Multi-Level Rewriting for Stream Processing to RTL compilation

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 414

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Dario Korolija, Prof. Tobias Grosser

Multi-Level Rewriting for Stream Processing to RTL compilation

by

Christian Ulmann

Supervised by

Prof. Gustavo Alonso

October 2022

D INFK

ABSTRACT

High-level synthesis (HLS) mainly aims to map sequential programs to hardware and seldomly considers alternative input types. Even though sequential programs have parallelizable parts, they are inherently not a good match for the massively parallel hardware on which HLS tries to map them. Some HLS tools have libraries that help encode parallelism in the input language, but they are combined with C-like input and require plumbing to integrate it into existing systems. Instead of consuming C as an input language, this thesis proposes a streaming abstraction (essentially a domain-specific language) that encodes thread parallelism directly and can be lowered to heavily pipelined circuits. To achieve this goal, we introduce compiler transformations that extend a dataflow abstraction used in dynamically scheduled HLS to support pipelining and build the streaming abstraction on top of it. This work demonstrates that the usage of domain-specific abstractions for hardware is a way to simplify hardware compilation substantially. The resulting hardware circuits consume data at line rate, i.e., at the maximum achievable bandwidth, without user interaction during compilation.

ACKNOWLEDGEMENTS

I thank Dario Korolija for investing so much time in helping me understand how FPGAs and the surrounding tooling work. Without his support, I would not have been capable of writing this thesis.

Furthermore, I thank Tobias Grosser for helping me enter the compiler community and providing so much helpful assistance. Over the last few years, I had the pleasure of collaborating with Tobias on different compiler projects, which sparked my passion for this subject.

Also, many thanks to Morten Borup Petersen and the rest of the CIRCT community for taking the time to answer my questions, discuss issues, and review my pull requests.

Finally, I thank Prof. Gustavo Alonso and the System Group for allowing me to write this thesis.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

With the broader adoption of field-programmable gate arrays (FPGAs) in data centers and clouds, developing applications that run directly in hardware is becoming more and more relevant for an increasing variety of use cases. Using hardware description languages (HDLs) can be cumbersome, especially for software engineers that do not have an electrical engineering background. Even when ignoring the language barrier, HDLs (often Register Transfer Languages (RTLs)) are by design more verbose due to having rich and complicated feature sets. Many of the features exposed to the users are not required for application developers. One way to counteract this problem is using high-level synthesis (HLS) that allows transforming familiar languages, usually C, to HDL. Thus, HLS tries to reduce the knowledge required to work closely with FPGAs. As hardware is inherently parallel, mapping a sequential input language, e.g., C, to it is a non-trivial problem HLS needs to resolve. Many tools try to detect parallelizable program parts or rely on the users to annotate them with pragmas [35]. Pragmas allow developers to annotate code with additional information that a compiler (or an HLS tool) can benefit from, e.g., mark loops as parallelizable. Such an approach can indeed work, as many existing tools demonstrate, but it can be very complicated to work with. Some methods pose strict limitations on the input structures, while others might be in-transparent in why specific inputs produce slow circuits. This problem is only made worse by the fact that many implementations are either monolithic, closed-source, or both. Thus, non-expert users are left guessing what went wrong when their circuits do not work as expected.

Nithin et al. [15] showed the potential of using domain-specific languages (DSL) compared to classical C-like inputs. That project focused on a machine learning DSL and is sadly no longer actively worked on. This thesis investigates if such a DSL HLS approach can be applied to another domain: stream processing. Stream processing is a way to describe computations on (potentially infinite) streams of data. Operators that work on streams do so by performing simple computations for each element. Instead of sharing state, operators exchange information explicitly with data streams. Therefore, they can work in parallel when enough data is streamed through a network. In essence, this computational model is very similar to how hardware works. Compared to C-like input, a streaming DSL is naturally parallelizable and pipelineable. Therefore, stream processing is a better match for hardware than sequential languages. Apart from being simpler to map to hardware, the higher-level input language is expected to yield optimization potential without investing in program analysis passes. Furthermore, a DSL allows to naturally restrict the structure of input programs by not exposing features that are not supported.

This work builds on top of CIRCT [9], which in turn is built on top of MLIR [26] and LLVM [25]. CIRCT defines a set of hardware abstractions in the form of MLIR dialects to provide a way to produce and transform RTL. CIRCT's RTL abstractions can be exported as Verilog for further usage. Due to being part of the MLIR ecosystem, CIRCT can interact with existing MLIR dialects, which makes it a framework for investigating HLS. Our streaming abstraction reuses parts of such an implemented HLS flow [30] that was inspired by Dynamatic [21]. Dynamatic and CIRCT's flow use a dynamically scheduled HLS (DHLS) approach that is very flexible when confronted with variable latency operations. When operations only show slow behavior infrequently, dynamic scheduling gives lower latency and higher throughput than its static counterparts. DHLS usually works with sequential input programs and thus has some assumptions about the computation it models. Reusing an abstraction for sequential programs to model stream computations pushes it beyond its original use case. The existing implementation effort of a DHLS flow in CIRCT [30] already discussed how one might implement pipelining in a DHLS setting but only implemented it for a limited set of cases. The `handshake` dialect, the dataflow abstraction in CIRCT, nicely matches the nature of a stream computation. Therefore, we implemented safety transformations to support pipelining in the `handshake` dialect and upstreamed these changes to CIRCT. By adding pipelining support to a dataflow abstraction, we enable fast executions of streams, reduce the initiation interval of sequential circuits, and provide the foundation for future work in parallel and pipelined dataflow models.

The remainder of this thesis document is structured as follows: It starts off by covering the necessary background in Chapter 2. Afterward, Chapter 3 describes the problem of the existing dataflow abstraction and suggests a way to resolve these issues. Chapter 4 proposes a streaming abstraction and gives details of its implementation in the form of an MLIR dialect. Additionally, it discusses how the dialect can be converted to CIRCT's `handshake` dialect. All the implemented work, i.e., the task pipelining transformation and the streaming abstraction, is evaluated in Chapter 5 by demonstrating the impact of the pipelining transformation and use cases of the streaming abstraction. Finally, Chapter 6 concludes this thesis and discusses possible future work.

2 BACKGROUND

This chapter will cover the necessary background material for this thesis. It starts off by introducing and defining the concepts of stream processing. Then, the notion of control flow graphs and some concepts involving them are explained. After that, we dive into high-level synthesis by describing different approaches while focusing on dynamically scheduling, as this is what we build upon. Finally, we explain MLIR and CIRCT in more detail as the implementation part of this thesis builds on top of these existing libraries.

2.1 STREAM PROCESSING

Stream processing, sometimes called dataflow processing, is a broad term that describes different systems and programming models that work with data in an element-by-element fashion. To ensure that there is no misconception of the term "stream processing", this section provides a brief description of what this thesis considers it to be.

2.1.1 *Overview*

Abstractly, a stream is a potentially infinite sequence of data elements. A stream can be consumed by operators that transform data elements and might produce new streams that contain modified data elements. Thus, a stream computation can be expressed as a directed graph, where each node is an operator, and each edge corresponds to a stream. The only way two operators can communicate is by sending data over a stream, i.e., they have no shared state. Once a data element is submitted to a stream, it can no longer be modified by its sender. Therefore, all operators can be executed in parallel without any issues as long as they have inputs to process. Streams can terminate, which is observable by their consumers. A consumer might react to that by, for example, terminating its outputs or by emitting accumulated results.

Stream processing is often used for large-scale data processing and was implemented in industry systems like Apache Spark [37] and Apache Flink [4].

2.1.2 *Example*

An example stream computation is shown in Figure 1. The incoming stream is split into two separate streams, both of which are duplications of the incoming elements. These outputs can be fed into

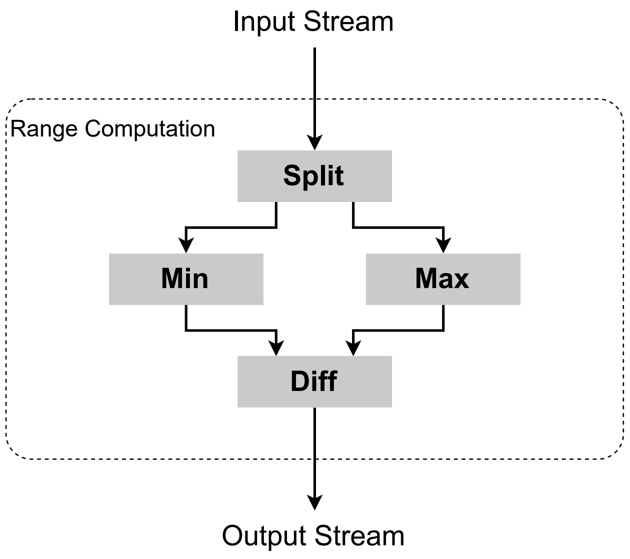


Figure 1: A stream computation that computes the range of an incoming stream. Both `Min` and `Max` function as reductions that will emit their result once their input streams are terminated.

reductions that compute the minimum and maximum, respectively. Once the input stream terminates, the reduction will emit their respective results on their output streams, and their difference will be computed by the last operation.

2.2 CONTROL FLOW GRAPHS

Control flow graphs [1] (CFGs) are an established way to represent the control flow of a sequential program. A CFG can be denoted as a directed graph where nodes are basic blocks (BBs), and edges form the immediate successor relation. A BB is a sequence of operations that has no branching in it and thus has one entry and one exit point. A CFG has one entry block and can have an arbitrary number of exit blocks. In general, it is possible to construct a CFG with a single exit node from any given input CFG. As a CFG is a form of a directed graph, all the established concepts like paths, cycles, degrees, etc., and algorithms can be used to reason about them.

