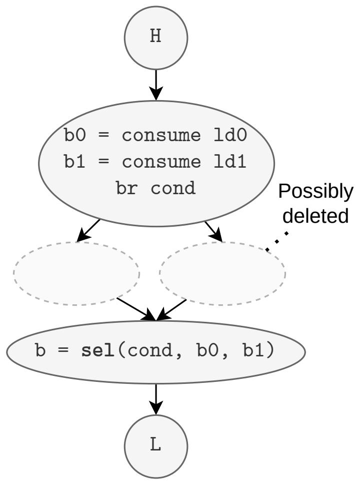
To match the order of `load_consume` calls made in the CU with the order of speculative `send_load_addr` calls in the AGU we can hoist the `load_consume` calls to the same block where the corresponding `send_load_addr` were hoisted in the AGU. Then, the CU can either use the load value or discard it. After hoisting, we need to update all ϕ instructions that use



(a) Original CFG.



(b) After DAE transformation. AGU CFG with hoisted loads.



(c) After DAE transformation. CU CFG with a select instruction instead of a ϕ-node.

Figure 5.6: The hoisting of load consume calls in the CU requires changing ϕ-nodes to select instructions.

the load value, since the basic block containing the loaded value will have changed. Alternatively, we can transform ϕ instructions using the load value into `select` instructions, as shown in Figure 5.6, provided that the branch conditions needed to build the `select` predicate dominate the basic block where the `select` will be placed.

Similarly to the effect of store speculation, the hoisting of instructions needed to implement the consumption of speculative loads might result in empty basic blocks. These can be removed using standard CFG simplification passes.

5.5 Safety and Liveness

In this section, we prove that our transformations preserve the sequential consistency of the original program and that they do not introduce deadlock. In Section 4.7 in the previous chapter, we proved that our DAE transformation without speculative memory requests does not result in deadlock. We now extend this proof to also consider speculative memory requests introduced in this chapter. Since our speculation transformation causes some stores to not be committed to memory, we consider sequential consistency in this section. Deadlock freedom is a corollary of sequential consistency, so we focus only on the latter. As shown in Chapter 4, deadlock can only occur when there is a mismatch between the number of generated memory requests in the AGU, and the number or order of loads and stores in the CU. We show that on every CFG path the order of speculative store requests in the AGU matches the order of store values in the CU, and that the non-poisoned store value sequence in the CU matches the store sequence of the original code. Thus, from the preservation of sequential consistency follows the fact that the number of memory requests made from the AGU matches the number of loads and stores in the CU.

In the following discussion, we assume blocks with a single store; the proof trivially extends to blocks with multiple stores since all speculative stores in the same block are treated the same. We also assume that all stores are speculative, since the relative order between non-speculative and speculative stores is guaranteed by definition. Given a a non-speculative store s_1 and a speculative store s_2, our Algorithm 5.1 will not change the relative program order of s_1 and s_2, i.e., if $s_1 \prec s_2$ in the original program order, then it is not possible to hoist s_2 such that $s_2 \prec s_1$. This follows from the control dependency definition (Section 5.3)—s_2 hoisting stops at its LoD control dependency source $srcBB$, which must come after the block containing s_1 in topological order. If $srcBB$ came after s_1 in topological order, then the block containing s_1 would also have a LoD control dependency on $srcBB$ and would have been hoisted. Since s_1 was assumed to be non-speculative, this would create a contradiction. A similar argument can be made if $s_2 \prec s_1$ in the original program order.

Intuition for Proof

Algorithm 5.2 goes over every possible path through the CU CFG. On any given such path traversal, it will insert at most one store poison call per store request. A request whose true block is reached in a given path will not be poisoned on the same path.

Theorem 5.1 (Sequential Consistency Preservation of DAE Speculation)**.** *Given an ordered list of n speculative store requests $L_a = \{a_0, a_1, ..., a_{n-1}\}$ made in the AGU loop CFG on some fixed iteration k, Algorithms 5.2 and 5.3 transform the CU CFG such that every possible path through its loop CFG on iteration k produces an ordered list of n tagged store values $L_v = \{(v_0, p_0), (v_1, p_1), ..., (v_{n-1}, p_{n-1})\}$, such that each $(a_i, v_i, p_i), 0 \le i < n$ triple corresponds to a $A[a_i] \leftarrow v_i$ store in the original program CFG, and $p_i = 1$ (poison bit) if that store is not executed on the path through the original loop CFG on iteration k.*

Proof. We use a proof by induction on the transformed CFG.

Base Case $L_a = \varnothing$ (no speculated requests in the AGU). Algorithm 5.2 does not change the CU CFG. Thus, the order of store addresses in the AGU and store values in the CU trivially matches, $L_a = L_v = \varnothing$.

Inductive Hypothesis Assume Lemma 5.1 holds at basic block B_i in the current CFG path. All store requests $a_j \in L_a$ contained in blocks reached before B_i in the path were matched with the correct store value call $(v_j, p_j) \in L_v$, such that $p_j = 1$ if $A[a_j] \leftarrow v_j$ was not executed on the path in the original loop CFG.

Inductive Step The next store address in the AGU L_a sequence is $a_{j+1} \in L_a$. The next store value in the CU CFG path should be $(v_{j+1}, p_{j+1}) \in L_v$, where $p_{j+1} = 1$ iff the store $A[a_{j+1}] \leftarrow v_{j+1}$ is not reached on the current CFG path in the original program. Algorithm 5.2 considers the $edge_{src} \rightarrow edge_{dst}$ next. There are three cases:

1. $edge_{dst} = trueBB$, where $trueBB$ is the block containing the store $A[a_{j+1}] \leftarrow v_{j+1}$ in the original program CFG. In this case, Algorithm 5.2 will not poison this store on this path through the CU CFG, i.e. the next item in the L_v sequence will be the correct $(v_j, 0)$.

2. $edge_{dst} \neq trueBB$ and $trueBB$ is not reachable from $edge_{dst}$, in which case Algorithm 5.2 will insert a poison store on this edge. Algorithm 5.3 will map this poison store to a basic block, with the effect that taking the $edge$ will result in the poison call being executed and control transferring to $edge_{dst}$. The next item in the L_v sequence will be the correct $(v_j, 1)$.
3. $edge_{dst} \neq trueBB$ and $trueBB$ is reachable from B_l, in which case Algorithm 5.2 will traverse the path until Case 1 or 2 is matched.

Since theorem 5.1 holds for the base case, for basic blocks on the path up to B_i, and for some successor block of B_i, it must hold at any block on the path. If it holds at any block on the path, it holds for the whole path. Since a given store request r is poisoned at most once on a given CFG edge and since, by definition of Algorithm 5.2, any given path will contain at most one edge where r is poisoned, we conclude that Lemma 5.1 holds for all paths. □

5.6 Applications

In this section, we highlight two applications, other than HLS-generated accelerators, that can benefit from our speculation implementation: DAE-based prefetchers in CPUs/GPUs, and memory systems in CGRAs. In the next section, we use HLS as an evaluation vehicle to study the effectiveness of our speculation transformation. However, we emphasize that our speculation support in DAE does not rely on any HLS-specific features and can be applied wherever speculation is combined with the DAE technique.

5.6.1 CPU/GPU Prefetchers

Most existing works on CPU/GPU prefetchers follow the DAE principle and rely on the compiler to decoupled address generation from compute [60, 167, 104, 9, 6, 59, 190]. All of these works suffer from the control-dependency LoD problem (Section 5.3). The work in [104] discusses adding speculation and predicated stores to the CPU microarchitecture to mitigate LoD, but their compiler only supports simple diamond and triangle control flow shapes. In this paper, we have demonstrated generalized compiler support for speculation in DAE, making these works viable for general control flow and thus applicable to a broader set of codes.

Concrete Example

The CPU prefetcher proposed in [104] (on which most of the other cited work is based) separates address generation from compute and extends the ISA with `store_addr`, `load_produce` `store_val`, `load_consume`, and `store_inv` instructions that can be directly targeted by our compiler.

5.6.2 Coarse Grain Reconfigurable Architectures

A CGRA consists of an array of Processing Elements (PEs), each with small memories, connected by a network. A CGRA compiler is typically co-designed with the hardware, as the PEs are typically statically scheduled. The job of the compiler is to map the Control/Data Flow Graph (CDFG) to the PEs, and many works follow the DAE technique to tackle the memory wall problem [166, 188, 187, 83, 111, 180, 233, 171]. Our work can help mitigate LoD events when mapping to CGRAs.

Concrete Example

The CGRA proposed in [171] is a popular example of modern streaming dataflow CGRA. All communication in the CGRA is FIFO-based, and address generation is explicitly decoupled at compile time into specialized AGUs. The CGRA compiler generates commands to produce address streams and to consume/produce values. Control flow is handled with predication and there is a `SD_Clean_Port` command to throw away a value from an output port that can implement predicated stores. Our speculation approach can be directly used in this work.

5.6.3 High-Level Synthesis

In HLS, the CDFG of an algorithm is implemented directly in hardware following a spatial execution model. Such an execution model and the freedom to customize the memory system make decoupling easier in HLS compared to the temporal CPU/GPU execution model. Thus, HLS-generated accelerators can directly benefit from our work today without any changes, and it is in this domain that we evaluate our implementation in the next section.

As explained in the introduction chapter, although existing HLS compilers are successful in building non-trivial accelerators for regular code (e.g., [192]), their static scheduling techniques are sub-optimal for irregular codes. Many research works in academia and industry have exploited DAE in HLS to improve the efficiency of HLS-generated accelerators for irregular codes [42, 41, 51, 54, 44, 89, 212, 213, 216]. By adding compiler speculation support, DAE in HLS can be used on a broader set of codes, which we demonstrate in the next section.

Dynamic Loop Fusion

Another useful application of speculation in a DAE architecture is dynamic loop fusion in HLS, which will be demonstrated in the next chapter. In that application, it is important that for every loop iteration all memory requests are send to a specialized DU, even if they will ultimately not be executed due to control flow. Speculation, as presented in this chapter, is the most efficient way to achieve this requirement. This application is described in more detail in Section 6.6 in the next chapter.

5.7 Evaluation

In the evaluation section from the previous chapter, we have compared our LSQ with speculation support to the state-of-the-art Dynamatic LSQ [125]. Altough we have shown that our speculation support enables us to achieve a much better speedup on codes with LoD problems than previous work, we have focused on evaluating the entire LSQ, not just the impact of speculation. In this section, we study the impact of speculation in more detail and we answer the following questions:

- What is the performance benefit of using a DAE architecture (enabled by our speculation approach) to accelerate codes with LoD control dependencies?
- What is the cost of mis-speculation in our approach?
- What is the impact on code size (accelerator area usage) of our speculation approach?
- What is the scalability for nested control flow, which increases the number of poison stores and basic blocks?

We make the implementation work and evaluation of this chapter publicly available [209].

5.7.1 Methodology

We generate algorithm-specific accelerators using HLS targeting an Intel Arria 10 FPGA. The C input codes are taken directly from benchmark suites without adding any HLS-specific annotations (excluding dynamic structures, e.g., queues, where these were replaced with HLS-specific libraries).

We use the LLVM-based Intel SYCL HLS compiler version 2023.1.0 [118] applying the standard DAE transformation (Chapter 3) and our proposed speculation transformation (Section 5.4) as LLVM passes. The codes use deterministic dual-ported on-chip SRAM capable of 1 read and 1 write per cycle. To enable Out-of-Order (OoO) loads, we use our LSQ with speculation support from Chapter 4.

We report cycle counts from ModelSim simulations. We do not report circuit frequency since our all four approaches have a similar critical path ($\pm 10\%$). Accelerator area usage is obtained after place and route using Quartus 19.2.

Baselines

For each benchmark, we synthesize the following architectures which represent current state-of-the-art approaches to HLS :

- `STA`: the default, industry-grade approach using static scheduling [118]. Loads that cannot be disambiguated at compile time execute in order.
- `DAE`: a DAE architecture without speculation. OoO loads are enabled by an LSQ. This is our approach to the HLS of irregular codes described in Chapter 3 and the current state-of-the-art in academia [213]. This approach suffers from control-dependency LoDs.

- `SPEC`: the same as `DAE`, but with our speculation technique which mitigates control-dependency LoDs.
- `ORACLE`: the same as `DAE`, but all LoD control dependencies are removed manually from the input code. This means that there are no control dependencies causing a LoD anymore and the AGU can be decoupled, but the final result in memory will likely be wrong, because some stores not meant to execute in the original code are executed by the `ORACLE`. The `ORACLE` results are used as an upper a bound on the performance of `SPEC` and show its area overhead.

Benchmarks

DAE architectures optimize the latency between memory and compute and are most beneficial for memory-bound codes [104], especially codes with an irregular memory access pattern that prevents static prefetching [60]. We evaluate nine such benchmarks from the graph and data analytics domain, using the GAP graph benchmark suite [21] and an HLS benchmark suite [45] of irregular programs. We select only codes that can benefit from our `SPEC` approach, i.e., codes with LoD control dependencies:

- `bfs`: breadth-first traversal through a graph.
- `bc`: betweenness centrality of a single node in a graph.
- `sssp`: single shortest path from a single node to all other nodes in a graph using Dijkstra's algorithm.
- `hist`: histogram, similar to Figure 5.1b (size 1000).
- `thr`: zeroes RGB pixels above threshold (size 1000).
- `mm`: maximal matching in a bipartite graph (2000 edges).
- `fw`: Floyd-Warshall distance calculation of all node-to-node pairs in a dense graph (10×10 distance matrix).
- `sort`: using bitonic mergesort (size 64).
- `spvm`: sparse vector matrix multiply that skips zero columns (20×20 matrix).

For the graph codes (`bfs`, `bc`, `sssp`) we use a real-world graph `email-Eu-core` with 1005 nodes and 25,571 edges. The small input sizes are due to long simulation times; the results scale to larger datasets executed in hardware.

5.7.2 Performance Results

Figure 5.7 reports normalized speedups of each technique over `STA`. Our `SPEC` approach gives on average a $1.9\times$ (and up to $3\times$) speedup over `STA`. This is within 5% of the `ORACLE` performance. In contrast, `DAE` without speculation sees a dramatic performance degradation over `STA`, because the AGU, DU, CU communication is sequentialized.

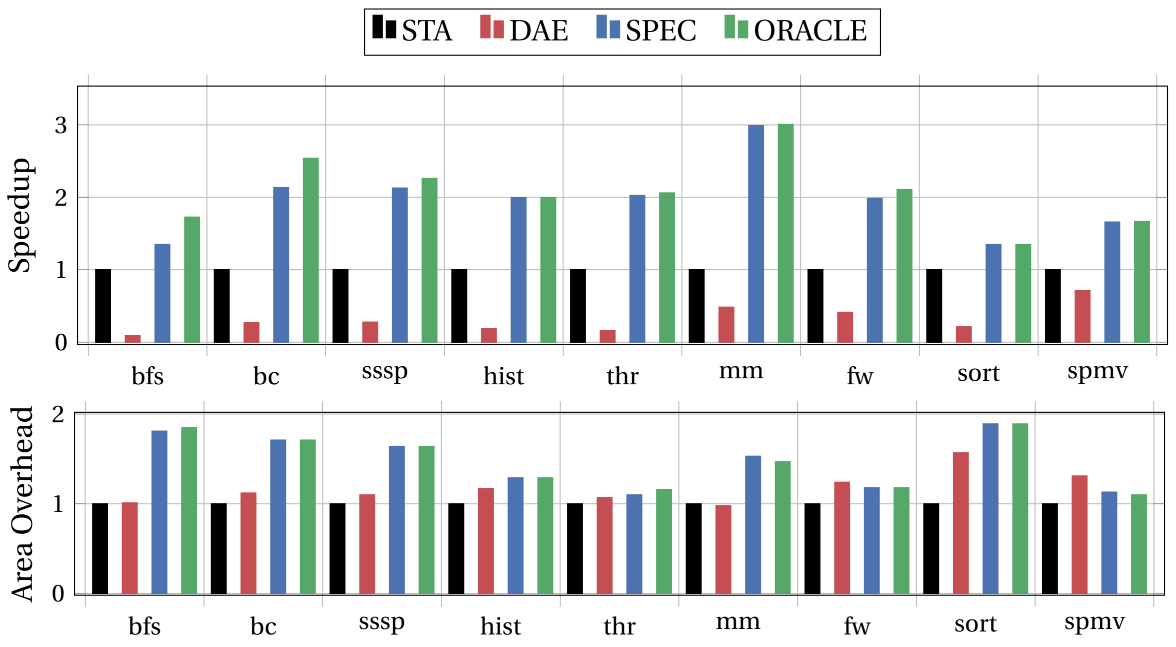


Figure 5.7: Speedup and area overhead of DAE, SPEC and ORACLE normalized to STA. SPEC achieves an average 1.9× (up to 3×) speedup over STA at a modest average area overhead of 1.36×. Using a DAE architecture without speculation results in a substantial slowdown (the DAE results).

Table 5.1: Absolute performance and area usage of the four evaluated approaches. (*bc uses two LSQs).

Kernel	**Poison**		**Branches**	**Cycles (1000s)**				**Area (1000s of ALMs [115])**			
	BBs	Calls	**executed**	STA	DAE	SPEC	ORACLE	STA	DAE	SPEC	ORACLE
bfs	1	1	5%	37.2	399	27.6	21.6	7.4	7.5	13.4	13.7
bc	2	2	5%, 18% *	109	406	51.1	42.9	9.7	10.9	16.6	16.6
sssp	1	1	5%	109	391	51.2	48.2	10.6	11.7	17.4	17.4
hist	1	1	98%	2.1	11.1	1	1	2.4	2.8	3.1	3.1
thr	1	3	3%	2.1	13.1	1.1	1	5.7	6.1	6.3	6.6
mm	1	2	69%	12.2	25.1	4.1	4	5.1	5	7.8	7.5
fw	1	1	15%	6.8	16.5	3.3	3.2	3.4	4.2	4	4
sort	1	2	51%	2.4	11.1	1.7	1.7	2.8	4.4	5.3	5.3
spmv	1	1	68%	13.3	18.7	8	8	3.9	5.1	4.4	4.3
			Harmonic Mean:	1	3.2	0.51	0.48	1	1.16	1.42	1.36

Branch Taken Rate

The performance gap between SPEC and ORACLE is highest on the bfs and bc codes, because of its deep pipeline between the load and store that form a RAW hazard. The deep pipeline means that more store allocations need to be held by the LSQ [151] to guarantee perfect pipelining. This, together with a low branches executed rate in these benchmarks (Table 5.1), can cause the LSQ to fill with store addresses that are mis-speculated, potentially stalling later loads that have to wait for future store addresses to arrive. This problem can be solved by increasing the store queue size in the LSQ. The increased number of requests and the need for more buffering is one of the limitations of our approach. Codes with a shallower pipeline that do not need large LSQ sizes have no mis-speculation penalty.

To prove this, we choose three benchmarks where we can instrument the input data so that we can vary the mis-speculation rate. Table 5.2 shows how the mis-speculation cost changes as a function of the branch taken rate. As can be seen, there is no correlation between the branch taken rate and the mis-peculation cost, with the slight variability in clock cycle counts attributable to the subtle difference in the number of true RAW hazards due to the varying data distribution.