2.2.1 *Dominance*

The dominance relation is an important source of information for a variety of compiler transformations.

Definition 1 (Dominator). *For a given CFG with entry block* e*, a block* n *dominates* b*, iff all paths from* e *to* b *include* n*.*

Definition 2 (Strict Dominator)**.** *For a given CFG, a block* n *strictly dominates* b*, iff* n *dominates* b *and* $n \neq b$*.*

Definition 3 (Immediate Dominator)**.** *For a given CFG, a block* n *immediately dominates* b*, iff* n *strictly dominates* b*, but no other strict dominator of* b*.*

Analogous to domination, there is the concept of post-domination, which takes the paths from b to a unique exit node into account.

2.2.2 *Loop Terminology*

CFGs can contain cycles, usually caused by loops in the input program. In this work we use the same terminology for loops as LLVM [10]. A loop in a CFG is a strongly connected component (SCC) where all entering edges point to the same block, the loop **header**. Each block that is a successor of the loop header but is not part of the loop is called an **entering** block. The remaining successors of a loop header, i.e., the ones of the loop, are called **latch**. Blocks with a successor that is not part of a loop are called an **exiting** block, while these successors are denoted as **exit** blocks.

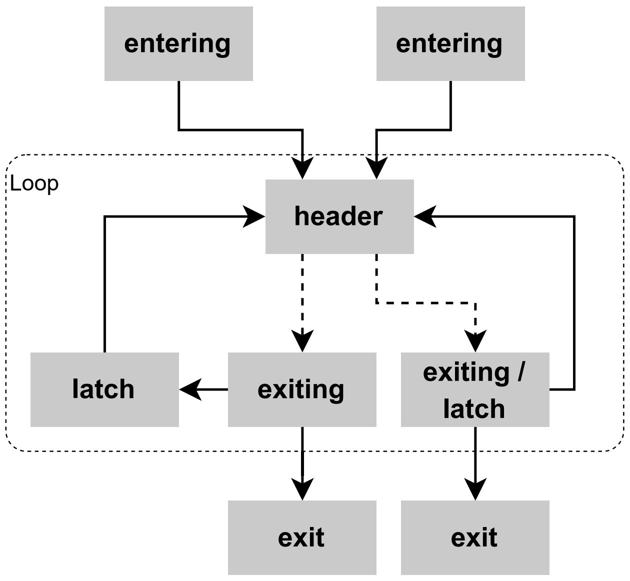


Figure 2: LLVM's loop terminology has clear names for all relevant blocks of a loop. A specific block can belong to multiple block groups, e.g., it can be both an existing block and a latch.

2.3 HIGH-LEVEL SYNTHESIS

In general, HLS aims to ease the development of applications for hardware by translating familiar languages, mainly C/C++, to hardware description languages (HDLs). Writing complex applications in HDL requires a lot of knowledge about how the hardware behaves. While this can be a necessity when one aims to design infrastructure, it is an unnecessary complication when writing data processing applications. Similar to how it is sometimes required to implement certain software parts in assembly, using higher-level languages is much more productive and simpler. Programming languages provide expressive features to model complex programs in just a few lines of code. On the other hand, writing equivalent programs in assembly requires careful thought about all the minor details, which are otherwise taken care of by a compiler. HLS aims to do the same thing, except that it does not replace assembly but HDL. By enabling developers to compile programs written in a high-level language down to HDL, HLS removes substantial amounts of complexity. To simplify the problem for the developers, it is, at least partially, moved into the HLS compilers. To produce performant circuits, HLS must exploit parallelism as much as possible. Thus, it has to first find program parts worth parallelizing. Some HLS systems rely on powerful compiler analysis to discover such parts, while others still rely on the developers to annotate them with pragmas [35]. Many industry tools [18, 36], in fact, support automatic optimizations but rely on pragmas to reduce the search space.

While HLS tries to hide the parallel nature of hardware, the usage of pragmas destroys this illusion. Forcing the usage of pragmas, especially in sequential languages, to reach high performance defeats the whole purpose of HLS.

2.3.1 *Classical HLS*

The classical (or statically scheduled) HLS approach [12], as used in many existing tools [3, 18, 36], transforms an input program into an intermediate representation (IR) that encodes both control and dataflow. The IR can be optimized by using analysis information or information provided by pragmas. Afterward, dataflow parts are assigned to different clock cycles in the scheduling process. Scheduling takes the dependencies and latencies of dataflow components into account when mapping them to cycles. All of this is done statically, i.e., during compile time. Therefore, a scheduler needs to be conservative when working with operations that have variable latencies, e.g., memory accesses. Such operations are treated as if they always cause the worst-case latency.

A controller, in the form of a finite-state machine (FSM), is used to manage the control flow, e.g., to support unbounded loops. The FSM uses status signals from the data path and external control inputs to orchestrate the datapath. In a certain sense, HLS tries to build a CPU with custom operations. This approach can work but has limited applicability due to the frequency differences between CPUs and FPGAs.

2.3.2 *Library & domain-specific HLS*

Another approach to HLS is to encode domain-specific knowledge in the input language to simplify the work for a compiler. Some approaches expose features in the form of libraries to a language. The compiler can then use this to build up an abstraction it can analyze much easier. Instead of providing libraries to use, some HLS approaches [15] rely on domain-specific language (DSL) as input. Analyzing a DSL can be much simpler than analyzing C-like programs. Furthermore, it allows encoding high-level constructs directly in the language, which simplifies development even further.

The internal functionalities of such HLS tools can be similar to the ones of classical HLS or DHLS, depending on the domain they want to work with.

2.3.3 *Dynamically scheduled HLS*

As an alternative to statically scheduled HLS, an approach called dynamically scheduled HLS [21] (DHLS) started emerging in recent years. As the name suggests, this type of HLS does not statically schedule operations to be executed in specific clock cycles. Instead, it constructs a circuit that executes operations once its predecessors are ready, independent of their respective latencies. Therefore, components with variable latencies do not cause unnecessary stalls when they are not required.

To further speed up the execution, out-of-order load-store queues (LSQ) [20] can be used to dynamically resolve memory conflicts. An LSQ connects to all memory operations of a DHLS circuit. It consumes and fires control signals to dynamically check memory operations for conflicts to potentially stall operations if required. As a DHSL circuit can handle arbitrary delays, LSQs substantially speed up their execution.

2.3.3.1 *Existing Approaches*

Like some classical HLS implementations, the existing DHLS implementations [21, 30] leverage existing software compiler frontends to parse C/C++ and produce an intermediate representation (IR) from it. In the case of LLVM-IR, this representation is a static single assignment (SSA) [31] CFG. Apart from reducing the implementation effort, using an infrastructure like LLVM enables the reuse of existing compiler transformations and optimizations to bring the IR into a usable form.

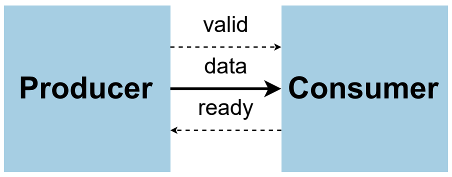


Figure 3: If both the valid and ready signals are asserted in the same cycle, data is transacted.

The existing DHLS approaches produce dataflow circuits from an IR in a structured way due to relying on the flexibility provided by compositional dataflow circuits [13]. Compositional dataflow circuits are based on elastic circuits [11]. Thus, they wrap all their in and outputs into a handshaked interface. This mechanism adds two 1-bit wires for each data wire, the `valid` wire from sender to consumer, and the `ready` in the other direction. Only when both signals are asserted a data transfer happens.

Existing DHLS implementations define a set of dataflow operators which have well-defined semantics. Having an explicit abstraction enables transforming a CFG into a dataflow abstraction. On this abstraction, some further steps can be performed [5, 6, 22] before it is finally converted to HDL.

The used abstraction has the following operations and components:

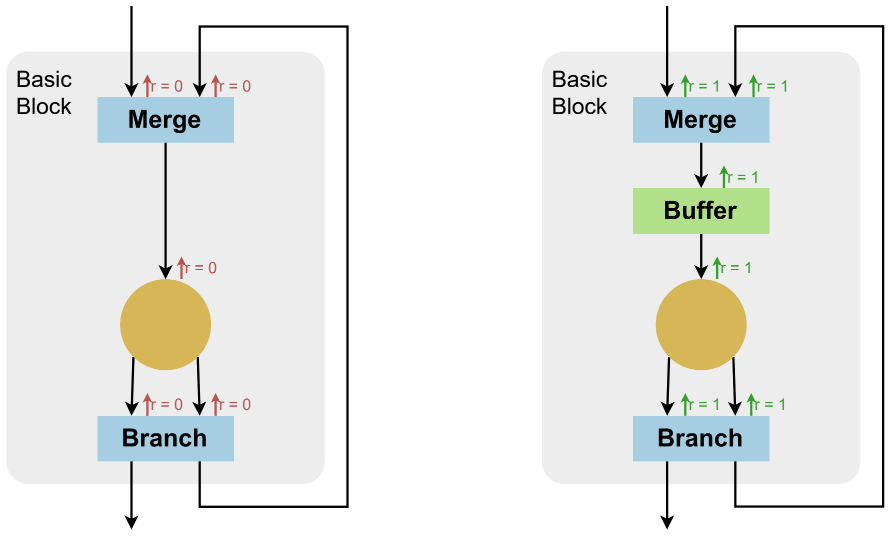
- **Buffers** are a storage unit which can hold up to N elements and are the elastic equivalent of registers.
- **FIFOs** are similar to buffers, but they have an additional bypass to forward elements directly, if the consumer is ready.
- **Forks** forward their input to all the consumers as soon as they are ready. Before all consumers consume the current input, a fork will not accept another input.
- **LazyForks** are similar to Forks, except that they only forward inputs when all their consumers are ready.
- **Joins** await all their inputs before emitting an output. Joins are control only, i.e., they do not have a data wire, only `valid` and `ready`.
- **Branches** have two outputs and will forward a token to the output which match the boolean input condition. They are thus similar to control flow branches.
- **Merges** are the counterpart to branches. They non-deterministically forward any input to their output.
- **Control Merges** function like the merge operation except that they output the index of the transacted input as well.