Table 5.2: `SPEC` cycle counts and standard deviation as rate of branches executed changes.

Kernel	**Mis-speculation rate**						σ
	100%	80%	60%	40%	20%	0%	
hist	1044	1013	1029	1029	1012	1051	16
thr	1082	1109	1047	1073	1058	1071	21
mm	4107	4096	4074	4063	4106	4081	18

To summarize, in geneal mis-speculations are not expensive in our approach, because there is no additional bookkeeping needed to be done on mis-speculation, as opposed to the high cost of mis-speculations in an OoO CPU. The sole purpose of our speculation is to allow for the runahead of address generation, and discarding mis-speculated addresses in the DU is trivial compared to flushing the pipeline of an OoO CPU.

5.7.3 Area Usage

Our speculation approach can increase the number of blocks in the CU, especially for codes with deeply nested control flow. An increased number of blocks can result in a higher area usage due to increased scheduler complexity [196].

Table 5.1 shows the absolute area usage of all accelerators. We observe virtually no area overhead of `SPEC` over `ORACLE` on the evaluated benchmarks. This is because most of the codes have at most two control-flow nesting levels where new poison blocks are inserted, and sometimes it is possible to reduce the number of blocks using our merging technique (e.g. two poison blocks in `mm` merged into one).

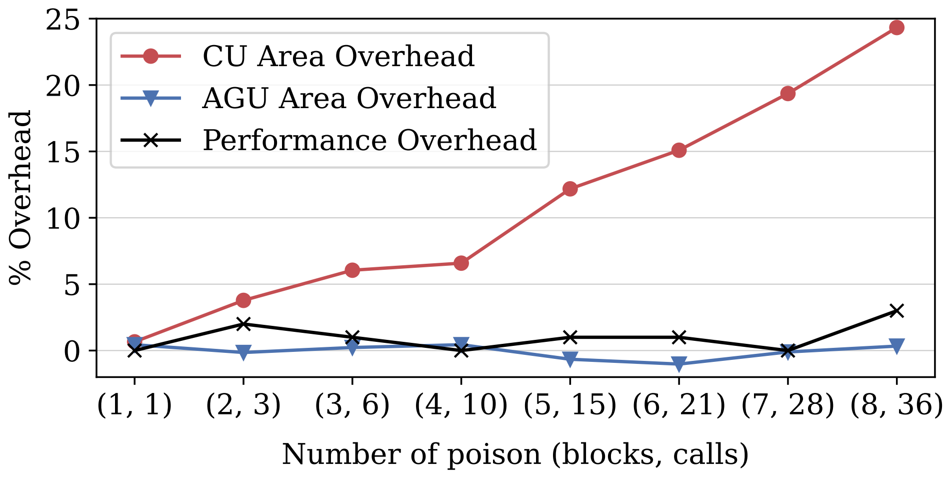


Figure 5.8: Change in area and performance overhead of SPEC over ORACLE as the number of poison blocks and calls grows.

Impact of Nested Control Flow on Area

To give a more meaningful measure of how nested control flow impacts the area overhead of our SPEC approach, we create a synthetic benchmark template where we can tune the number of poison blocks generated by SPEC:

```
if $x > 0$ then
    $store_1$
    if $x > 1$ then
        $store_2$
        if $x > 2$ then
            ...
```

Each nesting level in this template will result in one poison block in the SPEC architecture. With n stores and assuming one store per nesting level, there will be n poison blocks and $\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$ poison calls.

Figure 5.8 shows how the area and performance overhead of SPEC over ORACLE changes as more poison blocks are needed. The performance overhead is close to 0% and does not change with more poison blocks. The area overhead of the AGU unit is similarly close to 0%, because SPEC hoists stores out of the *if*-conditions, causing the blocks to be deleted. The area overhead of the CU unit grows by a few percent (< 5%) with each added poison block, but even for the pathological case of eight nested *if*-conditions the overhead is below 25%. In real codes, with more compute and lower control-flow nesting, the area overhead of SPEC should be minimal (within a few percent, as demonstrated in Table 5.1).

5.8 Related Work

Program Slicing

Program slicing is used beyond DAE architectures. Decoupled Software Pipelining (DSWP) [175] is a popular compiler transformation that decouples strongly connected components in the program dependence graph into separate pipeline stages mapped over multiple PEs communicating via FIFOs. The PEs can be CPU threads, or pipeline stages in an accelerator generated by HLS [150]. Control dependent pipeline stages in DSWP can also be executed speculatively, although stages with memory operations require versioned memory [225]. Other decoupling policies than the original DSWP formulation are also possible [166, 167, 44], as is time-multiplexing multiple pipeline stages over the same PE [179, 230]. Our work is different in that we decouple only address generating instructions and speculate only on memory operations,

Control Speculation

Control speculation has its roots in compilers for Very Large Instruction Word (VLIW) machines. Instruction scheduling in HLS is very similar to VLIW scheduling (no hardware support for speculation, static mapping to functional units, etc.), with many algorithms like modulo-scheduling and *if*-conversion originally developed for VLIW directly applicable to HLS [196, 3, 178]. Most recently, predicated execution in the form of gated SSA was proposed for HLS with speculation support [97]. The speculation support in this and other works requires costly recovery on mis-speculation [127, 220, 14, 154, 93, 231]. Efficiently squashing speculative computation on the wrong paths in a spatial dataflow architecture is hard, because the architectural state is distributed [32]. Our speculative DAE sidesteps this issue, not requiring any recovery: we speculate early (run ahead) in the AGU, and later handle mis-speculations in the CU by taking an appropriate path in its CFG.

Control Flow Handling in GPUs

Control-flow handling in GPUs is most commonly implemented via predication. The algorithms used to calculate predicate masks and re-convergence points bear a striking resemblance to our work [148]. The Single Instruction Multiple Threads (SIMT) stack approach in GPUs pushes predicate masks onto the stack when entering a control-flow nesting level, and performs a pop when exiting. Our Algorithm 5.1 implementing speculative requests can be seen as a pass through the CFG with only push operations, where the push is onto individual stacks of control-dependency sources. Dually, our Algorithm 5.1 inserting poison calls can be seen as a pass through the CFG with only pop operations where the placement of the pops follows a certain policy instead of popping at the immediate post-dominator as in the traditional SIMT formulation. Modern SIMT implementations often employ CFG path analysis to optimize the placement of pops to prevent SIMT deadlock/livelock or to improve performance [78].

5.9 Conclusion

This chapter demonstrated our general compiler support for speculative memory operations in DAE architectures that tackles the LoD problem resulting from control dependencies. We have proposed CFG transformations implementing speculation in the AGU, and poisoning of mis-speculations in the CU, with a proof of correctness.

We have presented three applications where our work improves support for the efficient execution of irregular codes: DAE-based CPU/GPU prefetchers that require compiler support, CGRA architectures, and HLS-generated specialized accelerators. We have evaluated our work on HLS-generated accelerators, showing an average 1.9× (up to 3×) speedup over non-DAE accelerators on a set of irregular benchmarks where DAE is not possible without our speculation. Our approach has no mis-speculation cost and a small area overhead, scaling well to deeply nested control flow.

Future work could extend our speculation approach with support for vector-parallelism by filling a vector of speculative requests in the AGU and producing a store mask in the CU, similar to the recent work on decoupled vector runahead prefetching in CPUs [163].

Chapter 6

Dynamic Loop Fusion

Dynamic Dataflow (DDF) HLS uses additional hardware to perform memory disambiguation at runtime, increasing loop throughput in irregular codes compared to Finite State Machine with Datapath (FSMD) HLS. However, most irregular codes consist of multiple sibling loops[1], which currently have to be executed sequentially by all HLS tools. FSMD HLS can perform loop fusion only on regular codes, while DDF HLS relies on loops with dependencies to run to completion before the next loop starts.

In this chapter, we present dynamic loop fusion for HLS, a compiler-hardware co-design approach that enables multiple loops to run in parallel, even if they contain unpredictable memory dependencies. Our only requirement is that memory addresses are monotonically non-decreasing in inner loops. The choice of restricting address expressions to monotonically non-decreasing functions allows us to perform memory disambiguation across loops, which is not possible with existing methods based on Load-Store Queues (LSQs), which make no assumptions about the underlying memory address distribution. As we show in Section 6.3, this monotonicity requirement is not too restrictive, with many codes falling into this class.

We also present a novel program-order schedule for HLS, inspired by polyhedral compilers, that together with our address monotonicity analysis enables dynamic memory disambiguation that does not require searching of address histories and sequential loop execution. Our evaluation shows an average speedup of 14× over FSMD HLS and 4× over DDF HLS.

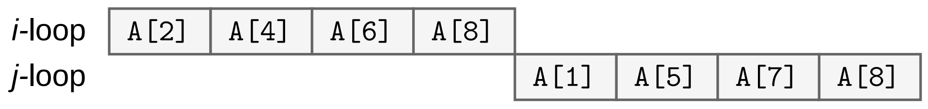
6.1 Introduction

HLS increases designer productivity, makes code more maintainable, accelerates verification, and makes design space exploration easier [193]. However, this is usually only true for regular codes where the compiler can discover instruction- and memory-level parallelism statically [196, 37]. Domains like graph analytics and sparse linear algebra contain irregu-

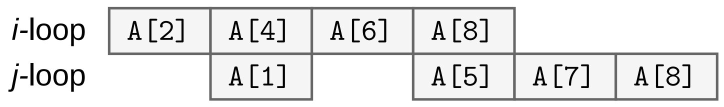
1. Loops with the same loop depth and the same loop parent.

```
for (i = 0; i < N; ++i)
    A[f(i)] = workA(A[f(i)]);
for (j = 0; j < M; ++j)
    B[g(j)] = workB(A[g(j)]);
```

(a) Two sibling loops with non-affine access patterns.



(b) Pipeline achieved by current static and dynamic HLS tools. The j-loop needs to wait for the i-loop to finish.



(c) Pipeline achieved by our work. The $A[1]$ access in the j-loop is deemed safe because the most recently accessed address in the i-loop is 2 and we know that the i-loop address expression is monotonically non-decreasing—we know that the i-loop will not access address 1 after it has accessed address 2.

Figure 6.1: Dynamic Loop Fusion enables fine-grained parallelism *across* loops with memory dependencies.

lar codes with unpredictable memory dependencies and control flow, which break the traditional static scheduling approach. This prompted research into DDF HLS [126] and approaches to combine it with existing industry-grade FSMD HLS compilers, which we have described in the previous chapters [214].

As described in Chapter 4, DDF HLS uses Load-Store Queues (LSQs) to perform dynamic memory disambiguation at runtime [112, 125, 212, 97, 61, 62]. These works effectively pipeline single loops with arbitrary memory dependencies, but they have to sequentialize multiple loops if they share a memory dependency. For example, they would sequentialize the *i*- and *j*-loops in Figure 6.1a, resulting in the Figure 6.1b pipeline. But, as shown in Figure 6.1c, there might be plenty of parallelism *across* the two loops at runtime when the address values become concrete.

There are two reasons why current DDF HLS tools have to sequentialize these loops. Firstly, they use a program-order schedule that relies on loops to run to completion before the next loop starts. For example, the LSQ used in Dynamatic HLS sequentializes LSQ requests based on the program order of basic blocks [125]; other approaches carry explicit dependencies through the pipeline, preventing downstream loops from starting without resolving the dependency [77, 212]. Secondly, they rely on the checking of address histories to detect hazards, without making any assumptions about the underlying address distributions. This makes them general, but requires them to wait for all addresses from one loop to be produced before they can start processing the next loop. These are two *key challenges* that we tackle in this paper.

Static loop fusion (called kernel, operator, layer, or task fusion in other domains) also fails to fuse the loops in our Figure 6.1a example, because the fused loop may introduce a negative dependency distance [132]—the compiler gives up if it cannot prove that $f(i) = g(j) \implies i < j$. This is assuming that the $f(i)$ and $g(i)$ functions can be analyzed by the compiler in the first place. If that is not the case, e.g., if they involve an array access, then loop fusion is also not applied. Optimizing compilers can apply preparatory transformations like loop peeling, interchange, or shifting to increase the chances of loop fusion being legal [244], however, these preparatory transformations do not integrate well with the instruction scheduling algorithms used in HLS [196, 37]. They often introduce additional control flow, loop exits, and new dependencies, which can result in a worse schedule produced by the HLS tool [246]. Moreover, if there are more than two loops, deciding the best subset of loops to fuse becomes NP-complete [63], and this is even before deciding if preparatory transformations should be used.

Our dynamic loop fusion approach can automatically synthesize a Read After Write (RAW) check that will protect the $A[g(j)]$ read in the Figure 6.1 code, achieving the fine-grained inter-loop parallelism from Figure 6.1c. We decouple each loop into an independently scheduled Processing Element (PE). Memory dependencies across loops are handled in a Data Unit (DU) specialized by our compiler for the program. Our only requirement is that the `f(i)` and `g(j)` functions are monotonically non-decreasing in the innermost loop (outer loops can be non-monotonic). This is a weaker requirement than the affine functions expected by static loop fusion, allowing us to fuse more loops, including codes with data-dependent addresses.

To the best of our knowledge, we are the first to propose dynamic memory disambiguation that can work *across* loops. We make the following contributions in this chapter:

- A compiler analysis, based on the chain of recurrences theory [16, 80], that checks if addresses are monotonically non-decreasing in inner loops, and that detects non-monotonic outer loops (Section 6.3).
- A hardware-efficient program-order schedule representation that does not require sequentializing loops. We show how the compiler instruments Address Generation Units (AGUs) with instructions that generate the schedule for each memory operation. We also show how outer loop that are not monotonic can be integrated with our schedule (Section 6.4).
- A parameterizable DU performing dynamic memory disambiguation across loops. We show how the compiler can specialize the DU given the dependency graph of the program and the address monotonicity analysis. We discuss how the DU optimizes DRAM bandwidth by using dynamic coalescing and on-chip store-to-load forwarding (Section 6.5).

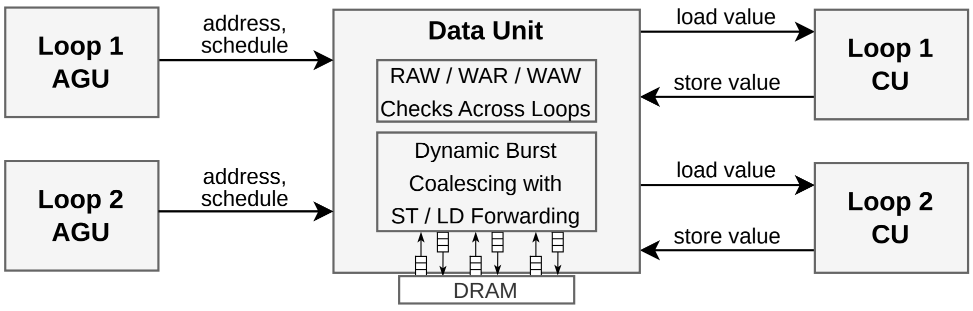


Figure 6.2: An example DAE streaming FPGA architecture.

- An evaluation on irregular applications showing an average speedup of 14× over FSMD HLS and 4× over DDF HLS. We discuss which codes benefit from dynamic fusion and we study the impact of store-to-load forwarding (Section 6.7).

6.2 Background

In this paper, we focus on codes using DRAM, as its unpredictable latency and limited bandwidth pose greater challenges than BRAM. There is no fundamental reason why we could not protect BRAM or use a memory hierarchy with BRAM caches, which we briefly discuss in Section 6.8.

In this section, we describe FPGA streaming architectures commonly used with DRAM. We discuss techniques to optimize DRAM bandwidth in irregular codes that inform the design of our DU. And we describe existing loop fusion approaches and their compiler theory, informing the design of our program-order schedule representation.

6.2.1 Baseline Streaming Architecture

Streaming FPGA architectures are a popular choice for implementing DRAM-based codes [53, 221, 199, 85, 226]. They decouple memory accesses and compute into separate PEs, either automatically [85, 226] or manually [53, 221, 199]. The use of a streaming architecture is predicated on an accurate memory dependency analysis so that memory shared between PEs can be transformed into FIFO communication. If the analysis fails, as it invariably does for irregular codes, then the shared data has to be communicated via DRAM and the execution of PEs has to be sequentialized, thus losing much of the benefits of using a streaming FPGA architecture.

To tackle the problem of irregular memory accesses, we propose to use a DU parameterized by our compiler, shown in Figure 6.2, that protects memory shared across loops by performing dynamic memory disambiguation at runtime. The DU interfaces with DRAM, but is also able to directly forward values from producer to consumer PEs if the respective load/store operations exhibit temporal locality, thus saving DRAM bandwidth as in traditional streaming FPGA architectures.

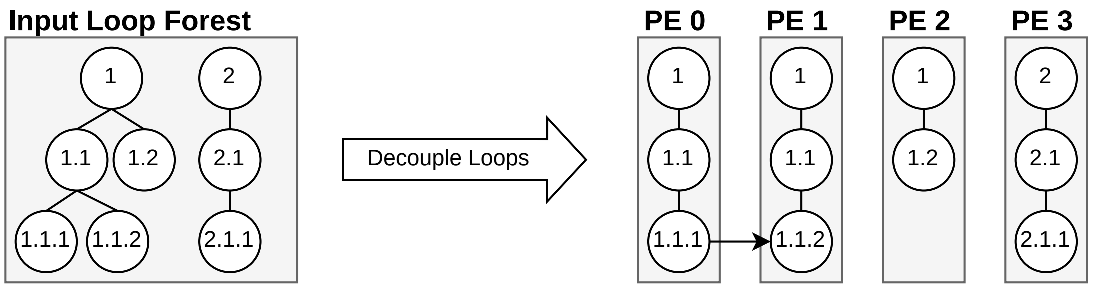


Figure 6.3: Example decoupling of a loop forest. A leaf loop is decoupled into its own PE, which includes loop control of the outer loops. Parent loop body instructions are included only if they come before the leaf loop in the topological order. FIFOs are used to communicate scalar data dependencies (e.g. from loop 1.1.1 in PE 0 to loop 1.1.2 in PE 1). FIFOs are written in the loop exit block and read in the loop pre-header block. Each loop PE might have an AGU producing memory requests fo the accesses in that loop PE.

Using DRAM Bandwidth Efficiently

We use Altera's DRAM IP generated by its HLS compiler to implement DRAM Load-Store Units (LSUs). Our DU can have multiple LSUs connected to the DRAM controller using a ring topology, depending on the number of load/store operations in the input program. To use DRAM bandwidth efficiently, the LSUs coalesce multiple loads/stores into one wide request to the memory controller in order to use the full DDR channel width (512-bit in our case). To achieve this for codes with irregular access patterns, the LSUs use additional logic and buffering to perform coalescing dynamically [235, 12]. DRAM requests are buffered until the largest possible burst can be made. If no new requests arrive in N consecutive cycles, then an incomplete burst is made (in our case N is set to 16 by the FPGA vendor).