- **Muxes** forward one of their input, depending on the input index provided. The inputs that were not selected will not be consumed and are still available in a following step.
- **Selects** are similar to muxes, except that they require all inputs to be present and that they consume all of them.
- **Sinks** can always consume their input and will simply drop it, i.e., they always assert `ready`.

Once the dataflow graph is brought into a satisfactory form, it has to be converted to an equivalent HDL representation. This transformation is achieved by instantiating and connecting HDL templates for all the operators.

2.3.3.2 *Buffer Insertion*

The dataflow representation produced by DHLS approaches is not always directly executable due to cyclic structures causing deadlocks. Most dataflow primitives cannot store values and thus can only accept inputs when the produced output can be forwarded to their successors. For a cycle of operations that all have this behavior, none of them will signal `ready`, which causes a deadlock. Buffers must be inserted into loops to ensure that such cyclic dependencies are broken up. An empty buffer can accept inputs, as it does not have to forward the data directly.



(a) A cyclic dataflow graph causes deadlocks.

(b) Inserting a buffer ensures that the cycle can hold a value.

Figure 4: Buffers are required to prevent deadlocks in dataflow cycles.

Apart from inserting buffers to eliminate deadlocks, adding buffers is necessary for reaching high performance. Having multiple datapaths through a circuit will require faster datapaths to stall. Inserting FIFOs on the fast paths will ensure that this stalling is hidden and no back-pressure can cause a throughput reduction. Furthermore, sequential buffers function similarly to RTL registers and will therefore split combinatorial paths over different cycles. In general, adding too many sequential buffers in a dataflow graph will increase its latency but will only affect throughput when there is a cycle. Adding too many FIFOs, on the other hand, only increases area usage while not affecting latency. Optimal buffering has been proposed for sequential circuits [22], so this work will not dive deeper into this topic.

2.4 MLIR

As LLVM grows in popularity, there are more and more custom frontends compiling their input to LLVM-IR and thus relying on LLVM to do the rest. All these custom compilers and their languages can profit from the reusability of LLVM by compiling the input to LLVM-IR.

One problem of LLVM-IR is that it is too low-level to capture some aspects of high-level languages nicely. This resulted in the development of language-specific IRs for many languages that use LLVM, e.g., Swift Intermediate Language (SIL) for Swift, MIR for Rust, etc. These IRs enable powerful language-specific optimizations as they can preserve most of the original semantics. As soon as these IRs are lowered to LLVM-IR, their high-level semantics are lost. For example, loops get replaced by conditional branches. This loss of structure makes some analysis tasks more complicated and, therefore, more expensive and harder to implement.

MLIR (Multi-Level Intermediate Representation) [26] aims to change this by introducing an extensible IR that allows for easier integration of new IR features to capture more semantics. To do so, it specifies a generic IR that can be extended. This structure not only offers generic parsers and printers but also allows defining reusable lowering and optimization passes.

2.4.1 *IR Design*

MLIR, like LLVM-IR, implements an SSA type of IR. SSA has the benefit of assigning each variable only once, which makes most data flow analyzes simpler. Furthermore, it is efficiently computable from classical control flow graphs.

Even though MLIR is heavily extensible, it has a basic structure that must be obeyed. To start off, we show an example MLIR function with some instructions. We then use this example to elaborate on the different MLIR constructs.

```
"func.func"() ({
^bb0(%arg0: memref<1024xf32>, %arg1: index):
  %0 = "memref.load"(%arg0, %arg1)
      : (memref<1024xf32>, index) -> f32
  %1 = "arith.constant"() {value = 0 : f32} : () -> f32
  %2 = "arith.cmpf"(%0, %1) {predicate = 9 : i64}
      : (f32, f32) -> i1
  %3 = "scf.if"(%2) ({
    %4 = "arith.addf"(%0, %0) : (f32, f32) -> f32
    "scf.yield"(%4) : (f32) -> ()
  }, {
    "scf.yield"(%0) : (f32) -> ()
  }) : (i1) -> f32
  "func.return"(%3) : (f32) -> ()
}) {function_type = (memref<1024xf32>, index) -> f32,
    sym_name = "demo"} : () -> ()
```

Listing 1: In the generic representation, each operation is printed in a verbose but uniform representation. The semantics of this example is irrelevant.

Most important are operations. They can not only represent instructions but functions and modules as well. Each operation has a unique name or upcode, a list of arguments, a list of results, a type, regions, and an attribute dictionary.

- Arguments are SSA values that are required by an operation. This list can also be empty, depending on the operand kind.
- Results are SSA values that are produced by an operation. Some operations do not produce results and therefore do not have to specify them.
- Every operation has a type that is constructed from the argument and result types.
- Blocks are similar to LLVM-IR basic blocks but have one major difference: They can contain operations that can hold regions. Each block is a list of typically sequentially executed instructions and always terminates with a terminator operation. In MLIR, blocks have block arguments instead of the ϕ-nodes [31]. If a block is executed, it has to be "called" with matching block arguments similar to a function call.
- A region forms an SSA CFG that belongs to an operation. This allows subdividing the general CFG into subgraphs that are operation-specific.

```
func.func @demo(%arg0: memref<1024xf32>, %arg1: index) -> f32 {
  %0 = memref.load %arg0[%arg1] : memref<1024xf32>
  %cst = arith.constant 0 : f32
  %1 = arith.cmpf ugt, %0, %cst : f32
  %2 = scf.if %1 -> (f32) {
    %3 = arith.addf %0, %0 : f32
    scf.yield %3 : f32
  } else {
    scf.yield %0 : f32
  }
  return %2 : f32
}
```

Listing 2: When pretty printing operation, attributes and types are omitted wherever possible to make the IR much less verbose. This program is the same as in Listing 1

- Attributes allow parametrizing operations with static information, e.g., loop bounds or callee names. Attribute values are known at compile time and are provided in an attribute dictionary that binds a value to the corresponding name. Similar to operation names, attribute names should be unique.

The example in Listing 1 is very verbose and somewhat hard to read. To make working with the IR easier, MLIR supports custom formatting that eliminates a lot of redundancy and tries to group relevant information (see Listing 2).

2.4.1.1 *Operation Definition*

Each operation kind has a declaration that defines a set of constraints and traits, which can be used to encode requirements for legal operations. For example, the operation `addf` has operand constraints that enforce the passed inputs to have a `FloatLike` type. Additionally, it has the traits `Commutative`, `SameOperandAndResultTypes`, and `NoSideEffects`. The first one informs general optimization passes to reorder the inputs, the second one yields an additional verification constraint, and the last trait allows the generic dead-code elimination to remove an `addf` operation if its result isn't used.

2.4.2 *Dialects*

The concept of dialects provides a way of organizing operations into abstractions that are isolated from others. Dialects can be treated specifically by optimizing, lowering, and analyzing passes. All operation, attribute, and type names are prefixed with the dialect name that serves as a namespace to avoid name clashes.

One of the existing dialects is the vector dialect. Its purpose is to represent vector types and operations, e.g., transposition, matrix multiplication, outer products, and many more. Representing these in a lower dialect or plain LLVM-IR is possible, but analyzing it is more difficult or even impossible. For example, a sequence of vector transpositions (as in Listing 3) can be eliminated or replaced by a single transposition. Detecting such cases is straightforward when inspecting such a sequence represented in the `vector` dialect. A simple pattern match that looks for a transpose whose input is produced by another transpose can detect and optimize such a case. On the other hand, detecting these redundant transpositions on the LLVM-IR level can be complicated and requires powerful and possibly slow analyzes.

```
%1 = vector.transpose %0, [1, 0]
  : vector<1x3xf32> to vector<3x1xf32>
%2 = vector.transpose %1, [1, 0]
  : vector<3x1xf32> to vector<1x3xf32>
```

Listing 3: Vector transposition

MLIR dialects can be mixed (as in Listing 2) to ensure interoperability and reusability. Mixing leads to more strictly grouped dialects, which can be observed in many lower-level dialects. For example, `memref`, `arith`, and `func` only contain operations that are specific to their respective domains.

2.4.3 *Progressive Lowering*

MLIR allows us to perform optimizations and lowering in a pipeline fashion. Lowering is the process of transforming a higher-level dialect into a lower one. Progressive lowering in this context means that the IR is converted step-by-step to lower-level dialects on which specific optimizations can be applied. MLIR passes can specify which operations are considered legal and illegal after they terminate to make the conversion as flexible as possible.

2.5 CIRCT

The motivation behind CIRCT is to unify the open-source community around EDA tools by providing a collection of hardware abstractions, tooling, and RTL generators (see Figure 5 for a dialect overview.). The community applies ideas from the MLIR [26] project to a hardware compiler infrastructure. While compilers like LLVM have a well-defined intermediate representation that can be used to interface between tools, the EDA domain relies on Verilog to do the same. Unfortunately, Verilog and SystemVerilog are very complicated languages with rich feature sets. Many tools only implement a subset of these

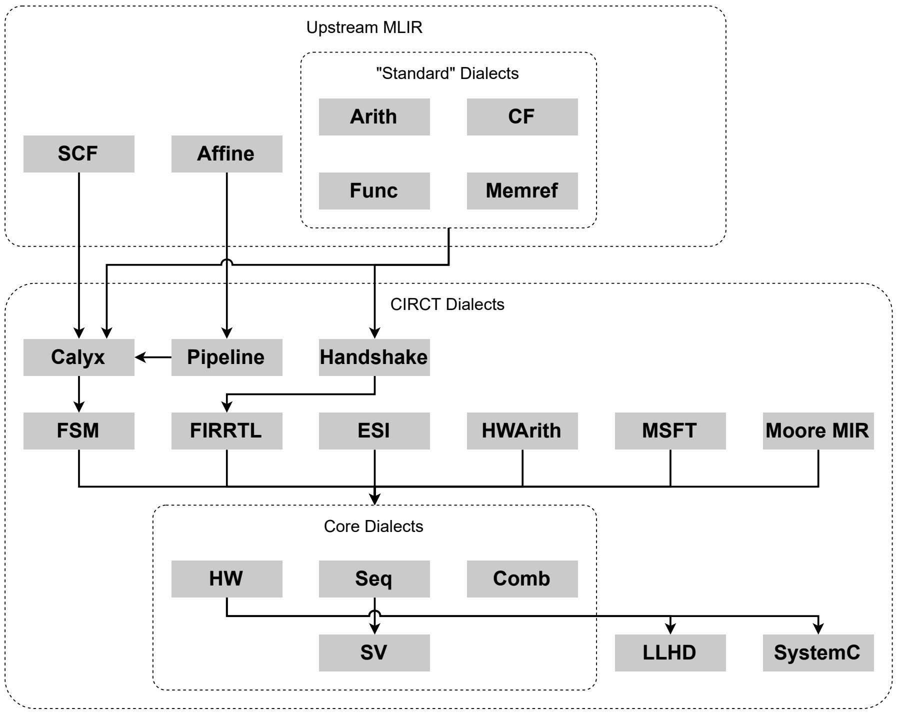


Figure 5: An overview of all dialects in CIRCT and MLIR dialects that interact through transformations in CIRCT. Each box is a separate dialect, and each arrow corresponds to a conversion pass.

features and thus are frequently incompatible. To address this, CIRCT has a configurable Verilog emission component that allows defining the used language feature set.

Apart from providing dialects for RTL-level abstractions, the project started to attract higher-level dialects for FIRRTL [19] and Calyx [29]. The FIRRTL dialect is part of a drop-in replacement for the Chisel compiler [2]. Chisel is an HDL implemented as a Scala-embedded DSL that can use generators and other higher-level features to simplify developing hardware. The Chisel compiler produces Verilog.

2.5.1 *Usage for HLS*

Due to CIRCT being built into the MLIR ecosystem, both hardware and software dialects can interact without having to cross framework boundaries. Therefore, CIRCT can serve as a driver for HLS research. It already contains dialects for dataflow (the `handshake` dialect), FSMs, and for static pipelines. Some transformations allow translating lower-level MLIR dialects into the `handshake` dialect, which essentially is a DHLS flow.

2.6 COYOTE

Even though FPGAs are no new technology, their adoption is still non-trivial. All models have a different set of supported functionalities and forms of interconnection. Thus, applications are often written for one specific version of an FPGA without the goal of eventually porting it to a new chip. Coyote [23] tries to change that by implementing a shell in which user logic can be executed. This shell provides an OS-like abstraction of the surrounding system that user programs do not have to be aware of the exact system they are running on.

To interface with the use logic, coyote uses AXI4 stream [28]. The AXI4 specification is provided by ARM and can be used royalty-free. AXI4 is heavily used in industry, e.g., by Xilinx. Thus, many existing systems have such interfaces.

3 PARALALLIZED DHLS

This section starts by covering some basics regarding parallelism in hardware. Then it discusses the problems of existing DHLS abstractions when they are used to model pipelined executions. Afterward, we present a more rigorous analysis of the task pipelining issues and present mechanisms to resolve them. This presentation includes an algorithmic description and corresponding correctness argument. Finally, we cover some of the remaining limitations that our approach cannot address.

3.1 EXPLOITING PARALLELISM IN HLS

The ability to perform computations in parallel is the main benefit of spacial accelerators like FPGAs compared to classical CPUs. Therefore, the main objective of HLS tools is to find and exploit parallelism in the input programs. The two main sources of parallelism in sequential programs are instruction-level parallelism (ILP) and independent loop iterations. CPU architectures have multiple approaches to resolving these issues. Very large instruction words [14] (VLIW) CPUs rely on a compiler to detect the ILP and perform scheduling by packing independent operations into large instruction words. VLIW is the CPUs equivalent of static scheduling, as it simplifies the hardware but increases the compiler's complexity. Superscalar, out-of-order CPU architectures take a more dynamic approach. These architectures can execute multiple instructions in parallel. To do so, they resolve conflicts dynamically and issue instructions when all their dependencies are ready. DHLS is conceptually similar, as it resolves conflicts dynamically, i.e., at run time. Other CPU features that exploit parallelism are vector instruction sets. These operations execute a single instruction on each vector element in parallel, thus the terminology "single instruction multiple data" (SIMD). The HLS equivalent is to replicate hardware units and feed them with the independent inputs (horizontal scaling [27]).

By the sequential nature of the usual HLS input programs, there is only a limited amount of parallelizable structure to extract. Thus, instead of parallelizing single executions, one can aim to parallelize different executions. Circuit must be pipelined to do so.

As this project focuses on HLS for streaming, pipelining is a natural way to increase hardware parallelism. Assuming that a streaming circuit can be optimized and pipelined to have an initiation interval (II) of 1, no horizontal scaling is required to reach peak throughput.

3.2 LIMITATIONS OF DHLS

Traditional DHLS translates from C-style control flow graphs (CFGs) to dataflow abstractions assuming a single-threaded execution. Therefore, the existing approaches cannot give any guarantees about how produced circuits behave if they are used in a pipelined manner. Due to their assumptions, existing tools enforce a high II, which behaves like a lock around the circuit. Removing the II constraint breaks the produced circuits. For example, a program with diverging control flow (Figure 6a) will accept new inputs even though they can cause incorrect behavior.

Definition 4 (Correct task pipelined execution)**.** *We consider a circuit to support correct task pipelined execution if the following properties hold statically:*

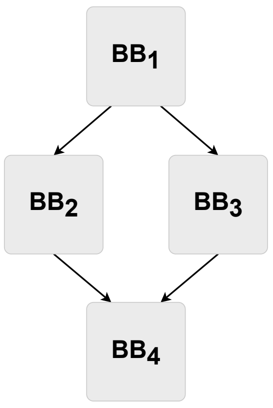
1. *It preserves the order of results, such that it is equal to the order of their corresponding inputs.*
2. *No non-deterministic behavior can be triggered.*

The provided example has two paths with different latencies and thus exhibits unexpected behavior. Either results can get reordered (invocation 1 and 2 in Figure 6b) or threads collide when entering a block (invocations 1 and 3), which causes non-deterministic ordering. A correct schedule should ensure that the ordering of the outputs matches the corresponding inputs (Figure 6c).

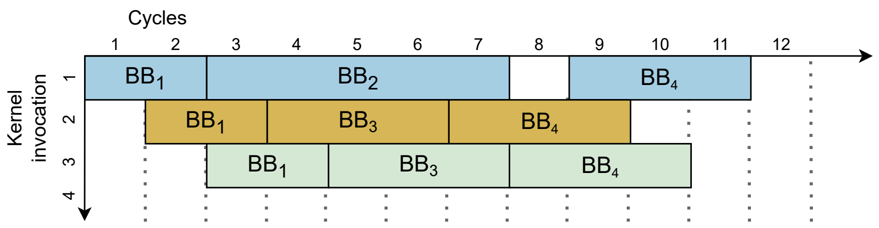
A similar problem arises for loops. Consider a CFG with a simple loop (Figure 7). No mechanism ensures that an incoming thread does not collide with threads that take the back edge. Therefore, such CFGs, without any additional mechanism, produce incorrect results when executed in a task pipelined manner.

One cause of these problems is the usage of non-deterministic merge-like nodes (Figure 7.i). This operation combines multiple inputs into one output, similar to ϕ-nodes [31]. The non-deterministic behavior only occurs if multiple inputs to a merge-like operation fire at the same time, which can be guaranteed to never happen in a single-threaded setting. In a CFG, a basic block (BB) has no control flow and executes its operations sequentially. As only one BB is active at each point in time, different predecessor blocks cannot fire in the same cycle. The only case where problems occur is when a set of blocks form a combinatorial cycle. As discussed in Section 2.3.3.2, this is already prevented. This mechanism breaks down in a task pipelined context, as multiple merge inputs can become valid.

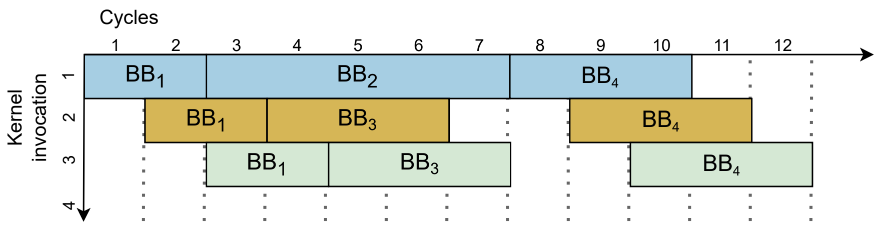
Instead of using merge-like nodes that combine multiple inputs without controlling the order of the inputs, one can use muxes. A mux node in a dataflow abstraction will only consume the value provided on the selected input while stalling the others. This functionality,



(a) A diamond CFG



(b) Schedule of the diamond CFG without an additional task pipelining protection. The order of BB$_4$ executions for invocations 1 and 3 is non-deterministic.