Asynchronous address supply is essential for efficient use of DRAM, because of the high access latency, and to allow the dynamically bursting LSU to look ahead in the address stream—addresses should be supplied in advance of the corresponding consumer/producer execution. Streaming FPGA architectures achieve this by following the decades-old Decoupled Access/Execute (DAE) principle [205], where the address generation is decoupled into its own thread of execution, running ahead of the compute threads that consume and produce values [166, 42, 51, 89]. Note that dynamic loop fusion is not limited to DAE architectures; it can be realized in other model of computations, e.g., with dynamic dataflow [126].

DAE Transformation

A DAE architecture is automatically generated by our compiler, following approach described in previous chapters. For completeness, we describe the steps of this transformations here as well, taking the context of codes with multiple loops into account.

Given a forest of loop trees, loop Compute Units (CUs) and AGUs are decoupled into their own PEs following the strategy from Figure 6.3. AGUs feed addresses to the DU; the DU sends load values to and receives store values from CUs. All communication is FIFO based, following a latency-insensitive protocol [88]. To analyze which values should be computed by which decoupled unit and which values should be communicated, we use the def-use chain encoded in the SSA form of the code (each SSA value usage can be traced to its unique definition [194]). We follow a standard approach to automatically generate a DAE architecture [214]:

1. **AGU:** Each memory operation to be decoupled is changed to a `send_address` FIFO write that sends the memory address to the DU.
2. **CU:** Dually, in the CU each memory operation to be decoupled is changed to a `consume_value` or `produce_value` FIFO read function that receive or send values to or from the DU.
3. **Dead Code Elimination (DCE):** We apply DCE in the CU to remove any unnecessary address generation code. In the AGU, we delete side effect instructions that are not part of the address generation def-use chains, and then also apply DCE followed by control-flow simplification to remove redundant basic blocks.

6.2.2 Static Loop Fusion

Polyhedral compilers represent memory operations inside loop nests as integer sets [96, 29, 98]:

1. The *domain* set describes the set of loop iterations in which a statement is executed.
2. The *schedule* set maps domain elements to a point in time. Given two schedule instances, we can determine which one comes first in program order.
3. The *access* set maps domain elements to a point in space, representing the accessed memory location.

For example, the domain (D), schedule (S), and access (A) functions of the i-loop store st_A and j-loop load ld_A in Figure 6.1a are:

$$D_{st_A} = \{st_A[i] : 0 \leq i < N\}, \qquad S_{st_A} = \{st_A[i] \rightarrow [0, i]\}, \qquad A_{st_A} = \{st_A[i] \rightarrow f(i)\}$$
$$D_{ld_A} = \{ld_A[j] : 0 \leq i < M\}, \qquad S_{ld_A} = \{ld_A[j] \rightarrow [1, j]\}, \qquad A_{ld_A} = \{ld_A[j] \rightarrow g(j)\}$$

The set intersection of two access relations can be used to find dependencies between the two corresponding operations.

Static loop fusion for the code in Figure 6.1a can be expressed as a transformation on the schedule of the load: $\mathcal{T}_{Fusion} = \{[1, j] \rightarrow [0, i]\}$ (together with transformations to account for $N \neq M$). $\mathcal{T}_{Fusion}$ might introduce a new dependency between the store and load. The transformation is only legal if the dependency distance of the new dependency is non-negative: this implication has to hold $A_{st_A}[k] = A_{ld_A}[l] \implies k < l$, where k, l are some iterations in the fused loop. In other words, if in the original program a given store writes to an address that a given load later uses, then in the fused loop the store must execute in an earlier iteration than the load.

The legality of loop fusion can be reduced to checking the legality of pairwise loop permutation [186]—the permutation should not break dependencies. However, if the address expressions do not form affine functions, then the legality check does not have enough information about dependency distances to be useful. One can over-approximate non-affine functions as affine [23], but this does not help in all cases, e.g., over-approximation can introduce spurious dependencies on codes with data-dependent addresses. Our dynamic loop fusion is more lenient, requiring only monotonically non-decreasing addresses. However, we stress that our aim is not to replace the polyhedral approach to static loop fusion. Clearly, static loop fusion is preferable whenever possible, especially since it can be combined with other transformations in one framework [186]. Rather, we aim to enable fusion in cases where static approaches are fundamentally infeasible.

6.3 Address Monotonicity

We now describe the concept of address monotonicity in more detail and contrast it with affine addresses.

6.3.1 Motivation for Monotonicity

Assume that we have a memory dependency across loops. If we can prove at compile time that the address of the dependency source is monotonically non-decreasing, then at runtime the loop with the dependency destination only has to check if the address it accesses is lower than the most recently accessed address in the source loop—the dependency destination does not need to see the full history of memory accesses made in the other loop. This paves the way for our efficient hardware dynamic memory disambiguation across loops described in Section 6.5. We now describe how addresses can be proven to be monotonically non-decreasing.

6.3.2 Monotonic Chain of Recurrences

Compilers can represent expressions inside loops as a *Chain of Recurrences (CR)* [16, 80, 184]:

$$\{base, \odot, step\},$$

where $base$ and $step$ can themselves be a CR, and $\odot \in \{+, \times, \div\}$. To reason about memory addresses, we typically use the constraints: $base, step \in \mathbb{N}$ if they are not a CR; and $\odot = \{+, \times\}$. Both LLVM and GCC provide a CR analysis called Scalar Evolution (SCEV) [24, 183].

A CR is *affine* iff it is an add recurrence and iff its step is a constant expression not containing any CRs [98]. A CR is *monotonically non-decreasing* iff its step is non-negative [237]. For brevity, we use the term *monotonic* to mean monotonically non-decreasing in the rest of the paper.

Monotonic CRs are more general than affine CRs and handle control flow better [237]. For example, the CR of a row-major $N \times N$ matrix traversal is affine and monotonic: $\{\{0,+,N\},+,1\}$. But the CR for an FFT traversal is not affine anymore, only monotonic: $\{\{0,+,1\},+,\{2,\times,2\}\}$.

An address expression is monotonic w.r.t. a given loop depth iff the loop CR expression consists of only monotonic CRs. Monotonically non-increasing addresses (i.e., using $step \in \mathbb{Z}$ and adding $\div$ to $\odot$) can also be supported by just flipping signs in the hazard detection logic, but we do not discuss this further in this paper.

6.3.3 Monotonicity in Real Codes

Most codes which traverse data structures produce addresses that are monotonic [101]. For example, graph representations, such as an adjacency list, store nodes in such a way that the traversal over all nodes, or over all neighbors of a node, produces monotonic addresses. Another examples are codes with variable stride accesses, such as FFT-like traversals found in many signal processing workloads, which also produce monotonic addresses.

Programmer Annotated Address Monotonicity

Data-dependent accesses cannot be analyzed using the CR formalism, yet their underlying access pattern is often monotonic. For example, sparse matrix formats like Compressed Sparse Row (CSR) produce address sequences that retain the partial order of the original row-major matrix traversal. Other data-dependent accesses that are not monotonic by definition can be made monotonic with pre-sorting. To support dynamic loop fusion on these codes, we allow the user to annotate memory operations asserting that the address is monotonic in a given loop.

6.3.4 Non-Monotonic Outer Loops

We require a monotonic CR for the innermost loop of the memory dependency source; the outer loop CRs can be non-monotonic. Consider this producer-consumer example:

```
for (i=0; i<ITERS; ++i)
    for (j=0; j<N; ++j)
        store A[j];
for (k=0; k<M; ++k)
    load A[k];
```

The store innermost j-loop is monotonic, but the outer i-loop is not—advancing the i-loop causes the store address to reset. We encode this information in our schedule (Section 6.4), so that in this case our DU will know that it has to wait for the last i-loop iteration to be sure that a given $A[j]$ store address in the j-loop will not be repeated.

For any given schedule n-tuple, where n is the nesting level of the corresponding memory operation, our compiler detects which, if any, of the $n-1$ outer loops has to wait for its last iteration to be sure that the address is not reset in the n-th loop (the innermost loop). For each such loop depth, we add an additional bit to the schedule signaling to the DU if that schedule element was generated on the last iteration of its corresponding loop. Our DU uses this information in the hazard detection logic. This bit is only a hint and is not essential for the correctness of the hazard detection logic, as will be explained in Section 6.5. No last-iteration-hint bits are generated for monotonic loop depths.

Detecting Non-Monotonicity

Given an address expression $f(i_1, i_2, ..., i_n)$ nested within n loops (where n is the innermost loop depth), a $k, 1 \leq k < n$ loop depth is *non-monotonic* if there exists a $j > k$ loop depth such that $CR_k.step < (CR_j.step \times tripCount_j)$, where $CR_k.step$ is the step component for loop k, and $tripCount_j$ is the number of times loop j executes. In other words, a given outer loop k is non-monotonic if there exists a deeper nested loop whose entire execution contributes a larger value to the address value than one k-loop iteration. A CR_k for loop k might not exist, in which case that loop depth is trivially marked as non-monotonic.

For example, the outer loop in a row-major $N \times M, N > 1, M > 1$ matrix traversal is monotonic, because its step is M, which is not lower than $CR.step \times tripCount = M$ of the inner loop. On the other hand, the outer loop in a column-major traversal is non-monotonic, because its step value is 1, which is lower than $CR.step \times tripCount = M \times M$ of the inner loop.

The above expressions are usually symbolic. We substitute symbols with their maximum values (after a value range analysis). This makes our monotonicity checks conservative—we might get false positives, but never false negatives. The checks could be performed at runtime instead, which would make the result precise. However, false positives did not occur in our evaluation, so we leave this for future work.

6.4 Program-Order Schedule for Hardware

Our schedule representation allows multiple loops to run in parallel, as opposed to being sequentialized as in existing dynamic memory disambiguation approaches for HLS [112, 125, 212, 97, 62]. Section 6.2.2 discussed the schedule representation used in polyhedral compilers. We use a similar representation at runtime, but with the following optimizations for hardware:

1. Each loop depth is represented by one element in the schedule tuple, instead of a multi-dimensional point.
2. Each schedule element is incremented by 1 for each invocation of the loop body corresponding to that element—no dependencies between schedule elements are introduced across loops. Repeated invocations of inner loops do not cause the corresponding schedule elements to wrap around.

3. Schedule comparisons between two operations involve just one comparison between the schedule elements corresponding to the innermost shared loop depth of the operations, as opposed to comparing whole tuples as is the case in the polyhedral schedules.

Consider these two nested loops for example:

```
  for (i=0; i<N; ++i)
     for (j=0; j<2; ++j)
        ld_0;
        st;
     for (k=0; k<4; ++k)
        ld_1;
```

Our DAE pass will decouple this code into two loop PEs:

```
for (i=0; i<N; ++i)
   for (j=0; j<2; ++j)
      ld_0;
      st;
```

```
for (i=0; i<N; ++i)
   for (k=0; k<4; ++k)
      ld_1;
```

Assume $i = 1, j = 0$ for the left PE; and $i = 0, k = 3$ for the right PE. The st schedule will be $\{2,3\}$; the ld_1 schedule will be $\{1,4\}$. To check if a st schedule instance comes before a ld_1 schedule instance in program order, written as $schedule_{st} \prec schedule_{ld_1}$, we compare the schedule elements corresponding to the i-loop. Similarly, to check $schedule_{st} \prec schedule_{ld_0}$, we compare the j-loop schedule elements.

The below table shows the difference in evolution of our and the polyhedral schedule representation for the st operation:

loop iterations:	i=0, j=0	i=0, j=1	i=1, j=0	i=1, j=1
polyhedral schedule:	{0, 0, 0, 1}	{0, 0, 1, 1}	{1, 0, 0, 1}	{1, 0, 1, 1}
our schedule:	{1, 1}	{1, 2}	{2, 3}	{2, 4}

The additional dimensions in the polyhedral schedule are used to represent program order within loops. How can we avoid the additional dimensions in our schedule and still recover program order within loops? For example, we want to know that $schedule_{ld_0} \prec schedule_{st}$ even when both schedules will be equal to $\{2,3\}$. Our insight is to *configure the schedule comparator* based on the topological order of memory operations in the program. In a $schedule_{ld_0}[1] \odot schedule_{st}[1]$ comparison, where the index 1 refers to the i-loop, we will configure $\odot = \leq$. Dually, to check $schedule_{st} \prec schedule_{ld_0}$, we would synthesize the comparison: $schedule_{st}[1] < schedule_{ld_0}[1]$.

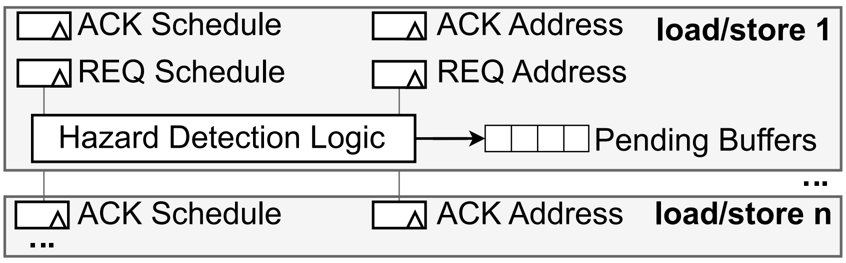


Figure 6.4: Data Unit (DU) consisting of of n Load Store Units.

In summary, our compiler pass statically configures schedule comparators used in the DU for each dependency pair, so that we can recover total ordering without additional schedule dimensions and without the need to compare entire schedule tuples.

6.4.1 Integration of Non-Monotonic Outer Loops

For each non-monotonic outer loop k, we add a $lastIter$ bit to the schedule that will be set in the AGU if the corresponding request was generated on the last k-loop iteration. Our DU uses $lastIter$ bits as hints to expedite disambiguation—they are not essential for correctness. Non-monotonic loops for which $lastIter$ bits cannot be generated are still supported.

6.4.2 Schedule Generation in AGUs

Our compiler adds schedule-generating instructions for each AGU memory request as follows:

1. At the start of the AGU, an n-tuple $schedule$ is initialized to 0, where n is the request loop depth.
2. At each loop depth $1 \leq i \leq n$, a $schedule[i]$ increment instruction is inserted to the beginning of the first non-exiting basic block of the i-loop body.
3. For each non-monotonic loop k, we add a $lastIter[k]$ comparison instruction that evaluates to true if this is the last k-loop iteration. This involves calculating loop predicates one iteration in advance. The $lastIter$ bit is just a hint and is set to false if the loop predicate cannot be calculated one iteration in advance.
4. At the end of the AGU, each $schedule$ element is set to a sentinel value that signals to the DU that there will be no more requests from this AGU.

Schedules are implemented in 32-bit registers and are shared between all memory operations in the same AGU. Future work could use range analysis to decrease schedule bit sizes.

6.5 Data Unit with Hazard Detection

Each program base pointer that has unpredictable dependencies, or that has dependencies across loops that cannot be fused statically, is assigned its own DU to perform dynamic disambiguation. Given a producer-consumer memory dependency pair, the responsibility of the DU is to stall the consumer memory access until the dependency is resolved by the producer. Figure 6.4 shows a high-level DU organization. In our implementation, each program load and store gets its own port; future work could study port sharing.

Each load and store keeps track of the address and schedule corresponding to the most recent ACK received from, and the next request to be sent to, the memory controller. It also has buffers to hold addresses, schedules, and values (in case of stores) for pending requests (not yet acknowledged requests) .

The hazard detection logic compares the address and schedule of its next request with the address and schedule of the most recent ACK of its dependency sources. The next request will only be sent to the memory controller and moved to the pending buffer if the check succeeds. The check and enqueueing logic is spread across multiple pipeline stages—there is no negative load latency impact, because, thanks to the DAE architecture, load addresses run ahead of load consumers giving us ample cycle budget. The pending buffers are implemented in registers to enable associative searching needed for store-to-load forwarding (Section 6.5.5)—their size depends on the DRAM burst size.

In the rest of this section, we describe how the monotonicity property and our schedule representation are used to enable dynamic memory disambiguation across loops.

6.5.1 Hazard Detection Problem Statement

We are trying to check if a memory operation a has a data hazard with memory operation b. Assume a is nested in n loops, b is nested in m loops, and they both share a loop at depth $k, k \leq n, k \leq m$. Informally, given a $req.schedule_a$ and $req.address_a$ corresponding to the next a request, and $ack.schedule_b$ and $ack.address_b$ corresponding to the most recent ACK for operation b, our hazard detection logic deems the next a request safe if either of the two conditions holds:

1. The next a request comes before the most recent b ACK in program order.
2. The next a request comes after the most recent b ACK in program order, however, $req.address_a$ will not be accessed by operation b in the the schedule range from $ack.sdchedule_b$ to $req.schedule_a$.

We now describe each of these points in more detail, before composing the equations implementing these two checks into a general Hazard Safety Check. In the following discussion, we use the term "$(schedule_a, schedule_b)$ time range" to mean the sequence of memory requests b' such that $schedule_a[k] < schedule_{b'}[k] < schedule_b[k]$, where k is the innermost common loop depth of operation a and b. We use open parenthesis and box brackets to represent open and closed intervals, respectively.

6.5.2 Comparing Schedules

If operations a and b do not share any loops ($k = 0$), then the relative schedule program order will always match their topological program order and we do not need to synthesize any comparisons. Otherwise, if the shared loop depth $k > 0$, we synthesize the following comparison to check if the next a request comes before the most recent b ACK:

(Program Order Safety Check)

$$\begin{gathered} req.schedule_a[k] \odot ack.schedule_b[k] \parallel \\ \big(req.schedule_a[k] \odot req.schedule_b[k] \;\&\; noPendingAck_b\big) \end{gathered}$$

Where $\odot = \leq$ if $a \prec b$ in topological program order, else $\odot = <$. The $noPendingAck$ term is a single bit that is set if b is not waiting for any ACKs. The second equation line makes sure that the a request is deemed safe if there are no further b requests in the $[ack.schedule_b, req.schedule_a)$ time range.

Since we only use the schedule element corresponding to the innermost shared loop of the two memory operations, we do not need to synthesize the rest of the schedule.

6.5.3 Checking Address Reset in Schedule Range

If the above check fails, then for request a to be safe we check that operation b will not access $req.address_a$ in the $(ack.schedule_b, req.schedule_a)$ time range. If all operation b loop depths are monotonic, this is a simple $req.address_a < ack.address_b$ check. If some b loops are non-monotonic, we need to guarantee that $ack.address_b$ will not be reset in the considered schedule range:

(No Address Reset Check)

$$lastIterCheck \;\&\; req.schedule_a[l] = ack.schedule_b[l] + \delta$$

Here, $\delta = 1$ if $a \prec b$, else $\delta = 0$; l is the deepest non-monotonic loop depth in the b operation loop nest such that $l \leq k$; and the $lastIterCheck$ term is an `AND`-reduction of the b $lastIter$ bits:

$$ack.lastIter_b = (bit_1, ..., bit_k, \underbrace{bit_{k+1}, ..., bit_{m-1}}_{\text{AND-reduction}}, bit_m),$$

where $bit_j, 1 \leq j \leq m$ is set to 1 at compile time if the j loop is monotonic and thus optimized away from the reduction; otherwise bit_j will be set dynamically on the last iteration of the j loop according to the procedure from Section 6.4.1.