(c) Schedule of the diamond CFG with an additional task pipelining protection.

Figure 6: Schedules of a diamond CFG with different latencies. Using the circuits intended for sequential execution causes reordering and collisions of block invocations, which results in a non-deterministic order of results. The protection mechanism ensures correct orderings by stalling the execution of BB$_4$ for the faster path.

combined with a way to determine the port to forward, can be enough to enable correct task pipelining. The challenge is to find the places to insert these muxes. This work suggests a way to recursively decompose SSA CFGs into different graph types that require a specific protection mechanism.

Before we discuss the details of this, it is important to state that this limitation was discovered M. B. Petersen [30]. Our work analyzes the problem more rigorously and extends the existing implementation with handling for feed-forward cases.

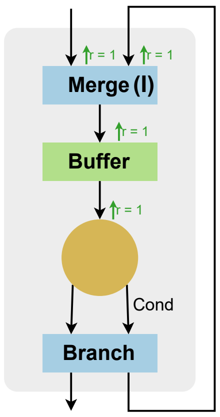


Figure 7: A loop can cause issues due to the merge node accepting from any input that is available.

3.3 SUBGRAPH TYPES

This subsection describes the different kinds of subgraphs that can cause incorrect behavior. Additionally, the transformations required to guarantee correct execution are presented.

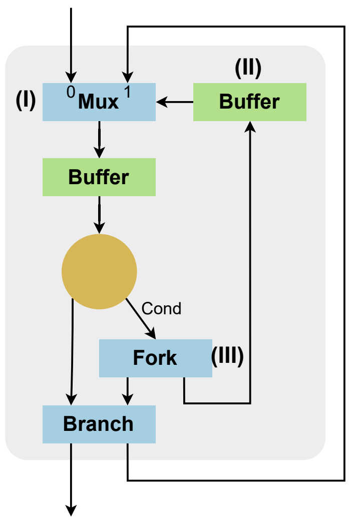
3.3.1 *Canonical Form*

Some assumptions on the structure of the CFGs are required to simplify the following reasoning:

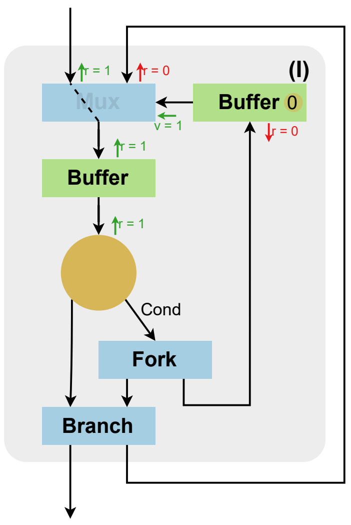
1. In all cases, a subgraph has exactly one `source` node and one `sink` node. Furthermore, it must be true that `source` dominates `sink` and that `sink` in turns post-dominates `source`.
2. Each block that is not part of a strongly connected component (SCC) has at most two successors and two predecessors.
3. A loop has one exiting and exit node.
4. There are no structural infinite loops.

3.3.2 *Loops (Feed-back CFGs)*

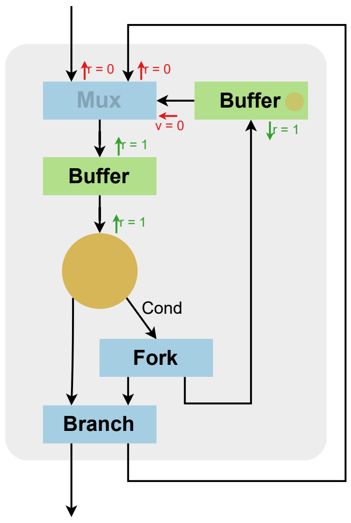
A loop in a CFG corresponds to an SCC with one node having an incoming edge, as described in Section 2.2.2. We do not support pipelining of different threads within a loop, as this comes with challenges. The most complicated one is that threads can have different latencies, which require a reorder buffer implementation, which is currently not present. A simple mechanism that locks a subgraph can guarantee correctness. Similar to a lock in multi-threaded programming, this mechanism blocks all threads from entering the critical section as long as it contains one. Once a thread exits the subgraph, the next may enter. While this mechanism locks subgraphs, different threads can still be active in the same circuit, just not in the same loop.



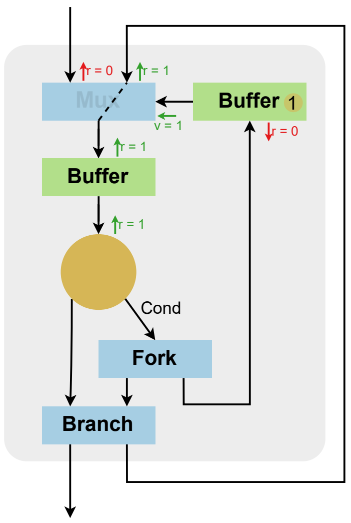
(a) Deadlock prevention



(b) Initialized loop guard



(c) Accepting no inputs

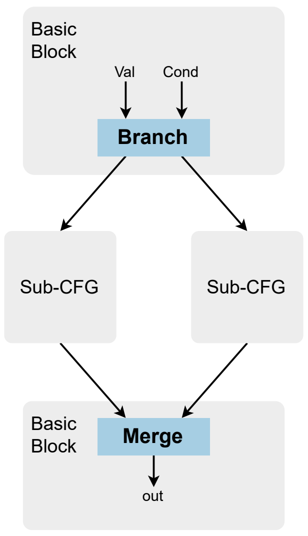


(d) Accepting internal backedges

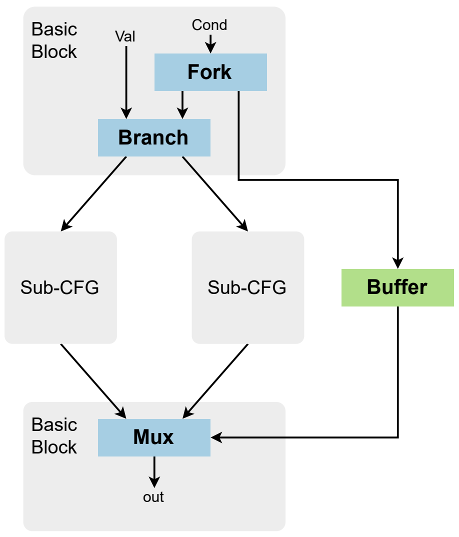
Figure 8: Loop protection mechanism and its different possible states during execution.

Such a mechanism replaces the existing merge node (Figure 7.i) with a mux (Figure 8a.i) that allows to select the input from the correct predecessor BB. The input's index to select is provided by a newly added buffer of size 1 (Figure 8a.ii). The mux consumes the buffer value once the corresponding input is valid (Figure 8b for external inputs, Figure 8d for internal loop back edges). While the buffer is empty, the mux will not be able to select any additional inputs (Figure 8c). Upon reset, the buffer is initialized with a token such that the first thread entering the subgraph will be selected (Figure 8b.i). When the control flow reaches a branch that determines if the loop exits or if the next iteration will start, it will refill the buffer with a new token that indicates the correct predecessor (Figure 8a.iii).

In the case of nested loops, one lock that protects the outermost loop is sufficient to protect the complete nest. Since only one thread enters the outermost loop, there can be at most one thread in any of its inner loops.



(a) Diverging control flow



(b) Reorder prevention

Figure 9: A CFG with diverging control flow cannot guarantee correct execution under task pipelining. Inserting a reordering prevention mechanism ensures that results are in order and do not collide.

3.3.3 *Feed-forward CFGs*

Feed-forward subgraphs have either a diamond (Figure 9) or a triangular shape. Formally, a feed-forward subgraph is formed by a pair $(\mathrm{source}, \mathrm{sink})$ with $\mathrm{out\_deg}(\mathrm{source}) = 2$ and $\mathrm{in\_deg}(\mathrm{sink}) = 2$. The triangular case has a direct edge from source to sink while the other part forms a subgraph. The diamond case has some more restrictions: Let s_0 and s_1 be the two successors of source and p_0 and p_1 be the two predecessors of sink. It is required for each s_i that there is only a path to one of the p_js. If both ps are reachable, this is not considered a feed-forward CFG.

If one recursively ensures correct execution in all subgraphs, a mechanism that prevents reorderings and collisions at the point of convergence can guarantee correctness. Storing the directions the threads select at the entry block's branch allows us to collect the results from the correct subgraph (Figure 9b). As a buffer preserves a

FIFO order and a mux will only consume selection arguments once the corresponding input fires, this guarantees correctness.

Note that the buffer size determines the number of threads the CFG can contain. Thus, the buffer size should be in the order of the latency of the slow path.

3.3.4 *Irreducible CFGs*

The last types of CFG this transformation considers are irreducible subgraphs. The only difference to the feed-forward case is that when removing both `source` and `sink`, only one graph remains, i.e., one successor of `source` can reach both predecessors of `sink` (see Figure 10 for an example).