The first term in the No Address Reset Check guarantees that all non-monotonic child loops of k are on their last iteration, and thus will not reset the b address. The second term guarantees that the b address will not reset as a result of advancing in some parent loop of k. Only bits corresponding to non-monotonic loop depths are considered in the `AND`-reduction. Similarly, if all $[1, k]$ loops are monotonic, then the second term is omitted.

Example

Consider the following code:

```
for (; a < A; ++a)          // depth 1: non-monotonic
    for (; b < B; ++b)        // depth 2: monotonic
        for (; c < C; ++c)    // depth 3: non-monotonic
            for (; e < E; ++e) // depth 4: monotonic
                mem_op_b;
        for (; d < D; ++d)    // depth 3
            mem_op_a;
```

Here, the b address is non-monotonic at loop depth 1 and 3. The innermost common loop depth of operations a and b is $k = 2$. The innermost non-monotonic b loop depth that is lower than k is $l = 1$. Thus, the No Address Reset Check synthesizes $req.schedule_a[1] = ack.schedule_b[1]$ to guarantee that b will not have any more l-loop iterations until reaching the $req.schedule_a$ point. And it will check if $ack.lastIter_b[3]$ is set to guarantee that the b address will not reset by advancing in the non-monotonic $3 > k$ loop.

6.5.4 Hazard Safety Check

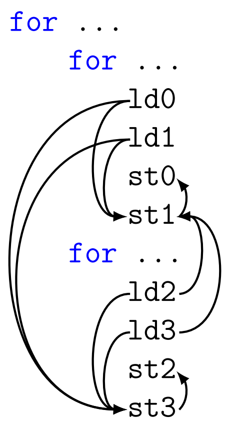
With the ability to compare program order schedules and guaranteeing that addresses do not reset in a given schedule range, we can now construct a general data hazard check. The next a request is safe to execute w.r.t the most recent b ACK if:

$$\begin{aligned} &ProgramOrderSafetyCheck \parallel \\ &\big(req.address_a < ack.address_b \;\&\; NoAddressResetCheck\big) \end{aligned} \tag{Hazard Safety Check}$$

Complexity

The Hazard Safety Check simplifies to just one $req.address_a < ack.address_b$ comparison if a and b do not share loops. If b has non-monotonic loops, then the No Address Reset Check adds at most one `AND` reduction and one equality check. The number of comparisons grows to three if there is a shared loop thanks to the Program Order Safety Check. In general, given a program with n operations, if we check every possible dependency pair, then the number of comparisons is $\mathcal{O}(n^2)$—reducing complexity becomes important as the number of loads and stores grows. Loads do not have to check for hazards against other loads. Also, Write After Read (WAR) checks where the written value depends on the read value can be omitted, as previous work has already pointed out [124].

However, by exploiting the transitive property of our Hazard Safety Check we can prune many more hazard pairs. Assume that we have three memory operations with the following topological program order $c \prec b \prec a$. The safety check of a against c can be omitted, since a already checks against b, and b checks against c. Operation c still has to be checked



```
Before prunning:
    44 hazard pairs
After pruning:
    10 hazard pairs
WAR pairs pruned due to
write depending on read:
    2
Pairs pruned due to transitive
property of Hazard Safety Check:
    32
```

Figure 6.5: Result of pruning hazard pairs in the later evaluated FFT code. Each memory operation checks for safety against at most one operation per loop depth (e.g., ld_0 checks against st_3 in its first loop depth, and against st_1 in the second).

against a if there is a Control Flow Graph (CFG) path via a loop backedge from a to c. With pruning, the worst case number of comparisons reduces to $\mathcal{O}(nd)$, where d is the maximum loop depth. For example, in the an FFT code which we later evaluate, the above pruning procedure decreased the number of hazard safety checks from 44 to 10 (32 checks were pruned due to our transitivity property, 2 due to a store to load dependency). Figure 6.5 shows the result of such pruning.

6.5.5 Store-to-Load Forwarding

We support store-to-load forwarding by allowing loads to directly access values from a dependent store's pending buffer. We specialize the Hazard Safety Check for RAW dependencies: instead of using the address and schedule of to the most recent store ACK, we use the address and schedule of the next store request. In addition, we perform an associative search of the pending store buffer, using the load address as a key. If the modified RAW check succeeds, then the dependent value will either already have been committed and acknowledged, or it is in the store pending buffer and our associative search will find it. Hits from the buffer search can be used by the load directly, without issuing a DRAM request. If there are multiple values with the same address in the pending buffer, the youngest is chosen (this is cheap to implement in FIFO buffers).

The case where two stores that can both forward a value with the same address to the same load is impossible. Assume the following program order of operations that all use the same address: $store_0 \prec store_1 \prec load$. The $store_1$ will not be able to move its value to its pending buffer until after the $store_0$ value has been acknowledged—its Write After Write (WAW) hazard detection will stall it. Conversely, the $load$ will not use the $store_0$ value, because it will stall on the RAW check against $store_1$—the $load$ will wait for $store_1$ to move its value to its pending buffer.

With forwarding, some WAW checks cannot be pruned anymore, because load RAW checks do not use store ACKs. In our above example, if all operations are in the same loop, then the $store_0$ WAW check against $store_1$ cannot be pruned, because the $load$ ACK might be updated as a result of store forwarding from $store_1$, with the forwarded value not yet being acknowledged in $store_1$.

6.5.6 Intra-Loop RAW Hazards

A timely disambiguation of RAW hazards, where both the load and store are in the same loop PE, is crucial since any unnecessary stalls would be repeated on every iteration, resulting in a large throughput reduction. As our evaluation in Section 6.7 will show, store-to-load forwarding becomes crucial in intra-loop RAW dependencies.

In addition to forwarding, there is another term needed in the RAW Hazard Safety Check to make intra-loop RAW hazard checks timely. Consider this simple code:

```
for (i = 0; i < N; ++i)
    d = data[i];
    data[i] = work(d);
```

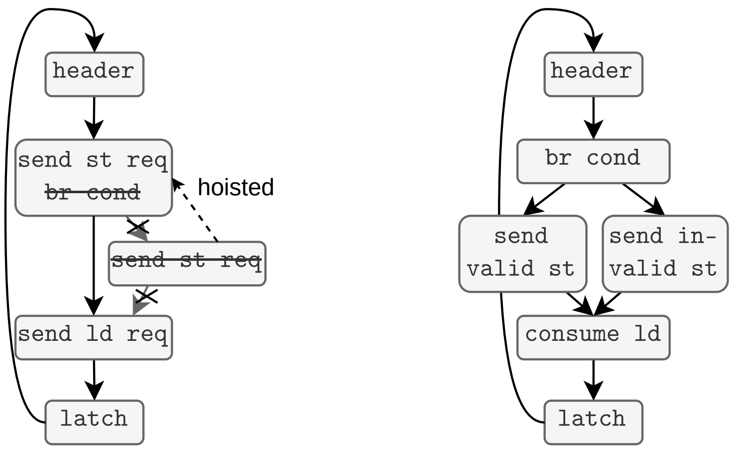
The load and store address distribution is $\{0,1,2,...\}$—there is no actual RAW hazard, but assume that we do not know this at compile time. In this situation, the RAW Hazard Safety Check for a given load at iteration k will only succeed once the next store request in the DU is for iteration $k-1$ and there are no outstanding store ACKs. If the next store request is for an earlier iteration, e.g., an earlier store request is waiting for its store value, then the load would have to be stalled, even though it would be perfectly safe to execute it.

We solve this issue by adding a $NoDependence$ single-bit term to the RAW Hazard Safety Check. For each intra-loop RAW hazard pair, $NoDependence$ is set in the AGU to the result of $req.address_{load} > req.address_{store}$, where $req.address_{load}$ is the next load address to be sent to the DU, and $req.address_{store}$ is the most recent store address that was sent to the DU. When $NoDependence$ is true, and the No Address Reset Check evaluates to true, then the load can be deemed safe since the monotonicity property implies that all store addresses up to $req.schedule_{load}$ are lower than $req.address_{load}$.

Note that a similar check is not needed for intra-loop WAW dependencies, since stores do not stall the datapath if sufficient buffering is provided for the store values.

```
for ...
    if (cond)
        store;
    load;
```

(a) Store in *if*-condition.



(b) AGU speculates requests. **(c)** CU kills mis-speculations.

Figure 6.6: Memory requests in *if*-conditions are speculated using the transformations described in chapter 5.

6.6 Handling Control Flow

The Hazard Safety Check relies on the ability of the DU to detect that a given memory operation has completed a certain schedule time range or a certain address range. This assumes that AGUs supply an operation's schedule and address for every loop iteration. This assumption is broken by operations inside *if*-conditions, which can lead to a deadlock. Consider the code in Figure 6.6(a). If the *if*-condition in this loop is never true, then the store will never update its ACK address and schedule, and thus the RAW Hazard Safety Check in the DU would never succeed. Eventually, the AGU would fill the load request FIFO, resulting in a deadlock.

This could be avoided by using separate AGUs for each memory operation—the store AGU would be guaranteed to at least send a final sentinel value, which would eventually cause the RAW hazard check to succeed. However, this would again mean that some loops need to run to completion before the check can be performed.

A better approach is to *speculatively* send memory requests. We adapt our work from the previous chapter that implements speculation in a DAE architecture. In our example, the store request can be hoisted out of the *if*-condition in the AGU. Then, the store values going to the DU from the CU can be tagged with a *valid* bit that signals if the value should be committed or not, depending on the actual control flow at runtime. Figure 6.6 shows the AGU and CU CFGs that implement such speculation.

Previous work used speculation to remove Loss of Decoupling (LoD) problems in DAE architectures [212, 104, 103]. A LoD arises when the AGU has dependencies on values that have to be loaded from a DU or calculated by a CU, preventing the AGU from running ahead [26]. Our approach is the same as previous work, but we apply it to all *if*-conditions with the goal of producing an $(address, schedule)$ pair for each loop iteration in the AGU. As a side benefit, speculation also makes us immune to the control-dependency LoD problem.

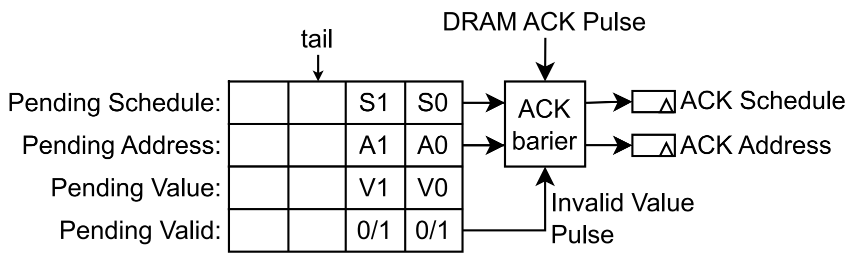


Figure 6.7: Handling of mis-speculated stores in the DU. Before being moved to the pending buffer, invalid stores are also checked for safety to uphold the transitive property of the Hazard Safety Check. They do not submit DRAM requests. When reaching the head of the pending buffer, they update the ACK registers without having to wait for an ACK.

Mis-speculated loads are executed normally in the DU. The read in the CU CFG is moved to the same location where it was speculated in the AGU. This guarantees that the order of load requests made from the AGU is the same as the order of load value consumption in the CU, on every CFG path. After reading a speculated load value, the CU can simply not use it if it takes a CFG path where the load value is not needed. Since the basic block location of the speculated load value consumption changes, we also need to adjust any ϕ-nodes that use the load value.

Mis-speculated stores are detected using the *valid* bit in store values coming from the CU. Invalid stores are never committed to memory—there is no need for costly rollbacks. However, invalid stores should eventually update the ACK registers to signal that a given time and address range was completed by the store. Figure 6.7 shows our approach to this.

If a whole loop that contains memory operations is under an *if*-condition, then we fold the *if*-condition into the loop body and execute the whole loop speculatively. This was not a performance problem in our evaluation, but future work could investigate a whole loop speculation scheme that does not require executing all loop iterations.

6.7 Evaluation

We implemented our compiler-hardware co-design in the Intel HLS compiler [117] version 2023.1.0. Figure 6.8 shows our tool flow. We have a generic DU template which is parameterized given information about the input code gathered by our compiler analysis, such as the number of memory operations and their respective address expression monotonicity analysis, the dependency pairs for which hazards need to be checked, and their respective topological ordering in the program CFG, which is used to check the respective program order of two memory operations efficiently. Our implementation and evaluation are publicly available [210].

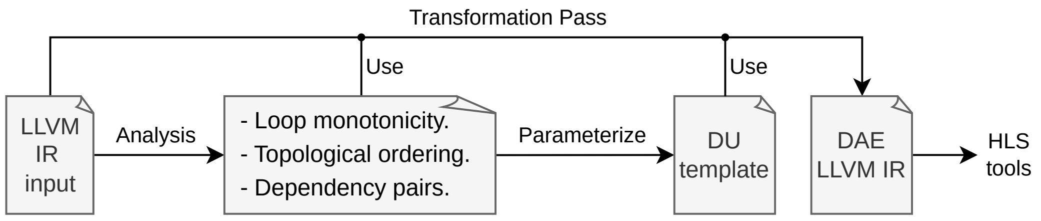


Figure 6.8: Our compiler/hardware co-design flow. We use the Intel HLS tool in this paper. Our DU is parametrized by the number of loads and stores. The DU disambiguation logic is parameterized for each hazard pair (dependency source and destination) based on the loop nest monotonicity of the dependency source; and the relative topological ordering of the dependency source relative to the destination.

6.7.1 Methodology

We evaluate dynamic loop fusion on ten benchmarks where there is a possibility for parallelism across loops that is not exploited by current static and DDF HLS tools. All baselines use the Intel HLS compiler:

- `STA`: baseline Intel FSMD HLS compiler performing automatic static loop fusion. This approach uses the same dynamically coalescing LSU as our DU.
- `LSQ`: an implementation of dynamic scheduling within the Intel HLS compiler using the work from the precious chapters [214]. An LSQ is used for memory accesses, but without support for dynamic coalescing. This approach is representative of all current LSQ implementations in HLS [112, 125, 212, 97, 61, 62].
- `FUS1`: the dynamic loop fusion approach described in this paper, but with no store-to-load forwarding.
- `FUS2`: `FUS1` with store-to-load forwarding enabled.

We execute our benchmarks in hardware on the Altera Arria 10 GX1150 FPGA board [119] with 2 banks of DDR4 memory (the memory controller uses two 512-bit channels). We use large datasets to ensure data is distributed across DRAM pages, resulting in variable latency. Each code is executed three times and the minimum time is reported. Area, reported as Adaptive Logic Modules (ALMs) [115], and frequency are taken from Quartus 19.2 reports after place and route. Our approach does not increase the number of DSPs or BRAMs.

6.7.2 Benchmarks

We use irregular codes from DDF HLS research [45, 126, 214], choosing codes where there are sibling loops that can benefit from our dynamic loop fusion. For some benchmarks, we unroll outer loops to expose two inner loops that can be dynamically fused; or we compose multiple kernels to simulate applications composed of multiple tasks. Some codes have address expressions that can be analyzed for monotonicity; some codes use data-dependent accesses that are asserted to be monotonic by the programmer. We now list our benchmarks and the parameters used in this evaluation:

- **RAWloop, WARloop, WAWloop:** each benchmark has two loops, each with one memory access, forming a RAW, WAR, or WAW dependency across loops. We use these benchmarks to compare our speedup to the maximum theoretical speedup. Complexity $\mathcal{O}(n)$. We set $n = 10\,000\,000$.
- **bnn:** one layer of a sparse binarized neural network. There are two loops, both with data-dependent accesses that prevent fusion. We mark the inner loops as monotonic since we know that the sparse representation is monotonic. Complexity $\mathcal{O}(n^2)$. We set $n = 10\,000$.
- **pagerank:** uses a compressed sparse row (CSR) format to iterate over the graph. Another two loops in the algorithm have a regular access pattern, but they cannot be fused because the irregular loop is between them. The complexity is $\mathcal{O}(iters \times (nodes + edges))$. We set $iters = 10$, $nodes = 325\,729$, $edges = 1\,497\,134$ using the graph called *web-NotreDame* from [146].
- **fft:** an FFT with the middle loop unrolled by a factor of two. The non-affine accesses prevent loop fusion. The `LSQ` and `STA` approach is equivalent for fft, because there are no hazards within loops that would need an LSQ. Complexity $\mathcal{O}(n \log n)$. We set $n = 1\,048\,576$.
- **matpower:** sparse matrix power using the CSR format with the outer loop unrolled by a factor of 2. Complexity $\mathcal{O}(nz^3)$, where nz is the number of non-zero matrix values. We set $nz = 4\,096$.
- **hist+add:** addition of two histograms. The `STA` approach can fuse the two histogram loops, but not the addition. Three $\mathcal{O}(n)$ loops. We set $n = 10\,000\,000$.
- **tanh+spmv:** $tanh$ applied to a vector before it is used in a COO sparse matrix-vector multiplication. The $tanh$ loop has a store in an if-condition, which we speculate. One $\mathcal{O}(n)$ loop followed by a $\mathcal{O}(nz)$ loop, where nz is the number of non-zero matrix values. We set $n = 10\,000$, $nz \simeq 10\,000$.

6.7.3 Results

Table 6.1 printed at the end of this chapter shows detailed area and performance results for the four approaches that we evaluate. Figure 6.9 visualizes the speedup of dynamic loop fusion over the other approaches, together with the area overhead. Dynamic loop fusion with forwarding is on average **14× faster than FSMD HLS** and **4× faster than DDF HLS** that uses an LSQ.

The following paragraphs describe the results in detail.

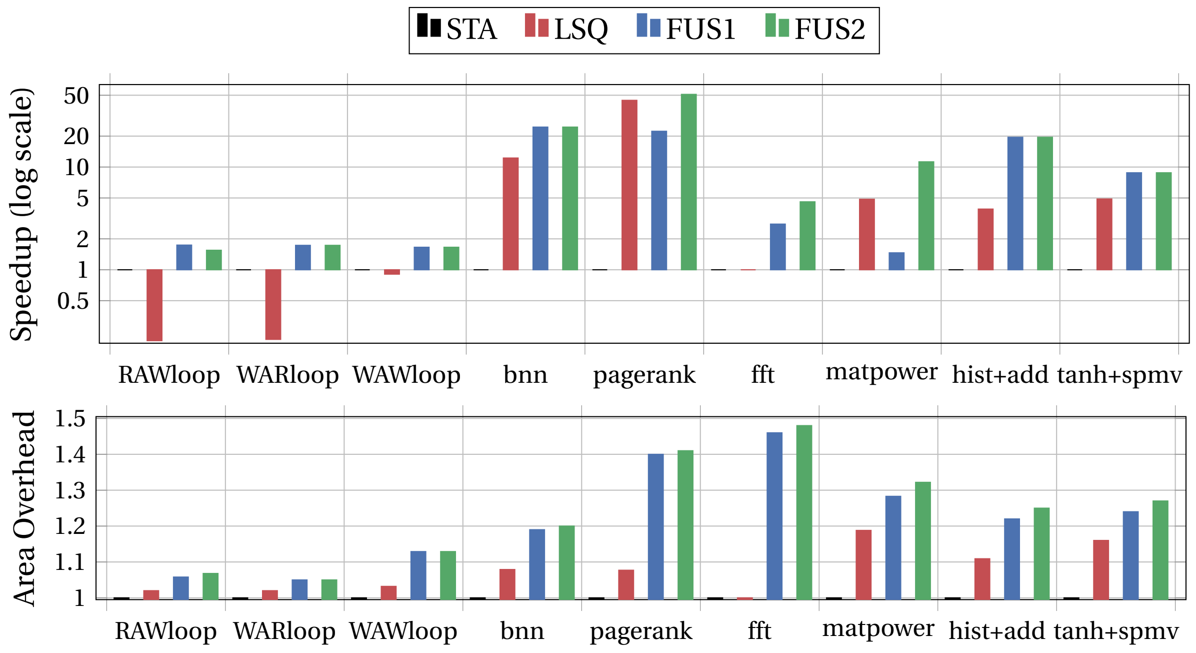


Figure 6.9: Speedup and area usage normalized to the FSMD HLS approach. A speedup below 1 for the LSQ approach implies a slowdown relative to the FSMD approach.

Theoretical Speedup

The RAW/WAR/WAW loop benchmarks have a theoretical speedup of 2×, but FUS2 achieves a speedup of around 1.7×. The lower speedup is due to the lower FUS2 circuit frequency on these benchmarks. The LSQ approach sees a slowdown relative to STA in the RAW/WAR loop benchmarks, because it cannot use a dynamically bursting LSU which stalls the load loop significantly (the LSQ used in [214] uses a non-bursting LSU to guarantee that hazards are not violated [212]). Store loops, e.g., WAWloop, do not suffer as much from a lack of bursting in the LSQ approach, because stores do not stall the LSQ pipeline.

Impact of Store-to-Load Forwarding

We observe that forwarding has no observable benefit on codes where the forwarding happens across loops, e.g., RAWloop. This is expected in our evaluation setup, since without forwarding, the only penalty is an initial wait for the store ACK to be updated. Forwarding across loops may become beneficial if the DRAM bandwidth becomes a bottleneck, which is likely to occur in practice once data parallelism is exploited. Forwarding becomes crucial if the store and load are in the same loop and the dependency distance is lower than the store latency (e.g., fft, matpower, or pagerank). Future work could use a more precise cost model and enable forwarding only where beneficial, e.g., always use forwarding for RAW dependencies inside loops, but for RAW dependencies across loops enable it only once the memory bandwidth is saturated.

Which Codes Benefit from Dynamic Loop Fusion?

It only makes sense to fuse loops with similar time complexities. Consider the pagerank benchmark as an example where fusion offers only a modest 1.1× speedup over the `LSQ` approach. The code consists of two $O(n)$ loops which go over graph nodes and one $O(n^2)$ loop which goes over edges. Even if all three loops are fused, the runtime will still be dominated by the $O(n^2)$ loop. We used the `web-Google` graph [146] with 875,713 nodes and 5,105,039 edges, which only has a theoretical speedup of ≈ 1.3 over `LSQ`.

We see the biggest benefit of using dynamic loop fusion in the ability to unroll outer loops of irregular codes without having to worry about breaking data dependencies (e.g., fft and matpower), and in the ability to perform task fusion at a fine-grained level (e.g., hist+add and tanh+spmv).

Area Overhead

Dynamic loop fusion with forwarding comes at an average area increase of 24% and frequency degradation of 9% over FSMD HLS. The most area-hungry component is the dynamically coalescing LSU. The `STA` approach also uses the costly coalescing LSUs, which amortizes the area overhead of fusion. The `LSQ` approach uses a simpler LSU, which explains its low area overhead.

For example, in the RAWloop benchmark, the `FUS2` DU consumes 1,550 ALMs (1,200 of which are dedicated to the pending buffers and its associative searching), whereas a single load LSU consumes 2,840 ALMs and the DRAM interconnect consumes 68,089 ALMs. If the OpenCL kernel runtime and DRAM interconnect are not counted, then our area overhead of dynamic loop fusion with forwarding increases to 2.1×. However, codes not using DRAM will not need the area budget for pending buffers, resulting in an overhead closer to what we report in table 6.1.

Hazard pairs pruning has a large impact on the area and critical path of codes with many loads and stores. For example, the FFT code uses two DUs, each with 4 loads and stores. The unpruned FFT `FUS2` version uses 32% more area and achieves a 28% lower frequency than the pruned version. Our DU implementation is not particularly optimized and we expect its already modest critical path and area overhead can be brought down significantly, especially if the logic between the bursting LSU and our DU can be shared—due to the closed source nature of the HLS tool that we used in this evaluation, this was not possible in this work.

6.8 Limitations

Our dynamic loop fusion approach can be integrated with common loop transformations used in HLS. For example, loop tiling does not break the monotonicity of inner loops. Dataflow designs—concurrent loops communicating via FIFOs–are automatically generated by our compiler given the program loop forest, as shown in Figure 6.3. Loop unrolling—that is replicating the datapath of the inner loop—is also compatible with our DU, since we do not impose any limits on the number of memory ports. However, our current implementation does not work with automatic loop unroll pragmas, and manual unrolling is needed instead. This is because unrolling pragmas are typically implemented in the closed-source back-end of vendor compilers, and our compiler passes operate on LLVM IR in the middle-end. We do not study unrolling in this work, because the types of irregular codes that we consider in this paper do not lend themselves to the same automatic parallelization as regular codes, e.g., due to unpredictable loop-carried dependencies.

In this work, we consider DRAM streaming applications, relying on a dynamically bursting and coalescing LSU to discover memory parallelism at runtime and on store-to-load forwarding to increase temporal locality. In our current design, the amount of on-chip data reuse is limited by the size of the pending buffers, which have to be kept small to make associative searching feasible. A cache memory hierarchy implemented in BRAM could further decrease the number of DRAM requests and increase temporal locality. Recent work has advanced the state-of-the-art of non-blocking caches on FPGAs by storing Miss Status Holding Registers (MSHRs) in BRAM and using hash-based, instead of associative, searching [240, 13]. In a DU with cache, the pending buffers could be changed to MSHRs with added schedule information and our store-to-load forwarding could be removed altogether, since temporal locality would be provided by the cache. We will explore this future work in Section 7.3 in the next, concluding chapter.

Using a BRAM-based cache and loop unrolling are orthogonal goals—supporting multiple memory ports is cheaper to do in BRAM than in DRAM, both in terms of the circuit area and available bandwidth. However, since the automatic partitioning of BRAM into multiple banks cannot be performed for irregular code, a memory arbiter would have to be integrated to support multiple BRAM ports, as for example in [46, 243, 242].

6.9 Related Work

Our loop monotonicity analysis benefits from decades of research on abstract interpretation of recurrences [5, 28, 149, 16, 80, 184]. Loop monotonicity has first been exploited in a practical setting by Gupta *et al.* to synthesize race detection runtime checks in fork-join parallel programs [101]. However, they did not consider shared loops and non-monotonic outer loops.

We have already discussed the basics of the polyhedral compiler transformation framework in Section 6.2.2, stating that codes with non-affine memory addresses or loop bounds cannot be analyzed. Recent work has combined the Inspector/Executor (I/E) approach [201, 160, 70] with polyhedral transformations [207, 197, 228]. The idea behind the I/E approach is to generate inspector code which gathers values of variables unknown at compile-time; and/or rearranges data structures in memory for better locality and to increase dependency distances. The small overhead of the inspector code is offset by the throughput improvement obtained in the executor code. For example, Strout *el al.* proposed the Sparse Polyhedral Framework which uses "uninterpretable functions" to represent non-affine terms such as data-dependent memory accesses [207]. By proving basic properties about an uninterpretable function in the inspector code (e.g., monotonicity) a large amount of potential data dependencies can be ruled out, allowing the executor code to exploit more parallelism [227]. Most recently, [52] proposed sparse fusion, an I/E technique that inspects the access patterns of multiple irregular loops and then creates an execution schedule that allows sibling loops to execute in parallel. Although we share the same goal as these works, our approach is fundamentally different. Instead of relying on inspector code to discover data dependencies, we resolve data dependencies "on the fly" in our DU. We also provide a finer-grained monotonicity compiler analysis, discovering which specific loop depths cause an address expression to visit an earlier value, rather than deciding on the monotonicity of. Our dynamic loop fusion work has no such limitation.

All previous work on dynamic memory disambiguation in HLS sequentializes loops that share a data dependency [125, 77, 212, 2, 97]. Cheng *el al.* investigated compile time checks to prove that two loops do not access the same memory locations [48]—their approach is the same as existing polyhedral optimizers, but uses a different formulation. Others have exploited the SCEV framework to augment the static analysis with dynamic checks in HLS [152, 153, 68]—these approaches are similar to multi-versioned SIMD CPU code, where the fast (SIMD) path is taken if a set of conditions evaluates to true at runtime. All these works either only improve the throughput of single loops, or execute separate loops in parallel only if all iterations are independent. They rely on the commutative property because they cannot guarantee sequential consistency of accesses to the same memory spaces. Our dynamic loop fusion work has no such limitation.

6.10 Conclusions

In this chapter, we have presented dynamic loop fusion, a compiler-hardware co-design approach that enables dynamic memory disambiguation across loops without the need for address history searches. Our hazard detection logic is enabled by constraining our transformation to monotonic loops and by a novel program-order schedule representation, and by assuming monotonically non-decreasing addresses are in inner loops. We have presented a compiler analysis, based on the chain of recurrences formalism, to detect loop monotonicity. We have also shown that most codes contain addresses that are monotonic, making our approach applicable to a large class of applications. On an evaluation of 10 irregular codes, dynamic loop fusion provided an average speedup of 14× over FSMD HLS and 4× over DDF HLS. As far as we know, our work is the first attempt to exploit parallelism across loops with memory dependencies in irregular codes.

Table 6.1: Performance, area usage, and circuit frequency of the `STA`, `LSQ` [214], `FUS1`, and `FUS2` approaches. The second column reports the number of PEs and DUs generated by our `FUS` approach, together with loads and stores per DU.

Kernel	**Number of**				**Area in 1000s of ALMs**				**Freq in MHz**				**Time in seconds**			
	PE	DU	LD	ST	STA	LSQ	FUS1	FUS2	STA	LSQ	FUS1	FUS2	STA	LSQ	FUS1	FUS2
RAWloop	2	1	1	1	78	79.6	82.5	83.3	304	268	263	239	6.8	33.3	3.9	4.4
WARloop	2	1	1	1	78.1	79.6	82.2	82.2	279	264	261	261	7.1	33.5	4.1	4.1
WAWloop	2	1	1	1	78.3	80.8	88.4	88.4	294	269	251	251	6.8	7.5	4.1	4.1
bnn	2	1	2	2	78.9	85.1	93.5	95.2	279	244	266	257	39.2	3.2	1.6	1.6
pagerank	3	2	2/1	2/1	81.5	87.8	114.1	115.2	262	237	246	246	35.7	0.8	1.6	0.7
fft	2	2	4/4	4/4	102.7	102.7	150.4	152.2	246	246	221	219	7.8	7.8	2.8	1.7
matpower	2	1	4	2	82.1	97.6	105.4	108.6	274	193	260	257	18	3.7	12.3	1.6
hist+add	3	2	2/2	1/1	79.2	87.9	97.0	99.3	286	220	282	270	3.9	1	0.2	0.2
tanh+spmv	2	2	2/1	1/1	80.2	93.1	99.5	101.8	274	225	260	264	4.4	0.9	0.5	0.5
				Harmonic Mean:	1	1.07	1.22	1.24	1	0.86	0.92	0.9	1	0.12	0.1	0.07

Chapter 7

Conclusion

In this chapter, we summarize our research contributions and show how they contribute to our thesis that a closer compiler-hardware co-design enables a more efficient HLS. Then, we discuss future work that will address the limitations of our existing implementation. Finally, we conclude with an open-ended discussion of the remaining research challenges in the fields of HLS and spatial computing and we describe possible research efforts that could address them.

7.1 Summary

This thesis shows that closer co-operation between the compiler and the instantiated hardware IPs during the HLS process can result in hardware that achieves higher throughput, uses fewer resources, and can operate at a higher frequency when synthesizing irregular codes compared to previous state-of-the-art HLS tools.

The above goal was achieved by proposing selective dynamic scheduling discovered by the compiler in Chapter 3, which compared to the state-of-the-art Dynamic Dataflow (DDF) HLS tool [126] lowers the area usage by 2.1×, increases the maximum circuit frequency by 2.5×, while also increasing throughput in many codes by 2×—resulting in an area-delay improvement of over 10×. The foundational technique enabling selective dynamic scheduling is automated program slicing, informed by our compiler analysis, which decouples the program along sources of irregularity. We have shown that such a decoupled execution is a more efficient way to introduce dynamic behavior into circuits compared to previous state-of-the-art approaches.

In Chapter 4, we have shown that a closer co-design of a Load-Store Queue (LSQ) IP and the compiler passes that instantiate the LSQ, together with a compiler-based DAE transformation decoupling the address generation part of the code, results in a much lower cost of supporting dynamic memory scheduling than previous state-of-the-art works. This work further improves the area efficiency and maximum circuit frequency of our approach compared to DDF HLS [126] when an LSQ is needed, with an area-delay improvement of 14×.

In Chapter 5, we have extended our dynamic memory scheduling approach, which relies on the DAE technique, with speculation support in the compiler. This contribution is not only vital in the context of using DAE in HLS-based accelerators, but as we have argued in [215], it also enables the use of DAE speculation in many different high-performance architectures. This contribution further increases our area-delay advantage by a factor of two, meaning that our approach results in an area-delay improvement of up to $30\times$ compared to DDF HLS [126] on codes that can benefit from our speculation.

Finally, in Chapter 6, we have demonstrated that a close compiler-hardware co-design and the idea of compiler-based decoupling can enable new forms of parallelism by proposing dynamic loop fusion. If the memory addresses in an irregular code are monotonically non-decreasing, then dynamic loop fusion can replace the LSQ-based memory memory disambiguation proposed in Chapter 4, allowing us to execute multiple loops in parallel, where they had to be sequentialized before. Even single-loop programs, the Data Unit (DU) from the dynamic loop fusion chapter can reduce the area usage and critical path compared to our LSQ, because it does not rely on expensive address history searches. When possible, dynamic loop fusion gives us another $> 4\times$ area-delay improvement over DDF HLS by enabling parallelism across loops.

In summary, our thesis makes several important steps to make the HLS of irregular codes more efficient. On codes that can benefit from all our contributions, our approach to HLS of irregular codes can result in up to two orders of magnitude over the previous state-of-the-art DDF HLS approach [126]. This proves our claim made in the introduction chapter that a closer compiler-hardware co-design and the idea of decoupled execution can improve the quality of the HLS of irregular code by orders of magnitude.

7.2 Composability of Transformations

Our transformations proposed throughout the technical chapters are composable. A given irregular code might be decoupled into multiple Finite State Machines with Datapath (FSMDs) using the technique from Chapter 3. Then, memory operations in individual FSMDs might be connected to our LSQ as described in Chapter 4. This involves decoupling the FSMD that holds the memory operation further to create a DAE architecture. The case where multiple FSMDs produce memory requests for the same LSQ, and thus have to communicate tag tokens, is also covered by treating the tag tokens as input and output dependencies in the Chapter 3 transformation. Similarly, the algorithms from Chapter 5 that implement speculative memory requests in such a DAE architecture compose without issues with the above transformations.

In our actual implementation, the transformations are expressed as rewrite rules[1]. We first analyze the input code to determine which of the transformations are beneficial, creating the required rewrite rules. During the analysis stage, we can already calculate how many FSMDs are required, and we map each rewrite rule to a specific FSMD. Then, we apply the actual transformations by creating the required number of FSMDs, and we iteratively apply matching rewrite rules (we order the rewrite rules according to a predetermined partial order).

The dynamic loop fusion transformation from Chapter 6 can be seen as a specialization of the LSQ from Chapter 4. The DU in dynamic loop fusion also performs dynamic memory disambiguation, but it does so across the entire program in parallel, instead of in one loop at a time, and it is more efficient in terms of area usage and critical path because there is no need for an address history search as is the case in an LSQ. However, whereas the LSQ can work on an arbitrary memory address distribution, the DU in dynamic loop fusion can only work on monotonically non-decreasing addresses. Thus, when possible, an implementation should choose to use the DU from Chapter 6 for dynamic memory disambiguation, even when the program has a single loop, and fallback to an LSQ when the address distribution cannot be proven to be monotonic.

7.3 Limitations & Future Work

Our implementation has several limitations, highlighted throughout the technical chapters, that can be addressed with future work.

Support for Vector Parallelism

Our current memory interfaces can be extended to support a vectorized datapath. Such a support would not be able to just use existing techniques, since there might be data hazards between the elements within a vector. This is the main reason why our evaluations we did not included vectorized code.

General vector execution support for irregular code is a hard problem. Vector ISAs in CPUs use masking to support irregular code. Our memory interfaces (the LSQ from Chapter 4 and the DU from Chapter 6) could use a similar approach. In fact, the typically memory protocols like AXI or BRAM requests already support the masking of individual bytes.

1. `https://github.com/robertszafa/elastic-sycl-hls/blob/main/include/AnalysisReportSchema.h`

Support for a Cache Hierarchy

To further increase data locality in HLS-based accelerators for irregular codes we could use a cache hierarchy, exploiting the high throughput of FPGA BRAMs. Recent work has advanced the state-of-the-art of non-blocking caches on FPGAs by storing Miss Status Holding Registers (MSHRs) in BRAM and using hash-based, instead of associative, searching [240, 13]. We could apply the same principles of a closer compiler-hardware co-design to produce finely tuned cache hierarchies specialized for a given code. A cache would subsume the store-to-load forwarding of the DRAM-based LSQ in Chapter 4 and of the dynamic loop fusion forwarding in Chapter 6, increasing the data locality compared to our current implementation.

More Robust Code Splitting

Implementing our ideas in the Intel SYCL C++ compiler meant that we could high-quality achieve results relatively quickly, however, a more realistic implementation would need to use a different design. This is because the backend of Intel SYCL C++ compiler is closed source, introducing a number of complications. One of the complications is that our code splitting transformations that automatically generate a DAE architecture have to be implemented using SYCL kernels. SYCL kernels are a user facing feature, and from an engineering perspective are undesirable as an implementation vehicle of optimization transformations, not to mention that each additional SYCL kernels incurs area overhead due to the runtime having to manage its invocation, arguments, completion, etc.

Due to these problems, future work could implement our ideas in an open-source HLS compiler, that would allow for a more robust handling of the DAE code splitting transformation. One could go even further by using an IR and computational model with explicit support for decoupling sections of code, something we discuss in the next section. We discuss CIRCT [91] as one possible such tool in the next section.