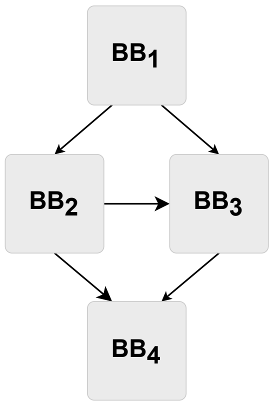
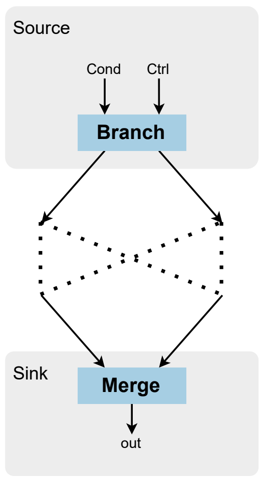


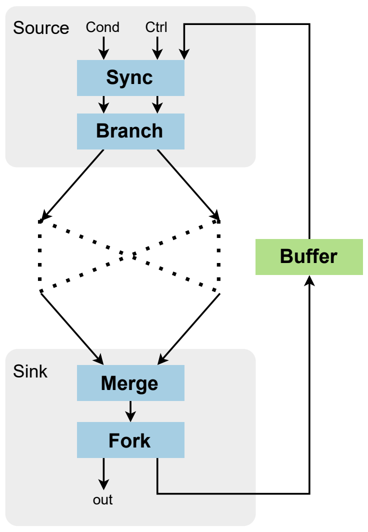
Figure 10: An irreduccible CFG that cannot be executed in a task pipelined manner.

These graphs have multiple points where control flow paths can merge again. Thus, such cases cannot support multiple threads without structural changes to the CFG. These subgraphs are locked by inserting a `sync` operation that is fed by a token which will only be available when the previous execution reaches the sink block (Figure 11b). `sync` forwards all its inputs as is but only does so once all of them are valid, i.e., it is similar to a barrier. The `sync` operation was previously not part of any DHLS dataflow abstraction. Thus, we introduced it to the `handshake` dialect.

Such CFGs generally only arise when using unstructured control flow constructs in the source abstraction (e.g., `goto` statements). Structured control flow (`if`, loops) always result in reducible CFGs. Due to these structures arising seldomly and time constraints, we did not implement this case but present it here for completeness.



(a) Irreducible CFG



(b) Locked irreducible CFG

Figure 11: An irreducible CFG needs a kind of locking mechanism that ensures that at most one thread is in it.

3.4 GRAPH DECOMPOSITION

To discuss this problem formally, it is necessary to state how the previously discussed subgraphs can be found. The only relevant subgraph types are the ones described beforehand. These subgraphs can be found by recursively traversing the CFG of a function as described in Algorithm 1. The algorithm produces three sets of subgraphs that then must be protected as described in Section 3.3.

3.5 CORRECTNESS ARGUMENT/PROOF

The following proof shows that the described transformation ensures that the dataflow graphs support task pipelining. Section 3.3 already discussed how the guard mechanisms applied to the different subgraph types guarantee these properties, but how these properties expand to the full CFG remains to be shown.

Let $CF(g)$ for a CFG g be a predicate that indicates if a graph satisfies the canonical form described in Section 3.3.1. Furthermore, we define $P(g) \equiv CF(g) \implies safe(T(g))$, where T is the described transformation, and $safe$ is a predicate that indicates if the graph is safe for task pipelined executions. We prove $\forall g. P(g)$ by strong structural induction. Thus, we have to show $P(g)$ for an arbitrary CFG g and assume $\forall g' \sqsubset g. P(g')$ as our induction hypothesis. Assume that $CF(g)$, as other cases are trivial. From $CF(g)$ follows that $g = (b, t)$, where b and t are entry and exit blocks of the CFG respectively. We proceed with case analysis on g:

Algorithm 1 Algorithm to detect different CFG parts

```
function FINDSUBGRAPHS(b)
    if $visited(b)$ then
        return
    end if
    Mark p as visited
    if $numSucc(b) = 0$ then
        return
    end if
    if $isLoopHeader(b)$ then
        $l \leftarrow loopAt(b)$
        $e \leftarrow l.exitNode$
        Add $(b, e)$ to the set of loops
        FINDSUBGRAPHS(e)
        return
    end if
    $d \leftarrow immediatePostDom(b)$
    if $d \in succs(b)$ then
        if $|succs(b)| > 1$ then
            Add $(b, d)$ to the set of feed-forward CFGs
            $s \leftarrow$ the other successor of $b$
            FINDSUBGRAPHS(s)
        end if
        FINDSUBGRAPHS(d)
        return
    end if
    if $\forall p \in preds(d) : \exists s \in succs(b) : dom(s, p)$ then
        Add $(b, d)$ to the set of feed-forward CFGs
        $\forall s \in succs(b)$ : FINDSUBGRAPHS(s)
        return
    end if
    Add $(b, d)$ to the set of irreducible CFGs
    FINDSUBGRAPHS(d)
    return
end function
```

CASE $b == t$: In this case, the region detection algorithm terminates directly, and no changes happen. As g only contains one BB, g trivially satisfies the correctness property.

CASE $\mathtt{isLoopHeader}(b)$: Let e be the exit node of the loop starting at b. (b, e) is a loop and thus will be guarded by essentially locking it. As e is the only exit node, it follows that e is dominated by b and post-dominates b. It follows that $CF((e, t))$ must hold. As $(e, t) \sqsubset (b, t)$ the induction hypothesis applies, and we conclude that the transformation will ensure a correct execution for graph g.

CASE $|\mathtt{succs}(b)| = 1$: Let s be the sole successor of b. From $CF(g)$ it follows that b must dominate s, and as b is the only successor, s post-dominates b. It follows that $CF((s, t))$ must hold as well. Due to $(s, t) \sqsubset g$, we get that the recursive call ensures the correctness of this subgraph. As the BB b only branches into s, no reordering can happen at this point, so this case is safe.

CASE $|\mathtt{succs}(b)| = 2$: Let $d = immediatePostDom(b)$. From $CF((b, t))$ it follows that d dominates t and t post-dominates d. Thus $CF((d, t))$ must hold, which implies the induction hypothesis. So we get that the subgraph formed by (d, t) will be transformed correctly. It remains to be shown that the property holds for (b, d). We show this with an additional case split:

SUBCASE $d \in \mathtt{succs}(b)$: This case implies that (b, d) forms a triangular CFG. Let sub be the only subgraph of (b, d). $CF(sub)$ trivially holds, so the induction hypothesis guarantees that the recursive transformation ensures correctness. (b, d) will be protected as described before, as its subgraph is correct, (b, d) is as well.

SUBCASE $\forall p \in \mathtt{preds}(d) : \exists s \in \mathtt{succs}(b) : \mathtt{dom}(s, p)$: s Wlog. we assume that $sub_1 = (p_0, s_0)$ and $sub_2 = (p_1, s_1)$ form such pairs, thus we get $CF((p_0, s_0))$ and $CF((p_1, s_1))$. As both $sub_1, sub_2 \sqsubset g$, the induction hypothesis yields that the required property holds for both subgraphs. Therefore, the reordering mechanism that is applied here will guarantee correct execution.

SUBCASE OTHERWISE: This implies that (b, d) forms an irreducible subgraph that will be locked. Thus, this case trivially satisfies the correctness property.

3.6 A NOTE ON MEMORY

In the presence of memory, the described mechanisms are not guaranteed to produce correct results. If multiple threads use the same memory, it is, in general, impossible to statically show that the threads cannot have memory conflicts. Every usable programming model should guarantee that threads behave as if executed in isolation. Without this property, pipelining can change semantics, which makes programming in such a model unpredictable.

One memory block can be used by an arbitrary number of memory operations, and all these operations can potentially have conflicts. Until the last memory operation of a thread is executed, it is unclear what exact addresses it will access. Therefore, a subsequent thread has to stall until all memory operations are performed. This stalling is similar to locking. Thus, no general pipelining can be implemented.

In some sense, different threads running in isolation, but still using the same memory, are similar to how database transactions work [17]. In contrast to threads in pipelined hardware, database transactions can be aborted and restarted, as orders must not be preserved.

4 STREAM PROCESSING

This chapter gives an overview of the streaming part of this work. Most of the work described in this section was only possible due to adding support for task pipelining to the DHLS flow (described in Chapter 3).

The chapter starts off by giving an architectural overview of this work. After that, it discusses the design of the streaming abstraction and its semantics. Then, the lowering to CIRCT's `handshake` dialect is covered. The last section discusses how the circuits produced from the streaming abstraction can interact with external components.

4.1 ARCHITECTURAL OVERVIEW

This section gives a high-level overview of the compilation flow implemented in this project. The flow consumes inputs in the form of a streaming abstraction that is a mixture of MLIR dialects and the newly designed `stream` dialect (Figure 12). The `stream` dialect provides types that represent streams and operations to work with them. The operations express high-level primitives, e.g., `map` and `reduce`, and can be parametrized with regions to express a specific computation. A region in this context can be compared to a lambda function in other high-level programming languages. Depending on the operation it is attached to, it takes a set of inputs and is expected to produce results with certain types. The computation in the region is expressed using operations from the MLIR standard dialects (see Listing 4).

```
func.func @top(%in: !stream.stream<i64>)
    -> !stream.stream<i64> {
  %out = stream.map(%in) : (!stream.stream<i64>)
      -> !stream.stream<i64> {
  ^0(%val : i64):
    %0 = arith.constant 10 : i64
    %r = arith.addi %0, %val : i64
    stream.yield %r : i64
  }
  return %out : !stream.stream<i64>
}
```

Listing 4: An example of a stream map operation expressed with a mixture of dialects. The computation consumes each input stream element and emits a new stream here all values were increased by the constant 10.