More Accurate Cost Model

Another problem of using a closed source HLS tool is that its cost model of the operation latencies, throughput, and area usage is not accessible. An accurate cost model is crucial to our analysis from Chapter 3 that decides if a given code section should be scheduled dynamically or statically. In our work, we have solved this problem with micro benchmarks to gather estimates of these metrics for our specific FPGA board. However, as we have mentioned in the limitations section in Chapter 3, such an approach is not scalable and might not produce accurate results. For example, the closed-source back-end tool might decide to perform operation chaining under certain conditions—executing multiple dependent operations in a single clock cycle—which would throw of our analysis. As mentioned in the above paragraph, an open source HLS tool would solve this problem by providing a direct API to its underlying cost model.

7.4 Discussion & Future Directions

In this section, we discuss future work that goes beyond the implementation problems of this thesis and provides a more forward-looking treatment of HLS and the field of spatial computing.

7.4.1 High-Level Synthesis from Sequential Control Flow to Guarded Atomic Actions

We discussed the Guarded Atomic Actions (GAA) computational model in Section 2.3, and mentioned that it has the potential of making our decoupling transformations developed in this thesis easier to implement from the compiler engineering perspective. We believe that the GAA model could provide the semantics needed to reason about decoupling sections of code without the need for explicit code splitting. The GAA computational model is already used to implement latency-insensitive designs composed of multiple co-operating Finite State Machines [170]—such architectures are very similar to the DAE architectures generated by our compiler passes.

The difficult part of using the GAA model in HLS is the translation of sequential control flow to GAA. The challenge lies in composing the rules in such a way that parallelism is maximized, but the accesses to shared state do not break sequential consistency encoded in the sequential control flow program. In a way, this is approach is attacking the problem of generating pipelined designs from a new vantage point and might turn out to have more potential than existing static pipelining algorithms and dataflow circuits. Furthermore, the GAA model has strong formal foundations, making circuits generated by an HLS tool using this model potentially easier to verify [64].

Guarded Atomic Actions MLIR Dialect

Future work could develop an MLIR dialect to model the GAA computational model. As mentioned above, a GAA dialect could significantly reduce the complexity of the decoupling transformations developed in this thesis. Such a dialect, together with a translation from a sequential control flow MLIR dialect (e.g., `scf`) could be used as a core of a new HLS tool. CIRCT [91] is a framework of MLIR hardware-oriented dialects and tooling that provides, among other things, an HLS driver. A new GAA-based HLS tool could reuse the existing CIRCT infrastructure.

7.4.2 New Intermediate Representations for High-Level Synthesis

Starting with the development of LegUP HLS at the University of Toronto [38], all current commercial HLS tools use LLVM IR. Recently, MLIR (described in Section 2.1.2) has emerged as a tool which significantly simplifies the creation of new IRs. Since then, many efforts have been made in the HLS community to encode hardware specific information in the IR by creating new MLIR dialects. Most of the efforts can be found in the CIRCT project, which contains tens of hardware-specific dialects aimed at making domain problems like scheduling or verification easier [91].

We believe that there is still a greater innovation potential in using hardware-specific IRs. One step towards this is efforts like the LLHD IR proposed in [204], which in addition to solving many of the pain points of using Verilog or VHDL as an IR passed between tools, enables spatial co-ordinate information to be attached to the IR. Such additional information can be used to enable high-level tools, like HLS compilers, to guide low level tools, like place & route tools, in order to achieve a given performance metric. Today, such an interface is starkly lacking, but will become increasingly important as the size of FPGAs grows and a greater co-operation between low- and high-level tools becomes necessary [100, 99].

7.4.3 New Spatial Dataflow Architectures

Using FPGAs is challenging because of their fine-grained reconfigurable architecture, which is requires lengthy place & route procedures to create a design. On the other hand, the reconfigurability is desirable as an insurance policy against future algorithmic changes that could not be accommodated by an ASIC. However, there is a large design space between FPGAs and ASICs in the level of reconfigurability. CGRAs trade bit-level configuration ability for a higher compute density and a shortened reconfiguration time [187, 165, 94]. It is safe to say that innovation in this field will continue and many of the techniques developed for HLS compilers, including the contributions of this thesis, will also make CGRA compilers work better on irregular codes.

7.4.4 The Inspector/Executor Approach

Finally, we would like to highlight a technique that can further increase the efficiency of accelerators for irregular code. In the Inspector/Executor approach, the original code is transformed to extract an inspector code section with the aim of executing it before the actual Executor computation to gather crucial parameters not available at compile time [228, 207]. We believe that such an approach would integrate well with the popular accelerator offload model where the host CPU sends data and invokes an accelerator kernel—the CPU could run the inspector code relatively cheaply, without having to provision area on the accelerator to implement the inspection. This would extend the close compiler-hardware co-design proposed in this thesis with additional co-operation with the runtime.

As a concrete example take the problem of dynamic memory disambiguation. In the inspector code on the CPU, we could gather the dependency distances and stream them to the FPGA accelerator. The FPGA could use the dependency distances to delay operations exactly the number of cycles needed for the dependency to be resolved. Such an approach would not require to use the LSQ, thus reducing circuit area and potentially increasing the frequency.

Bibliography

[1] *Advanced matrix extension (AMX) - x86*. URL: `https://en.wikichip.org/wiki/x86/amx`.

[2] Mythri Alle, Antoine Morvan and Steven 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. DOI: `10.1145/2463209.2488796`.

[3] J. R. Allen, Ken Kennedy, Carrie Porterfield and Joe Warren. “Conversion of control dependence to data dependence”. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’83. Austin, Texas: Association for Computing Machinery, 1983, pp. 177–189. ISBN: 0897910907. DOI: `10.1145/567067.567085`. URL: `https://doi.org/10.1145/567067.567085`.

[4] *AMD Xilinx Vitis*. URL: `https://www.xilinx.com/products/design-tools/vitis.html`.

[5] Zahira Ammarguellat and Williams Ludwell Harrison III. “Automatic recognition of induction variables and recurrence relations by abstract interpretation”. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, pp. 283–295.

[6] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito and Sridhar Ramaswamy. *NVIDIA Hopper Architecture In-Depth*. May 2022. URL: `https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`.

[7] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrián Cristal and Mikel Luján. “An empirical evaluation of high-level synthesis languages and tools for database acceleration”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pp. 1–8.

[8] ARM. *AXI protocol overview*. 2024. URL: `https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview`.

[9] José-María Arnau, Joan-Manuel Parcerisa and Polychronis Xekalakis. “Boosting mobile GPU performance with a decoupled access/execute fragment processor”. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA ’12. Portland, Oregon: IEEE Computer Society, 2012, pp. 84–93. ISBN: 9781450316422.

[10] Arvind and D E Culler. “Dataflow Architectures”. In: *Annual Review of Computer Science* (1986). DOI: `10.1146/annurev.cs.01.060186.001301`.

[11] K. Arvind and Rishiyur S. Nikhil. “Executing a Program on the MIT Tagged-Token Dataflow Architecture”. In: *IEEE Trans. Comput.* 39.3 (Mar. 1990), pp. 300–318. ISSN: 0018-9340. DOI: `10.1109/12.48862`. URL: `https://doi.org/10.1109/12.48862`.

[12] Mikhail Asiatici and Paolo Ienne. “Request, Coalesce, Serve, and Forget: Miss-Optimized Memory Systems for Bandwidth-Bound Cache-Unfriendly Applications on FPGAs”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.2 (Dec. 2021). ISSN: 1936-7406. DOI: 10.1145/3466823. URL: https://doi.org/10.1145/3466823.

[13] Mikhail Asiatici and Paolo Ienne. “Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 310–319. ISBN: 9781450361378. DOI: 10.1145/3289602.3293901. URL: https://doi.org/10.1145/3289602.3293901.

[14] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran and Wen-mei W. Hwu. “Integrated predicated and speculative execution in the IMPACT EPIC architecture”. In: *SIGARCH Comput. Archit. News* 26.3 (Apr. 1998), pp. 227–237. ISSN: 0163-5964. DOI: 10.1145/279361.279391. URL: https://doi.org/10.1145/279361.279391.

[15] Christiaan Baaij. “Cλash: From haskell to hardware”. MA thesis. University of Twente, 2009.

[16] Olaf Bachmann, Paul S. Wang and Eugene V. Zima. “Chains of Recurrencesa Method to Expedite the Evaluation of Closed-Form Functions”. In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. 1994. DOI: 10.1145/190347.190423.

[17] John Backus. “Can programming be liberated from the von Neumann style? a functional style and its algebra of programs”. In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: https://doi.org/10.1145/359576.359579.

[18] Helge Bahmann, Nico Reissmann, Magnus Jahre and Jan Christian Meyer. “Perfect Reconstructability of Control Flow from Demand Dependence Graphs”. In: *ACM Trans. Archit. Code Optim.* 11.4 (Jan. 2015). ISSN: 1544-3566. DOI: 10.1145/2693261. URL: https://doi.org/10.1145/2693261.

[19] Christel Baier and Joost-Pieter Katoen. “Principles of model checking”. In: 2008.

[20] L. Baugh and C. Zilles. “Decomposing the load-store queue by function for power reduction and scalability”. In: *IBM Journal of Research and Development* 50.2.3 (2006), pp. 287–297. DOI: 10.1147/rd.502.0287.

[21] Scott Beamer, Krste Asanovic and David A. Patterson. “The GAP Benchmark Suite”. In: *CoRR* abs/1508.03619 (2015). arXiv: 1508.03619. URL: http://arxiv.org/abs/1508.03619.

[22] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider and Torsten Hoefler. “Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019. DOI: 10.1145/3295500.3356173.

[23] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen and Cédric Bastoul. “The Polyhedral Model Is More Widely Applicable Than You Think”. In: *Compiler Construction*. Ed. by Rajiv Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 283–303. ISBN: 978-3-642-11970-5.

[24] Daniel Berlin and David Edelsohn. “High-level loop optimizations for GCC”. In: *Proceedings of the 2004 GCC Developers Summit* (Jan. 2004).

[25] Siddharth Bhat and Tobias Grosser. “Lambda the ultimate SSA: optimizing functional programs in SSA”. In: *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’22. Virtual Event, Republic of Korea: IEEE Press, 2022, pp. 168–178. ISBN: 9781665405843. DOI: 10.1109/CGO53902.2022.9741279. URL: https://doi.org/10.1109/CGO53902.2022.9741279.

[26] Peter L. Bird, Alasdair Rawsthorne and Nigel P. Topham. “The effectiveness of decoupling”. In: *Proceedings of the 7th International Conference on Supercomputing*. ICS ’93. Tokyo, Japan: Association for Computing Machinery, 1993, pp. 47–56. ISBN: 089791600X. DOI: 10.1145/165939.165952. URL: https://doi.org/10.1145/165939.165952.

[27] Per Bjesse, Koen Claessen, Mary Sheeran and Satnam Singh. “Lava: hardware design in Haskell”. In: *Acm Sigplan Notices* 34.1 (1998), pp. 174–184.

[28] W. Blume and R. Eigenmann. “An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks”. In: *1994 Internatonal Conference on Parallel Processing Vol. 2*. Vol. 2. 1994, pp. 233–238. DOI: 10.1109/ICPP.1994.59.

[29] Uday Bondhugula, Albert Hartono, J. Ramanujam and P. Sadayappan. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *SIGPLAN Not.* 43.6 (June 2008), pp. 101–113. ISSN: 0362-1340. DOI: 10.1145/1379022.1375595. URL: https://doi.org/10.1145/1379022.1375595.

[30] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala and Arvind. “The essence of Bluespec: a core language for rule-based hardware design”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 243–257.

[31] Stephen D. Brown, Robert J. Francis, Jonathan Rose and Zvonko G. Vranesic. *Field-programmable gate arrays*. USA: Kluwer Academic Publishers, 1992. ISBN: 0792392485.

[32] M. Budiu, P.V. Artigas and S.C. Goldstein. “Dataflow: A Complement to Superscalar”. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2005. DOI: 10.1109/ISPASS.2005.1430572.

[33] Mihai Budiu. “Spatial Computation”. PhD thesis. Carnegie Mellon University, Pittsburgh, PA, USA, 2003. URL: http://reports-archive.adm.cs.cmu.edu/anon/2003/abstracts/03-217.html.

[34] Mihai Budiu and Seth C. Goldstein. “Pegasus: An Efficient Intermediate Representation”. In: *Technical report, Carnegie Mellon University* (2003).

[35] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald and W. Yoder. “Scaling to the end of silicon with EDGE architectures”. In: *Computer* 37.7 (2004), pp. 44–55. DOI: 10.1109/MC.2004.65.

[36] Cadence. *Stratus High-Level Synthesis*. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html. [Accessed 08-01-2025]. 2025.

[37] Andrew Canis, Stephen D. Brown and Jason 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. DOI: 10.1109/FPL.2014.6927490.

[38] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown and Jason H. Anderson. “LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems”. In: *ACM Trans. Embed. Comput. Syst.* 13.2 (Sept. 2013). ISSN: 1539-9087. DOI: 10.1145/2514740. URL: https://doi.org/10.1145/2514740.

[39] L.P. Carloni, K.L. McMillan and A.L. Sangiovanni-Vincentelli. “Theory of latency-insensitive design”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2001). DOI: 10.1109/43.945302.

[40] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai and Zhiru Zhang. “Allo: A Programming Model for Composable Accelerator Design”. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656401. URL: https://doi.org/10.1145/3656401.

[41] Tao Chen and G Edward Suh. “Efficient data supply for hardware accelerators with prefetching and access/execute decoupling”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12.

[42] Tao Chen and G. Edward 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. DOI: 10.1109/MICRO.2016.7783749.

[43] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin and Arvind Krishnamurthy. “TVM: an automated end-to-end optimizing compiler for deep learning”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 9781931971478.

[44] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong and Deming Chen. “ThunderGP: HLS-based Graph Processing Framework on FPGAs”. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 69–80. ISBN: 9781450382182. DOI: 10.1145/3431920.3439290. URL: https://doi.org/10.1145/3431920.3439290.

[45] Jianyi Cheng. *JianyiCheng/HLS-benchmarks: HLS_Benchmarks_First_Release*. Version v1.0. Dec. 2019. DOI: `10.5281/zenodo.3561115`. URL: `https://doi.org/10.5281/zenodo.3561115`.

[46] Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason Anderson, John Wickerson and George A. Constantinides. "Efficient Memory Arbitration in High-Level Synthesis From Multi-Threaded Code". In: *IEEE Transactions on Computers* 71.4 (2022), pp. 933–946. DOI: `10.1109/TC.2021.3066466`.

[47] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne and John Wickerson. "DASS: Combining Dynamic amp; Static Scheduling in High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022). DOI: `10.1109/TCAD.2021.3065902`.

[48] Jianyi Cheng, Lana Josipovic, George A. Constantinides and John Wickerson. "Dynamic Inter-Block Scheduling for HLS". In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022, pp. 243–252. DOI: `10.1109/FPL57034.2022.00045`.

[49] Jianyi Cheng, John Wickerson and George 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. DOI: `10.1109/FPL53798.2021.00066`.

[50] Jianyi Cheng, John Wickerson and George 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*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 89–100. ISBN: 9781450391498. DOI: `10.1145/3490422.3502362`. URL: `https://doi.org/10.1145/3490422.3502362`.

[51] Shaoyi Cheng and John Wawrzynek. "Architectural synthesis of computational pipelines with decoupled memory access". In: *2014 International Conference on Field-Programmable Technology (FPT)*. 2014, pp. 83–90. DOI: `10.1109/FPT.2014.7082758`.

[52] Kazem Cheshmi, Michelle Strout and Maryam Mehri Dehnavi. "Runtime Composition of Iterations for Fusing Loop-carried Sparse Dependence". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '23. Denver, CO, USA: Association for Computing Machinery, 2023. ISBN: 9798400701092. DOI: `10.1145/3581784.3607097`. URL: `https://doi.org/10.1145/3581784.3607097`.

[53] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang and Jason Cong. "Extending High-Level Synthesis for Task-Parallel Programs". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021, pp. 204–213. DOI: `10.1109/FCCM51124.2021.00032`.

[54] Eric Chung et al. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38 (Mar. 2018), pp. 8–20. URL: `https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/`.

[55] *Configurable Spatial Accelerator (CSA) - Intel - WikiChip — en.wikichip.org.* https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator. [Accessed 12-12-2024].

[56] Jason Cong and Zhiru Zhang. "An efficient and versatile scheduling algorithm based on SDC formulation". In: *Proceedings of the 43rd Annual Design Automation Conference.* DAC '06. San Francisco, CA, USA: Association for Computing Machinery, 2006, pp. 433–438. ISBN: 1595933816. DOI: 10.1145/1146909.1147025. URL: https://doi.org/10.1145/1146909.1147025.

[57] J. Cortadella, M. Kishinevsky and B. Grundmann. "Synthesis of synchronous elastic architectures". In: *2006 43rd ACM/IEEE Design Automation Conference.* 2006. DOI: 10.1145/1146909.1147077.

[58] Philippe Coussy, Daniel D. Gajski, Michael Meredith and Andres Takach. "An Introduction to High-Level Synthesis". In: *IEEE Design and Test of Computers* 26.4 (2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[59] Neal C. Crago, Sana Damani, Karthikeyan Sankaralingam and Stephen W. Keckler. "WASP: Exploiting GPU Pipeline Parallelism with Hardware-Accelerated Automatic Warp Specialization". In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* 2024, pp. 1–16. DOI: 10.1109/HPCA57654.2024.00086.

[60] Neal Clayton Crago and Sanjay Jeram Patel. "OUTRIDER: efficient memory latency tolerance with decoupled strands". In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 117–128. ISSN: 0163-5964. DOI: 10.1145/2024723.2000079. URL: https://doi.org/10.1145/2024723.2000079.

[61] Steve Dai, Gai Liu, Ritchie Zhao and Zhiru Zhang. "Enabling adaptive loop pipelining in high-level synthesis". In: *2017 51st Asilomar Conference on Signals, Systems, and Computers.* 2017, pp. 131–135. DOI: 10.1109/ACSSC.2017.8335152.

[62] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten and Zhiru 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.* FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 189–194. ISBN: 9781450343541. DOI: 10.1145/3020078.3021754. URL: https://doi.org/10.1145/3020078.3021754.

[63] A. Darte. "On the complexity of loop fusion". In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425).* 1999, pp. 149–157. DOI: 10.1109/PACT.1999.807510.

[64] Nirav Dave, Michael Katelman, Myron King, Arvind and José Meseguer. "Verification of microarchitectural refinements in rule-based systems". In: *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011).* 2011, pp. 61–71. DOI: 10.1109/MEMCOD.2011.5970511.

[65] Jinyi Deng, Xinru Tang, Jiahao Zhang, Yuxuan Li, Linyun Zhang, Boxiao Han, Hongjun He, Fengbin Tu, Leibo Liu, Shaojun Wei, Yang Hu and Shouyi Yin. “Towards Efficient Control Flow Handling in Spatial Architecture via Architecting the Control Flow Plane”. In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 1395–1408. ISBN: 9798400703294. DOI: 10.1145/3613424.3614246. URL: https://doi.org/10.1145/3613424.3614246.

[66] Jack B. Dennis. “Data flow supercomputers”. In: *Computer* 11 (1980).

[67] Steven Derrien, Thibaut Marty, Simon Rokicki and Tomofumi Yuki. “Toward Speculative Loop Pipelining for High-Level Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 4229–4239. DOI: 10.1109/TCAD.2020.3012866.

[68] Florian Dewald, Johanna Rohde, Christian Hochberger and Heiko Mantel. “Improving Loop Parallelization by a Combination of Static and Dynamic Analyses in HLS”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.3 (Feb. 2022). ISSN: 1936-7406. DOI: 10.1145/3501801. URL: https://doi.org/10.1145/3501801.

[69] Edsger W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975.

[70] Chen Ding and Ken Kennedy. “Improving cache performance in dynamic applications through data and computation reorganization at run time”. In: *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. PLDI ’99. Atlanta, Georgia, USA: Association for Computing Machinery, 1999, pp. 229–241. ISBN: 1581130945. DOI: 10.1145/301618.301670. URL: https://doi.org/10.1145/301618.301670.

[71] Andrew Duller, Gajinder Panesar and Daniel Towner. “Parallel Processing the picoChip way!” In: *Parallel Processing* (Jan. 2003), p. 15. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.170.9443&rep=rep1&type=pdf.

[72] J. Duprat and J. -M. Muller. “The CORDIC Algorithm: New Results for Fast VLSI Implementation”. In: *IEEE Trans. Comput.* 42.2 (Feb. 1993), pp. 168–178. ISSN: 0018-9340. DOI: 10.1109/12.204786. URL: https://doi.org/10.1109/12.204786.

[73] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian and Pat Hanrahan. “Type-directed scheduling of streaming accelerators”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 408–422. ISBN: 9781450376136. DOI: 10.1145/3385412.3385983. URL: https://doi.org/10.1145/3385412.3385983.

[74] Stephen A. Edwards, Richard Townsend, Martha Barker and Martha A. Kim. “Compositional Dataflow Circuits”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 18 (2019).

[75] *Efficient Computer — efficient.computer*. https://www.efficient.computer/. [Accessed 12-12-2024].

[76] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipovic and Paolo Ienne. "Unleashing Parallelism in Elastic Circuits with Faster Token Delivery". In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022, pp. 253–261. DOI: 10.1109/FPL57034.2022.00046.

[77] Ayatallah Elakhras, Riya Sawhney, Andrea Guerrieri, Lana Josipovic and Paolo 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*. FPGA '23. Monterey, CA, USA: Association for Computing Machinery, 2023, pp. 39–45. ISBN: 9781450394178. DOI: 10.1145/3543622.3573050. URL: https://doi.org/10.1145/3543622.3573050.

[78] Ahmed ElTantawy and Tor M. Aamodt. "MIMD synchronization on SIMT architectures". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016.

[79] Robert A. van Engelen. "Efficient Symbolic Analysis for Optimizing Compilers". In: *Compiler Construction*. Ed. by Reinhard Wilhelm. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 118–132. ISBN: 978-3-540-45306-2.

[80] Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh and Kyle A. Gallivan. "A unified framework for nonlinear dependence testing and symbolic analysis". In: *Proceedings of the 18th Annual International Conference on Supercomputing*. ICS '04. Malo, France: Association for Computing Machinery, 2004, pp. 106–115. ISBN: 1581138393. DOI: 10.1145/1006209.1006226. URL: https://doi.org/10.1145/1006209.1006226.

[81] EPFL. *Dynamatic — dynamatic.epfl.ch*. https://dynamatic.epfl.ch/. [Accessed 11-12-2024].

[82] Gregory Faanes and Eric Lundberg. *Vector and scalar data cache for a vector multiprocessor*. U.S. Patent US20020144061A1.

[83] Zhihua Fan, Wenming Li, Shengzhong Tang, Xuejun An, Xiaochun Ye and Dongrui Fan. "Improving utilization of dataflow architectures through software and hardware co-design". In: *European Conference on Parallel Processing*. Springer. 2023, pp. 245–259.

[84] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. "The program dependence graph and its use in optimization". In: *TOPL*. 1987.

[85] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer and Torsten Hoefler. "StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems". In: *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 2021, pp. 315–326. ISBN: 9781728186139. URL: https://doi.org/10.1109/CGO51591.2021.9370315.

[86] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg and Alexandru Nicolau. “Parallel Processing: A Smart Compiler and a Dumb Machine”. In: SIGPLAN ’84. 1984. DOI: 10.1145/502874.502878.

[87] Kermin Elliott Fleming. “Scalable reconfigurable computing leveraging latency-insensitive channels”. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 2013. URL: https://hdl.handle.net/1721.1/79212.

[88] Kermin Elliott Fleming. “Scalable reconfigurable computing leveraging latency-insensitive channels”. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 2013. URL: https://hdl.handle.net/1721.1/79212.

[89] Shane T. Fleming and David B. Thomas. “Using Runahead Execution to Hide Memory Latency in High Level Synthesis”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017, pp. 109–116. DOI: 10.1109/FCCM.2017.33.

[90] Tom Forsyth. *SMACNI to AVX512 the life cycle of an instruction set*. https://tomforsyth1000 github.io/larrabee/LRBNI%20origins%20v4%20full%20fat.pdf. [Accessed 13-12-2024]. Handmade Seattle 2019.

[91] LLVM Foundation. *CIRCT*. https://circt.llvm.org/. [Accessed 11-12-2024].

[92] LLVM Foundation. *LLVM Loop Terminology (and Canonical Forms)*. https://llvm.org/docs/LoopTerminology.html. [Accessed 09-12-2024].

[93] Hagen Gädke and Andreas Koch. “Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens”. In: *Reconfigurable Computing: Architectures, Tools and Applications*. Ed. by Roger Woods, Katherine Compton, Christos Bouganis and Pedro C. Diniz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 185–195. ISBN: 978-3-540-78610-8.

[94] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar and Trevor Bauer. “Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 84–93. ISBN: 9781450361378. DOI: 10.1145/3289602.3293906. URL: https://doi.org/10.1145/3289602.3293906.

[95] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm and Kevin Skadron. “Energy-efficient mechanisms for managing thread context in throughput processors”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 235–246. DOI: 10.1145/2000064.2000093.

[96] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler and Olivier Temam. “Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies”. In: *International Journal of Parallel Programming* 34 (2006), pp. 261–317.

[97] Jean-Michel Gorius, Simon Rokicki and Steven Derrien. “A Unified Memory Dependency Framework for Speculative High-Level Synthesis”. In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. Edinburgh, United Kingdom: Association for Computing Machinery, 2024, pp. 13–25. ISBN: 9798400705076. DOI: 10.1145/3640537.3641581. URL: https://doi.org/10.1145/3640537.3641581.

[98] Tobias Grosser, Armin Groesslinger and Christian Lengauer. “Polly Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Processing Letters* 22.04 (2012), p. 1250010. DOI: 10.1142/S0129626412500107.

[99] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang and Jason Cong. “AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs”. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2021, pp. 81–92.

[100] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang and Jason Cong. “RapidStream: Parallel Physical Implementation of FPGA HLS Designs”. In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 1–12. ISBN: 9781450391498. DOI: 10.1145/3490422.3502361. URL: https://doi.org/10.1145/3490422.3502361.

[101] Rajiv Gupta and Madalene Spezialetti. “Loop monotonic computations: an approach for the efficient run-time detection of races”. In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. TAV4. Victoria, British Columbia, Canada: Association for Computing Machinery, 1991, pp. 98–111. ISBN: 089791449X. DOI: 10.1145/120807.120816. URL: https://doi.org/10.1145/120807.120816.

[102] J. R Gurd, C. C Kirkham and I. Watson. “The Manchester prototype dataflow computer”. In: *Commun. ACM* 28.1 (Jan. 1985), pp. 34–52. ISSN: 0001-0782. DOI: 10.1145/2465.2468. URL: https://doi.org/10.1145/2465.2468.

[103] Tae Jun Ham, Juan L. Aragón and Margaret Martonosi. “Decoupling Data Supply from Computation for Latency-Tolerant Communication in Heterogeneous Architectures”. In: *ACM Trans. Archit. Code Optim.* 14.2 (June 2018). ISSN: 1544-3566. DOI: 10.1145/3075620. URL: https://doi.org/10.1145/3075620.

[104] Tae Jun Ham, Juan L. Aragón and Margaret Martonosi. “DeSC: decoupled supply-compute communication management for heterogeneous architectures”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 191–203. ISBN: 9781450340342. DOI: 10.1145/2830772.2830800. URL: https://doi.org/10.1145/2830772.2830800.

[105] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 9780080556017.

[106] Paul Havlak. “Nesting of reducible and irreducible loops”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.4 (1997), pp. 557–567.

[107] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017.

[108] John L. Hennessy and David A. Patterson. “A New Golden Age for Computer Architecture”. In: *Commun. ACM* (2019). DOI: `10.1145/3282307`.

[109] Yann Herklotz, Delphine Demange and Sandrine Blazy. “Mechanised Semantics for Gated Static Single Assignment”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 182–196. ISBN: 9798400700262. DOI: `10.1145/3573105.3575681`. URL: `https://doi.org/10.1145/3573105.3575681`.

[110] C. A. R. Hoare. “Communicating sequential processes”. In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: `10.1145/359576.359585`. URL: `https://doi.org/10.1145/359576.359585`.

[111] Tu Hong, Ning Guan, Chen Yin, Qin Wang, Jianfei Jiang, Jing Jin, Guanghui He and Naifeng Jing. “Decoupling the multi-rate dataflow execution in coarse-grained reconfigurable array”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020.

[112] Jing Huang, Yuanjie Huang, Olivier Temam, Paolo Ienne, Yunji Chen and Chengyong Wu. “A low-cost memory interface for high-throughput accelerators”. In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES ’14. New Delhi, India: Association for Computing Machinery, 2014. ISBN: 9781450330503. DOI: `10.1145/2656106.2656109`. URL: `https://doi.org/10.1145/2656106.2656109`.

[113] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen and Chengyong Wu. “Elastic CGRAs”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’13. 2013. DOI: `10.1145/2435264.2435296`.

[114] Mike Hutton, Jay Schleicher, David Lewis, Bruce Pedersen, Richard Yuan, Sinan Kaptanoglu, Gregg Baeckler, Boris Ratchev, Ketan Padalia, Mark Bourgeault, Andy Lee, Henry Kim and Rahul Saini. “Improving FPGA Performance and Area Using an Adaptive Logic Module”. In: *Field Programmable Logic and Application*. 2004.

[115] Mike Hutton, Jay Schleicher, David Lewis, Bruce Pedersen, Richard Yuan, Sinan Kaptanoglu, Gregg Baeckler, Boris Ratchev, Ketan Padalia, Mark Bourgeault, Andy Lee, Henry Kim and Rahul Saini. “Improving FPGA Performance and Area Using an Adaptive Logic Module”. In: *Field Programmable Logic and Application*. Ed. by Jürgen Becker, Marco Platzner and Serge Vernalde. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 135–144. ISBN: 978-3-540-30117-2.

[116] C-T Hwang, J-H Lee and Y-C Hsu. “A formal approach to the scheduling problem in high level synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.4 (1991), pp. 464–475.

[117] Intel. *Intel C++ Compiler Handbook for Intel FPGAs*. https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/developer-guide/2024-1/intel-oneapi-dpc-c-compiler-handbook-for-intel.html. Accessed: 2024-08-02. 2024.

[118] Intel. *Intel/LLVM*. URL: https://github.com/intel/llvm/tree/sycl.

[119] Intel. *Intelő PAC with Intelő Arriaő 10 GX FPGA - Product Specifications | Intel — intel.com*. https://www.intel.com/content/www/us/en/products/sku/149169/intel-pac-with-intel-arria-10-gx-fpga/specifications.html. [Accessed 18-09-2024].

[120] *Intel HLS*. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[121] Abhishek Jain, Chirag Ravishankar, Hossein Omidian, Sharan Kumar, Maithilee Kulkarni, Aashish Tripathi and Dinesh Gaitonde. "Modular and Lean Architecture with Elasticity for Sparse Matrix Vector Multiplication on FPGAs". In: *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines*. 2023.

[122] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim and Murali Annavaram. "GPU register file virtualization". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 420–432. ISBN: 9781450340342. DOI: 10.1145/2830772.2830784. URL: https://doi.org/10.1145/2830772.2830784.

[123] Wesley M Johnston, JR Paul Hanna and Richard J Millar. "Advances in dataflow programming languages". In: *ACM computing surveys (CSUR)* 36.1 (2004), pp. 1–34.

[124] Lana Josipovic, Atri Bhattacharyya, Andrea Guerrieri and Paolo Ienne. "Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs". In: *2019 International Conference on Field-Programmable Technology*. 2019. DOI: 10.1109/ICFPT47387.2019.00031.

[125] Lana Josipovic, Philip Brisk and Paolo Ienne. "An Out-of-Order Load-Store Queue for Spatial Computing". In: *ACM Transactions on Embedded Computing Systems* (2017). DOI: 10.1145/3126525.

[126] Lana Josipovic, Andrea Guerrieri and Paolo Ienne. "From C/C++ Code to High-Performance Dataflow Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022). DOI: 10.1109/TCAD.2021.3105574.

[127] Lana Josipovic, Andrea Guerrieri and Paolo Ienne. "Speculative Dataflow Circuits". en. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmabl Gate Arrays*. ACM, Feb. 2019, pp. 162–171. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293914.

[128] Lana Josipovic, Axel Marmet, Andrea Guerrieri and Paolo Ienne. "Resource Sharing in Dataflow Circuits". In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022, pp. 1–9. DOI: 10.1109/FCCM53951.2022.9786084.

[129] Timothy Kam, Michael Kishinevsky, Jordi Cortadella and Marc Galceran-Oms. “Correct-by-construction microarchitectural pipelining”. In: *2008 IEEE/ACM International Conference on Computer-Aided Design*. 2008, pp. 434–441. DOI: 10.1109/ICCAD.2008.4681612.

[130] Ryan Kastner, Janarbek Matai and Stephen Neuendorffer. “Parallel Programming for FPGAs”. In: *CoRR* abs/1805.03648 (2018). arXiv: 1805.03648. URL: http://arxiv.org/abs/1805.03648.

[131] Richard A Kelsey. “A correspondence between continuation passing style and static single assignment form”. In: *ACM SIGPLAN Notices* 30.3 (1995), pp. 13–22.

[132] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558602860.

[133] *Khronos Sycl Registry*. URL: https://registry.khronos.org/SYCL/.

[134] Jan Willem Klop and JW Klop. *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.

[135] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis et al. “Spatial: A language and compiler for application accelerators”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 296–311.

[136] Martin Kristien, Bruno Bodin, Michel Steuwer and Christophe Dubach. “High-level synthesis of functional patterns with Lift”. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 35–45. ISBN: 9781450367172. DOI: 10.1145/3315454.3329957. URL: https://doi.org/10.1145/3315454.3329957.

[137] D.C. Ku and G. De Mitcheli. “Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.6 (1992), pp. 696–718. DOI: 10.1109/43.137516.

[138] Milind Kulkarni, Martin Burtscher, Calin Cascaval and Keshav Pingali. “Lonestar: A suite of parallel irregular programs”. In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. DOI: 10.1109/ISPASS.2009.4919639.

[139] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong and Zhiru Zhang. “HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 242–251. ISBN: 9781450361378. DOI: 10.1145/3289602.3293910.

[140] M. Lam. “Software pipelining: an effective scheduling technique for VLIW machines”. In: *SIGPLAN Not.* 23.7 (June 1988), pp. 318–328. ISSN: 0362-1340. DOI: 10.1145/960116.54022. URL: https://doi.org/10.1145/960116.54022.

[141] Leslie Lamport. “Specifying systems: the TLA+ language and tools for hardware and software engineers”. In: (2002).

[142] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: *CGO*. 2004.

[143] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache and Oleksandr Zinenko. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[144] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro and Murali Annavaram. “Warped-compression: enabling power efficient GPUs through register compression”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA ’15. Portland, Oregon: Association for Computing Machinery, 2015, pp. 502–514. ISBN: 9781450334020. DOI: 10.1145/2749469.2750417. URL: https://doi.org/10.1145/2749469.2750417.

[145] Roland LeiSSa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller and Bertil Schmidt. “AnyDSL: a partial evaluation framework for programming high-performance libraries”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276489. URL: https://doi.org/10.1145/3276489.

[146] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[147] Richard A Lethin. “How vliw almost disappeared-and then proliferated”. In: *IEEE Solid-State Circuits Magazine* 1.3 (2009), pp. 15–23.

[148] Adam Levinthal and Thomas Porter. “Chap - a SIMD graphics processor”. In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 77–82. ISBN: 0897911385. DOI: 10.1145/800031.808581. URL: https://doi.org/10.1145/800031.808581.

[149] Yuan Lin and David Padua. “Compiler analysis of irregular memory accesses”. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI ’00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 157–168. ISBN: 1581131992. DOI: 10.1145/349299.349322. URL: https://doi.org/10.1145/349299.349322.

[150] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson and David I. August. “CGPA: Coarse-Grained Pipelined Accelerators”. In: *Proceedings of the 51st Annual Design Automation Conference*. DAC ’14. San Francisco, CA, USA: Association for Computing Machinery, 2014, pp. 1–6. ISBN: 9781450327305. DOI: 10.1145/2593069.2593105. URL: https://doi.org/10.1145/2593069.2593105.

[151] Jiantao Liu, Carmine Rizzi and Lana Josipovic. “Load-Store Queue Sizing for Efficient Dataflow Circuits”. In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022, pp. 1–9. DOI: 10.1109/ICFPT56656.2022.9974425.

[152] Junyi Liu, Samuel Bayliss and George A. Constantinides. “Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS”. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 159–162. DOI: `10.1109/FCCM.2015.31`.

[153] Junyi Liu, John Wickerson, Samuel Bayliss and George A. Constantinides. “Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.9 (2018), pp. 1802–1815. DOI: `10.1109/TCAD.2017.2783363`.

[154] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau and Michael S. Schlansker. “Sentinel scheduling for VLIW and superscalar processors”. In: *SIGPLAN Not.* 27.9 (Sept. 1992), pp. 238–247. ISSN: 0362-1340. DOI: `10.1145/143371.143529`. URL: `https://doi.org/10.1145/143371.143529`.

[155] Grant Martin and Gary Smith. “High-Level Synthesis: Past, Present, and Future”. In: *IEEE Design and Test of Computers* 26.4 (2009), pp. 18–25. DOI: `10.1109/MDT.2009.83`.

[156] Kevin J. M. Martin. “Twenty Years of Automated Methods for Mapping Applications on CGRA”. In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 679–686. DOI: `10.1109/IPDPSW55747.2022.00118`.

[157] John McCalpin. “Memory bandwidth and machine balance in high performance computers”. In: *IEEE Technical Committee on Computer Architecture Newsletter* (Dec. 1995), pp. 19–25.