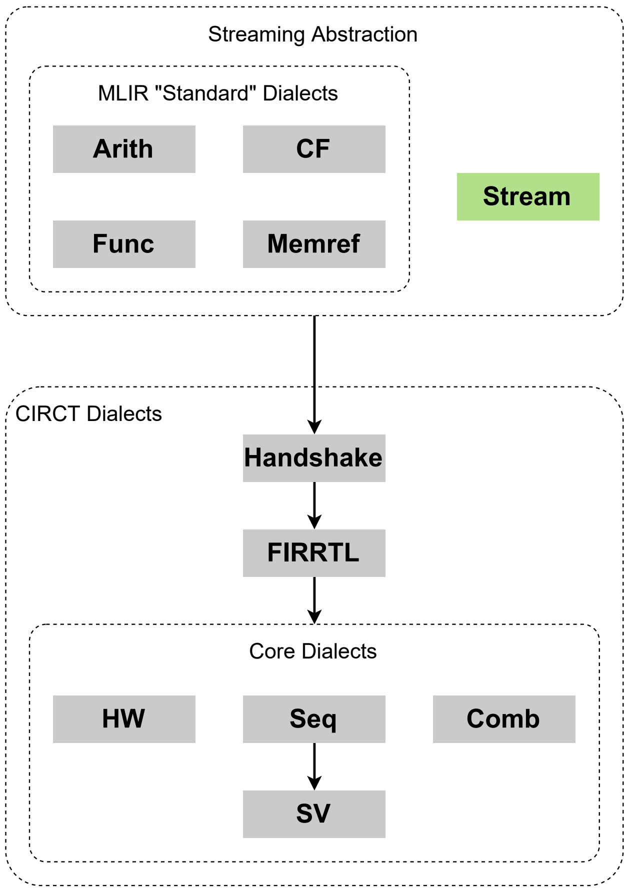


Figure 12: A specialized streaming IR offers both (a) programmer productivity and (b) reliable and efficient hardware generation.

Due to having explicit representations of stream operations, the compiler can apply domain-specific optimizations to simplify certain computations. Furthermore, generic optimizations, e.g., dead code elimination, common sub-expression elimination, constant folding, etc., can be reused for the `stream` and mixed-in MLIR dialects. Finally, canonicalization passes bring dialects into a canonical form to simplify subsequent optimizations and transformations.

Once the streaming abstraction is in the desired shape, e.g., all optimizations were applied, it can be lowered to CIRCT's `handshake` dialect. This transformation reuses substantial parts of CIRCT's existing lowering from the MLIR standard dialects to `handshake` [30]. The details of this non-trivial transformation are explained in Section 4.3. The remaining steps required to transform a representation from the `handshake` dialect down to Verilog was already part of the CIRCT project. The provided example (Listing 4) results in more than 1000 lines of Verilog. The exact size depends on the buffer strategy used (Section 2.3.3.2). Apart from the obvious difference in code size, it must also be mentioned that complicated tasks like scheduling are done by the compiler instead of the user.

4.2 LANGUAGE SPECIFICATION

This section introduces a general streaming abstraction that can be used to express a multitude of streaming applications. Such abstractions are general purpose in the sense that they can be used to not only target FPGAs but CPUs as well. Supporting different target architectures allows applications to be prototyped for one but still be used for the other without relying on complicated simulation tools. Furthermore, it lays the foundation for an abstraction that could be compiled into heterogeneous systems.

The streaming abstraction introduces streams as first-class constructs and defines a set of operations that consume and produce streams. The goal of this abstraction is to encode higher-level semantics without being target device-specific. An operation applied to a stream defines its semantics on a streaming level but hides the details of its execution. As part of this thesis, only the lowering to RTL was implemented, but other backends could be added. The scope of the abstraction includes all basic streaming operations that do not use substantial amounts of memory (see Section 3.6 for a discourse on why memory support was limited).

4.2.1 *Design Decisions*

Before describing the details of the operators, a brief explanation of some basic principles is required. Some of these decisions have a substantial influence on the operators.

First, each stream has only one producing operation and only a single consumer. There are different ways one might implement multiple consumers, e.g., all have to consume an element, round-robin consumption, first come, first served, etc. To avoid any confusion when trying to understand a program or when lowering it, splitting up a stream must be done explicitly.

Additionally, operators are not allowed to allocate memory. The rationale behind this decision is the complication memory causes in a pipelined context (see Section 3.6).

4.2.2 *Types*

Each stream has a corresponding element type that each sent element satisfies. To embed this into MLIR, a value corresponding to a stream has a parametric stream type, where the parameter describes the element type. A stream element can either be an arbitrary but fixed-sized integer or tuples of types. The streaming abstraction, and MLIR, in general, are statically typed. Therefore, each operator has perfect knowledge of the structure of a stream element.

```
element_type ::= "i" num
             ::= "tuple<"element_type ("," element_type)*">"
stream_type  ::= "stream.stream<" element_type ">"
```

Listing 5: Grammar that defines the stream types.

4.2.3 *Lambdas*

Different operations require a lambda parameter to define what the operator has to do in specific steps of execution. A lambda can take multiple inputs and produce multiple outputs. The number and types of both are constrained by the operation that expects a lambda. As the abstraction was implemented on top of MLIR, the supported instructions are all the instructions from the lower-level MLIR dialects (at the time of writing `arith` and `cf`).

A lambda creates its own scope and has no access to values or memory defined elsewhere. This is a much-needed restriction as it allows the compiler to skip alias analysis and enables task pipelining.

4.2.4 *Operations*

In this section, all the supported operators and their semantics are listed. The set of implemented operators covers most use-case that aims to run at max throughput. Each operator expects streams as inputs and produces only streams as outputs. All operators guarantee that inputs and outputs remain in order.

MAP The map operator applies a provided lambda on each incoming value to produce a new return value. The return type can differ from the input type but must match the output stream's element type.

FILTER The filter operator applies the provided lambda on each incoming tuple to decide if the tuple should be forwarded to the output or dropped.

REDUCE The reduce operator applies the lambda on each element in order while providing the result of the previous call to the lambda. The first execution of the lambda is provided with a initial value instead of a previous result. The lambda requires a type of form $(\mathsf{U}, \mathsf{T}) \rightarrow (\mathsf{U})$, where T is the element type of the incoming stream while U is the one of the resulting streams.

The reduce operator only emits a single result once the input stream terminates. Therefore, it is only relevant for finite streams.

SPLIT The split operator can split incoming streams into multiple output streams. It expects a lambda that produces multiple results, each of which will belong to one of the new output streams. For each input element, each output will receive exactly one element.

COMBINE The combine operator is the counterpart to the split operator. It supports a variadic number of input streams and executes a lambda with one element from each input to produce a single output element.

If one of the incoming streams terminates, the output stream will also be marked as complete.

SINK The sink operator consumes all incoming stream elements without producing any output. Not consuming a stream causes back-pressure and thus potentially deadlocks a circuit. A stream without a consumer should ideally not exist in a system, but as streams can be passed as arguments, such cases can occur.

4.2.5 *Registers*

All operations that expect a lambda support registers to keep states between different executions. A register can be added to each operation by defining a "registers" attribute that specifies its initial value (see Listing 6).

```
%out = stream.filter(%in) {registers = [0 : i1]}
  : (!stream.stream<i64>) -> !stream.stream<i64> {
^bb0(%val: i64, %reg: i1):
  %c1 = arith.constant 1 : i1
  %nReg = arith.xori %c1, %reg : i1
  stream.yield %reg, %nReg : i1, i1
}
```

Listing 6: A filter operation that uses a register to drop every second element from the stream.

If an operation declares a register, it has to have the corresponding lambda argument and return value. All of these require the same type as the initial value.

4.2.6 *Execution Guarantees*

While the streaming abstraction does not specify the exact way it will be executed, it provides a set of guarantees:

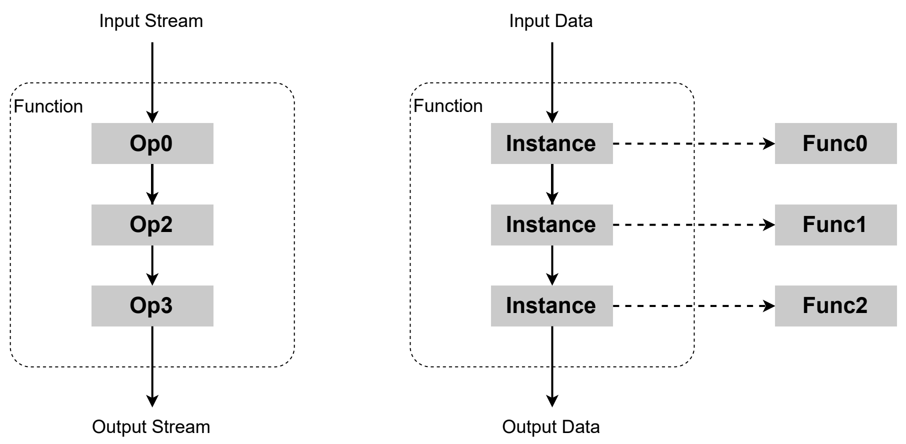
- The order of elements is preserved by each operator. Thus, a resulting stream has the same order as the corresponding inputs.

If a data element is dropped, it will just be omitted from the stream.

- An operation will process an element in isolation, i.e., there are no interferences or interleavings with other data elements except explicitly requested.

4.3 LOWERING TO HANDSHAKE

This section explains the lowering from the streaming abstraction to CIRCT's `handshake` dialect.



(a) A streaming application consisting of three consecutively applied operators on one input stream.

(b) The `handshake` representation corresponding to the streaming application.

Figure 13: Lowering a streaming application moves the operation implementations into separate functions that are instantiated by the top-level.

The conversion from the streaming abstraction to CIRCT's `handshake` dialect follows a simple but effective strategy. Streams are transformed into handshaked signals connecting different functions corresponding to operations. To not clutter the top-level, each operation instance is transformed into a separate function that is only instantiated in the top-level (see Figure 13). The signals connect the different instances in the same manner as in the input.