[158] Michael C McFarland, Alice C Parker and Raul Camposano. “The high-level synthesis of digital systems”. In: *Proceedings of the IEEE* 78.2 (1990), pp. 301–318.

[159] Oskar Mencer, Dennis Allison, Elad Blatt, Mark Cummings, Michael J Flynn, Jerry Harris, Carl Hewitt, Quinn Jacobson, Maysam Lavasani, Mohsen Moazami et al. “The History, Status, and Future of FPGAs: Hitting a nerve with field-programmable gate arrays”. In: *Queue* 18.3 (2020), pp. 71–82.

[160] N. Mitchell, L. Carter and J. Ferrante. “Localizing non-affine array references”. In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*. 1999, pp. 192–202. DOI: `10.1109/PACT.1999.807526`.

[161] Antoine Morvan, Steven Derrien and Patrice Quinton. “Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion”. In: *2011 International Conference on Field-Programmable Technology*. 2011. DOI: `10.1109/FPT.2011.6132715`.

[162] O. Mutlu, J. Stark, C. Wilkerson and Y.N. Patt. “Runahead execution: an alternative to very large instruction windows for out-of-order processors”. In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 2003, pp. 129–140. DOI: `10.1109/HPCA.2003.1183532`.

[163] Ajeya Naithani, Jaime Roelandts, Sam Ainsworth, Timothy M. Jones and Lieven Eeckhout. “Decoupled Vector Runahead”. In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 17–31. ISBN: 9798400703294. DOI: 10.1145/3613424.3614255. URL: https://doi.org/10.1145/3613424.3614255.

[164] Matthew Naylor and Simon Moore. “A generic synthesisable test bench”. In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2015, pp. 128–137. DOI: 10.1109/MEMCOD.2015.7340479.

[165] *NEXTSILICON — nextsilicon.com*. https://www.nextsilicon.com/. [Accessed 12-12-2024].

[166] Quan M. Nguyen and Daniel Sanchez. “Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1064–1077. ISBN: 9781450385572. DOI: 10.1145/3466752.3480048. URL: https://doi.org/10.1145/3466752.3480048.

[167] Quan M. Nguyen and Daniel Sanchez. “Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2023, pp. 1262–1274. DOI: 10.1109/HPCA56546.2023.10071026.

[168] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson and Zhiru Zhang. “Predictable accelerator design with time-sensitive affine types”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 393–407. ISBN: 9781450376136. DOI: 10.1145/3385412.3385974.

[169] R. Nikhil. “Bluespec System Verilog: efficient, correct RTL from high level specifications”. In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE ’04.* 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.

[170] Rishiyur S. Nikhil. “Abstraction in Hardware System Design: Applying lessons from software languages to hardware languages using Bluespec SystemVerilog”. In: *Queue* 9.8 (Aug. 2011), pp. 40–54. ISSN: 1542-7730. DOI: 10.1145/2016036.2020861.

[171] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani and Karthikeyan Sankaralingam. “Stream-dataflow acceleration”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 416–429.

[172] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. [Accessed 11-12-2024]. 2020.

[173] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann and Oliver Sinnen. “ILP-Based modulo Scheduling for High-Level Synthesis”. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES ’16. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2016. ISBN: 9781450344821. DOI: `10.1145/2968455.2968512`. URL: `https://doi.org/10.1145/2968455.2968512`.

[174] Karl J. Ottenstein, Robert A. Ballance and Arthur B. Maccabe. “The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages”. In: *PLDI ’90*.

[175] G. Ottoni, R. Rangan, A. Stoler and D.I. August. “Automatic thread extraction with decoupled software pipelining”. In: *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’05)*. 2005, 12 pp.–118. DOI: `10.1109/MICRO.2005.13`.

[176] M. Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland LeiSSa, Sebastian Hack, Jürgen Teich and Frank Hannig. “AnyHLS: High-Level Synthesis With Partial Evaluation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3202–3214. DOI: `10.1109/TCAD.2020.3012172`.

[177] Gregory M Papadopoulos and David E Culler. “Monsoon: an explicit token-store architecture”. In: *ACM SIGARCH Computer Architecture News* 18.2SI (1990), pp. 82–91.

[178] Joseph CH Park and Mike Schlansker. *On predicated execution*. Hewlett-Packard Laboratories Palo Alto, California, 1991.

[179] Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan, Randy Allmon, Neal Crago, Kermin Fleming, Mohit Gambhir, Aamer Jaleel, Tushar Krishna, Daniel Lustig, Stephen Maresh, Vladimir Pavlov, Rachid Rayess, Antonia Zhai and Joel Emer. “Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures”. In: *ACM Transactions on Computer Systems* (2015). DOI: `10.1145/2754930`.

[180] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher and Joel Emer. “Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 137–151.

[181] Blake Pelton, Adam Sapek, Ken Eguro, Daniel Lo, Alessandro Forin, Matt Humphrey, Jinwen Xi, David Cox, Rajas Karandikar, Johannes de Fine Licht, Evgeny Babin, Adrian Caulfield and Doug Burger. “Wavefront Threading Enables Effective High-Level Synthesis”. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: `10.1145/3656420`.

[182] W. W. Peterson, T. Kasami and N. Tokura. “On the capabilities of while, repeat, and exit statements”. In: *Commun. ACM* 16.8 (Aug. 1973), pp. 503–512. ISSN: 0001-0782. DOI: `10.1145/355609.362337`. URL: `https://doi.org/10.1145/355609.362337`.

[183] Sebastian Pop, Philippe Clauss, Albert Cohen, Vincent Loechner and Georges-André Silber. "Fast recognition of scalar evolutions on three-address ssa code". In: *CRI/ENSMP Research Report, A/354/CRI* (2004), pp. 1–28.

[184] Sebastian Pop, Albert Cohen and Georges-André Silber. "Induction Variable Analysis with Delayed Abstractions". In: *High Performance Embedded Architectures and Compilers*. Ed. by Tom Conte, Nacho Navarro, Wen-mei W. Hwu, Mateo Valero and Theo Ungerer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 218–232. ISBN: 978-3-540-32272-6.

[185] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan and Jason Cong. "Polyhedral-based data reuse optimization for configurable computing". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA: Association for Computing Machinery, 2013, pp. 29–38. ISBN: 9781450318877. DOI: 10.1145/2435264.2435273. URL: https://doi.org/10.1145/2435264.2435273.

[186] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan and Nicolas Vasilache. "Loop transformations: convexity, pruning and optimization". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 549–562. ISBN: 9781450304900. DOI: 10.1145/1926385.1926449. URL: https://doi.org/10.1145/1926385.1926449.

[187] Raghu Prabhakar and Sumti Jairath. "SambaNova SN10 RDU:Accelerating Software 2.0 with Dataflow". In: *2021 IEEE Hot Chips 33 Symposium (HCS)*. 2021, pp. 1–37. DOI: 10.1109/HCS52781.2021.9567250.

[188] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis and Kunle Olukotun. "Plasticine: A Reconfigurable Architecture For Parallel Paterns". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 389–402. ISBN: 9781450348928. DOI: 10.1145/3079856.3080256. URL: https://doi.org/10.1145/3079856.3080256.

[189] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: 42.3 (2014). DOI: 10.1145/2678373.2665678.

[190] Shantian Qin, Wenming Li, Zhihua Fan, Zhen Wang, Tianyu Liu, Haibin Wu, Kunming Zhang, Xuejun An, Xiaochun Ye and Dongrui Fan. "ROMA: A Reconfigurable On-chip Memory Architecture for Multi-core Accelerators". In: *2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application*. IEEE. 2023, pp. 49–57.

[191] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *SIGPLAN Not.* (2013). DOI: 10.1145/2499370.2462176.

[192] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan et al. “Warehouse-scale video acceleration: co-design and deployment in the wild”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 600–615.

[193] Parthasarathy Ranganathan et al. “Warehouse-scale video acceleration: co-design and deployment in the wild”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’21. Virtual, USA: Association for Computing Machinery, 2021, pp. 600–615. ISBN: 9781450383172. DOI: 10.1145/3445814.3446723. URL: https://doi.org/10.1145/3445814.3446723.

[194] Fabrice Rastello. *SSA-based Compiler Design*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 1441962018.

[195] B Ramakrishna Rau and Joseph A Fisher. “Instruction-level parallel processing: History, overview, and perspective”. In: *The journal of Supercomputing* 7 (1993), pp. 9–50.

[196] B. Ramakrishna Rau. “Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops”. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. 1994. DOI: 10.1145/192724.192731.

[197] Mahesh Ravishankar, John Eisenlohr, Louis-Noel Pouchet, J. Ramanujam, Atanas Rountev and P. Sadayappan. “Code generation for parallel execution of a class of irregular loops on distributed memory systems”. In: *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.30.

[198] Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. “Global value numbers and redundant computations”. In: *ACM-SIGACT Symposium on Principles of Programming Languages*. 1988.

[199] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou and Jason Cong. “ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 9–16. DOI: 10.1109/FCCM.2018.00011.

[200] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar and Kunle Olukotun. “Capstan: A Vector RDA for Sparsity”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1022–1035. ISBN: 9781450385572. DOI: 10.1145/3466752.3480047. URL: https://doi.org/10.1145/3466752.3480047.

[201] J.H. Saltz, R. Mirchandaney and K. Crowley. “Run-time parallelization and scheduling of loops”. In: *IEEE Transactions on Computers* 40.5 (1991), pp. 603–612. DOI: 10.1109/12.88484.

[202] Christof Schlaak, Tzung-Han Juang and Christophe Dubach. “Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators”. In: *ACM Trans. Archit. Code Optim.* 19.2 (Jan. 2022). ISSN: 1544-3566. DOI: 10.1145/3501768. URL: https://doi.org/10.1145/3501768.

[203] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019. URL: https://github.com/schoeberl/chisel-book.

[204] Fabian Schuiki, Andreas Kurth, Tobias Grosser and Luca Benini. “LLHD: a multi-level intermediate representation for hardware description languages”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 258–271. ISBN: 9781450376136. DOI: 10.1145/3385412.3386024. URL: https://doi.org/10.1145/3385412.3386024.

[205] James E. Smith. “Decoupled Access/Execute Computer Architectures”. In: *Proceedings of the 9th Annual Symposium on Computer Architecture*. ISCA ’82. Austin, Texas, USA: IEEE Computer Society Press, 1982, pp. 112–119.

[206] Michel Steuwer, Thomas Koehler, Bastian Köpcke and Federico Pizzuti. “RISE & shine: Language-oriented compiler design”. In: *arXiv preprint arXiv:2201.03611* (2022).

[207] Michelle Mills Strout, Mary Hall and Catherine Olschanowsky. “The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code”. In: *Proceedings of the IEEE* 106.11 (2018), pp. 1921–1934. DOI: 10.1109/JPROC.2018.2857721.

[208] Steven Swanson, Ken Michelson, Andrew Schwerin and Mark Oskin. “WaveScalar”. In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE. 2003, pp. 291–302.

[209] Robert Szafarczyk. *Compiler Support for Speculation in Decoupled Access/Execute Architectures - code & evaluation*. Nov. 2024. DOI: 10.5281/zenodo.14051996. URL: https://doi.org/10.5281/zenodo.14051996.

[210] Robert Szafarczyk. *Dynamic Loop Fusion in HLS - FPGA25 artifact*. Oct. 2024. DOI: 10.5281/zenodo.13898002. URL: https://doi.org/10.5281/zenodo.13898002.

[211] Robert Szafarczyk. *robertszafa/elastic-sycl-hls: Artifact FPT23*. 2023. DOI: 10.5281/zenodo.10124899. URL: https://doi.org/10.5281/zenodo.10124899.

[212] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “A High-Frequency Load-Store Queue with Speculative Allocations for High-Level Synthesis”. In: *2023 International Conference on Field Programmable Technology (ICFPT)*. 2023, pp. 115–124. DOI: 10.1109/ICFPT59805.2023.00018.

[213] 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)*. 2023, pp. 1–9. DOI: 10.1109/FPL60245.2023.00009.

[214] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis”. In: *2023 33nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2023.

[215] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Compiler Support for Speculation in Decoupled Access/Execute Architectures”. In: *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. CC ’25. Las Vegas, United States, 2025. DOI: 10.1145/3708493.3712695.

[216] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Dynamic Loop Fusion in High-Level Synthesis”. In: *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’25. Monterey, CA, USA: Association for Computing Machinery, 2025. DOI: 10.1145/3706628.3708871.

[217] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Dynamically Scheduled Memory Operations in Static High-Level Synthesis”. In: *IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2023, pp. 220–220. DOI: 10.1109/FCCM57271.2023.00048.

[218] Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede. “Reducing FPGA Memory Footprint of Stencil Codes through Automatic Extraction of Memory Patterns”. In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 2022, pp. 148–152. DOI: 10.1109/FPL57034.2022.00033.

[219] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai and Zhiru 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. DOI: 10.1109/ICCAD.2015.7372553.

[220] Benjamin Thielmann, Jens Huthmann and Andreas Koch. “Memory Latency Hiding by Load Value Speculation for Reconfigurable Computers”. In: *ACM Trans. Reconfigurable Technol. Syst.* (2012). DOI: 10.1145/2362374.2362377.

[221] James Thomas, Pat Hanrahan and Matei Zaharia. “Fleet: A Framework for Massively Parallel Streaming on FPGAs”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 639–651. ISBN: 9781450371025. DOI: 10.1145/3373376.3378495. URL: https://doi.org/10.1145/3373376.3378495.

[222] Ettore Tiotto, Víctor Pérez, Whitney Tsang, Lukas Sommer, Julian Oppermann, Victor Lomüller, Mehdi Goli and James Brodman. “Experiences Building an MLIR-Based SYCL Compiler”. In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2024, pp. 399–410. DOI: 10.1109/CGO57630.2024.10444866.

[223] N. Topham, A. Rawsthorne, C. McLean, M. Mewissen and P. Bird. “Compiling and Optimizing for Decoupled Architectures”. In: *Supercomputing ’95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. 1995, pp. 40–40. DOI: 10.1145/224170.224301.

[224] Richard Townsend, Martha A. Kim and Stephen A. Edwards. “From functional programs to pipelined dataflow circuits”. In: *Proceedings of the 26th International Conference on Compiler Construction*. 2017. DOI: 10.1145/3033019.3033027.

[225] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni and David I. August. “Speculative Decoupled Software Pipelining”. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 2007, pp. 49–59. DOI: 10.1109/PACT.2007.4336199.

[226] Wim Vanderbauwhede, Syed Waqar Nabi and Cristian Urlea. “Type-Driven Automated Program Transformations and Cost Modelling for Optimising Streaming Programs on FPGAs”. In: *International Journal of Parallel Programming* 47 (Feb. 2019). DOI: 10.1007/s10766-018-0572-z.

[227] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout and Mary Hall. “Automating Wavefront Parallelization for Sparse Matrix Computations”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 480–491. DOI: 10.1109/SC.2016.40.

[228] Anand Venkat, Manu Shantharam, Mary Hall and Michelle Mills Strout. “Non-affine Extensions to Polyhedral Code Generation”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’14. Orlando, FL, USA: Association for Computing Machinery, 2018, pp. 185–194. ISBN: 9781450326704. DOI: 10.1145/2544137.2544141. URL: https://doi.org/10.1145/2544137.2544141.

[229] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea and Seth C. Goldstein. “C to Asynchronous Dataflow Circuits: An End-to-End Toolflow”. In: *IWLS’01* (2001). DOI: 10.1184/R1/6603986.v1.

[230] Matthew Vilim, Alexander Rucker and Kunle Olukotun. “Aurochs: An Architecture for Dataflow Threads”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 402–415. DOI: 10.1109/ISCA52012.2021.00039.

[231] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah and Tony Nowatzki. “DSAGEN: Synthesizing Programmable Spatial Accelerators”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 268–281. DOI: 10.1109/ISCA45697.2020.00032.

[232] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu and Tony Nowatzki. “A hybrid systolic-dataflow architecture for inductive matrix algorithms”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 703–716.

[233] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania and Tulika Mitra. “CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA”. In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). ISSN: 1539-9087. DOI: 10.1145/3358177. URL: https://doi.org/10.1145/3358177.

[234] Wikipedia. *Itanium — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Itanium&oldid=1261228610. [Online; accessed 11-December-2024]. 2024.

[235] Felix Winterstein and George Constantinides. "Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs". In: *2017 International Conference on Field Programmable Technology (ICFPT)*. 2017, pp. 104–111. DOI: `10.1109/FPT.2017.8280127`.

[236] Henry Wong, Vaughn Betz and Jonathan 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. DOI: `10.1109/FPT.2013.6718409`.

[237] Peng Wu, Albert Cohen, Jay Hoeflinger and David Padua. "Monotonic evolution: an alternative to induction variable substitution for dependence analysis". In: *Proceedings of the 15th International Conference on Supercomputing*. ICS '01. Sorrento, Italy: Association for Computing Machinery, 2001, pp. 78–91. ISBN: 158113410X. DOI: `10.1145/377792.377809`. URL: `https://doi.org/10.1145/377792.377809`.

[238] *Xilinx Vivado HLS*. URL: `https://www.xilinx.com/products/design-tools/vivado/high-level-design.html`.

[239] Jiahui Xu, Emmet Murphy, Jordi Cortadella and Lana Josipovic. "Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking". In: *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '23. Monterey, CA, USA: Association for Computing Machinery, 2023, pp. 27–37. ISBN: 9781450394178. DOI: `10.1145/3543622.3573196`. URL: `https://doi.org/10.1145/3543622.3573196`.

[240] Shaoxian Xu, Sitong Lu, Zhiyuan Shao, Xiaofei Liao and Hai Jin. "MiCache: An MSHR-inclusive Non-blocking Cache Design for FPGAs". In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 22–32. ISBN: 9798400704185. DOI: `10.1145/3626202.3637571`. URL: `https://doi.org/10.1145/3626202.3637571`.

[241] Zeping Xue and David B. Thomas. "SynADT: Dynamic Data Structures in High Level Synthesis". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, pp. 64–71. DOI: `10.1109/FCCM.2016.26`.

[242] Hsin Jung Yang, Kermin Fleming, Michael Adler and Joel Emer. "LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories". In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2014, pp. 117–124. DOI: `10.1109/FCCM.2014.43`.

[243] Hsin-Jung Yang, Kermin Fleming, Michael Adler, Felix Winterstein and Joel Emer. "LMC: Automatic Resource-Aware Program-Optimized Memory Partitioning". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 128–137. ISBN: 9781450338561. DOI: `10.1145/2847263.2847283`. URL: `https://doi.org/10.1145/2847263.2847283`.

[244] Qing Yi and Ken Kennedy. “Transforming complex loop nests for locality”. AAI3047379. PhD thesis. USA: Rice University, 2002. ISBN: 049361673X.

[245] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim and Stephen A. Edwards. “Hardware synthesis from a recursive functional language”. In: *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2015, pp. 83–93. DOI: 10.1109/CODESISSS.2015.7331371.

[246] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong and Jason Cong. “Improving polyhedral code generation for high-level synthesis”. In: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2013, pp. 1–10. DOI: 10.1109/CODES-ISSS.2013.6659002.