4.3.1 *Types*

Stream types occur on values produced or consumed by streaming operations. Thus, such a value is a handshaked signal that connects two stream operators. To encode additional information relevant for streams, the datatype of this handshaked signal is not just the stream element type but a tuple of said type and a boolean flag that signals termination. The termination signal marks the end-of-stream (EOS) and is only in a sentinel pulse that carries no data.

4.3.2 *Operations with Regions*

Each operation holding a region expects a collection of input streams and produces a set of output streams. While every operation has different semantics, e.g., filters dropping elements, maps transforming them, and splits producing multiple outputs, their regions are standard MLIR dialects (apart from their terminator). Therefore, the conversion pass reuses CIRCT's existing conversion functionality [30] for said regions. All operations are pipelined and thus use the task pipelining transformation introduced in Chapter 3. The resulting handshake region is inlined into operator-specific `handshake` logic, which provides inputs and consumes the outputs. The reused lowering exposes an additional control in and outputs that stem from the structured lowering approach introduced by Dynamatic. In our case, these control signals are forged using the stream's data signal to activate a `join` operation.

```
%out = stream.filter(%in) : (!stream.stream<i32>)
    -> !stream.stream<i32> {
^bb0(%val: i32):
  %c0_i32 = arith.constant 0 : i32
  %0 = arith.cmpi sgt, %val, %c0_i32 : i32
  stream.yield %0 : i1
}
```

Listing 7: A simple filter operation dropping all inputs equal to 0.

For example, a `filter` operation that drops all stream elements with value 0 (see Listing 7), will produce a circuit as shown in Listing 8. Due to `handshake` reusing parts of the `arith` dialect, the input region can still be spotted.

END-OF-STREAM SIGNAL As previously mentioned, each stream has a corresponding end-of-stream signal. If a data element arrives at an operator and EOS is signaled, the data element is ignored, and the operator's computation has to be skipped (see Listing 8). Due to the data value being undefined, it cannot be guaranteed that the element will satisfy any implicit preconditions of the region. Causing a circuit to fall into an infinite loop or even just having very high latency is not desired. Thus, all computation has to be skipped on EOS. Apart from that, the stream operator needs to be reinitialized, e.g., register values.

```
handshake.func private @stream_filter(%arg0: tuple<i32, i1>)
      -> tuple<i32, i1> {
  %data, %eos = unpack %arg0 : tuple<i32, i1>

  // Only forward input when eos == 0
  %trueResult, %falseResult = cond_br %eos, %data : i32
  // Produce the necessary control signal
  %1 = join %falseResult : i32
  // Region start
  %2 = constant %1 {value = 0 : i32} : i32
  %3 = arith.cmpi sgt, %falseResult, %2 : i32

  // Drop element only when eos == 0, otherwise the data
  // field is either way ignored.
  %trueResult_0, %falseResult_1 = cond_br %eos, %eos : i1
  // Determine the dropping condition. If eos == 1, the
  // region was not executed, thus eos itself is the condition.
  %4 = mux %eos [%3, %trueResult_0] : i1, i1
  // Only forward data then needed
  %trueResult_2, %falseResult_3 = cond_br %4, %arg0
      : tuple<i32, i1>
  return %trueResult_2 : tuple<i32, i1>
}
```

Listing 8: The `handshake` representation of a stream filter operation drops all inputs with value 0 (omitting forks and sinks for brevity). When EOS, the second element of the input tuple `%arg0`, is true, the computation is skipped.

In the case of `reduce`, the data element carrying EOS is ignored, but the accumulated value is emitted before EOS is forwarded in a separate stream element.

REGISTER LOWERING Registers keep states between different executions of an operation's region. Therefore, a loop-back from the output setting the new value back to the input must be constructed. A one-element buffer to break the combinatorial cycle and to provide an initial value is inserted (see Figure 14).

Note that this construction implicitly produces sequential dependencies between different stream element computations. The dynamically scheduled nature of the abstraction ensures correctness but the potentially high throughput penalty needs to be kept in mind. Even though the circuits support pipelining, this is not helpful in such a scenario, as there is an explicit data dependency, which will stall the subsequent pipeline invocations.

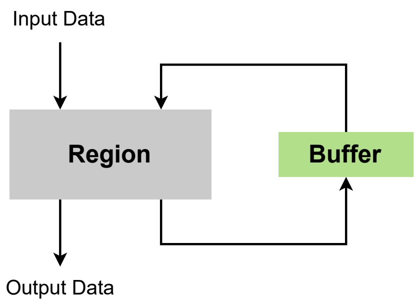


Figure 14: A register is implemented as a buffer that is inserted within a loop from the registers output to the input.

Similar to how data elements are prevented from entering the region when EOS is fired, registers will flush the corresponding buffers without forwarding the value into the region. In the case of EOS, the buffer is refilled with the initial value instead of the lambda's output.

The `reduce` operation has a special accumulation register that is treated similarly to register. Instead of just sinking the data when EOS is consumed, it forwards it to its output.

4.4 INTERACTION WITH EXTERNAL WORLD

The stream abstraction does not provide any operation that produces new streams from nothing. All data to work with must be streamed into the produced circuitry. We simplify the necessary integration into existing systems by implementing the stream compatible with the AXI4 interface. As stated previously (Section 4.3.1), a stream corresponds to a handshaked tuple consisting of data and an EOS element. Thus, the interface directly provides means to transfer data and terminate connections. Additional AXI4 signals, e.g., keep, can be encoded by changing the datatype to contain the necessary wires.

Apart from the provided easy-to-use stream interface, it is worth mentioning that it is possible to integrate external components into a streaming application. Streams can be connected to external components, and their results produce streams. Providing such functionality allows complex applications to use the streaming abstraction while implementing critical features in RTL.

Compared to other HLS approaches, our work produces pluggable circuits that can be used without additional integration costs.

5 EVALUATION

This chapter elaborates on the different kinds of benchmarks and use cases we implemented to demonstrate the capabilities of this work. It starts off by showing the impact of the task pipelining transformations with a set of micro-benchmarks. These benchmarks implement simple circuits and compare their throughput for different strategies to show that we outperform existing approaches' throughputs. Afterward, we present different streaming applications that can be constructed within our streaming abstraction and evaluate their performance when executed in hardware. We do this to demonstrate that we simplify the development of streaming applications while still getting optimal throughputs.

5.1 BENCHMARKING SETUP

The goal of the benchmarks is to show that our work can be used in existing systems. All the following benchmarks were tested with simulators beforehand to check if they work correctly. While executing circuits in simulations can show correct behavior, it is not architecture-specific and might thus hide some relevant hardware issues. To ensure our work has no hidden problem, we plug the generated circuits into Coyote [23], which provides AXI4 streaming interfaces to connect. As described in Section 4.4, the generated stream interfaces are very similar to AXI4. Thus, such an integration works seamlessly.

We execute the benchmarks on a cluster [16] with a Xilinx Alveo U55C Card [34]. The input data was streamed from a host CPU to the FPGA through PCIe, which corresponds to a classical accelerator setup. Due to the streaming interfaces, our work could easily be used in other settings, but this would not change our results.

All benchmarks use a 512-bit data stream to saturate the link. Furthermore, all the generated circuits reached the timing requirements for the 300 MHz clock. As our work reuses existing lower levels that do not implement area optimal buffering, we do not measure the area and resource usage.

5.2 TASK PIPELINING MICRO-BENCHMARKS

To investigate the impact of the task pipelining transformations in isolation, we constructed micro-benchmarks that contain patterns that match the different strategies. These micro benchmarks were implemented for CIRCT's existing HLS flow and thus do not use any streaming features.

5.2.1 *A feed-forward CFG*

```
int64_t compute(int64_t val) {
  int64_t res;
  if ((val >> 1) & 0x1) {
    if ((val >> 2) & 0x1) {
      res = val + 1;
      res += 1;
      res += 1;
      res += 1;
    } else {
      res = val + 10;
    }
  } else
    res = val;
  return res;
}
```

Listing 9: A C representation of the program we used to evaluate the throughput impact for feed-forward graphs. As the code is not optimized, the branches have different latencies.

The first benchmark implements a diamond CFG that has paths with different latencies (Listing 9). Apart from comparing the locked to the task pipelined circuits, we also tested a simple loop back. The loopback directly connects in and output AXI4 streams and serves as a lower bound for the latency. The data streams triggered both the fastest and the slowest path in 50% of the cases.

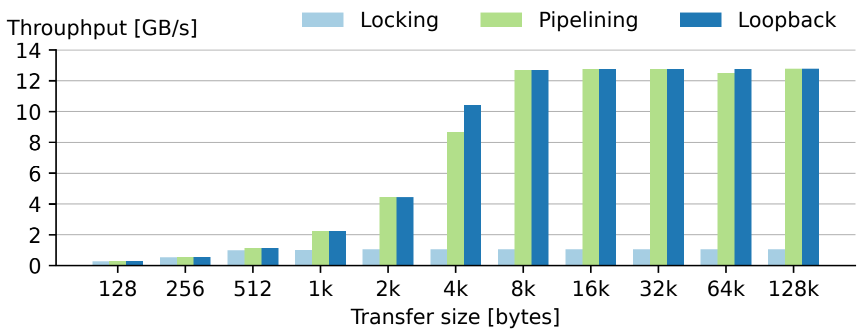


Figure 15: The throughputs of the different task pipelining protection mechanisms in a feed-forward case. The pipelined execution reaches the same throughput as a loopback. Locking, on the other hand, has to stall due to the latency of one element.

The results show that the task pipelining transformation runs at line rate, i.e., the optimal throughput that cannot be outperformed by any other implementation. On the other hand, the locked circuit pays a substantial throughput penalty (Figure 15). It is important to stress that previous DHLS enforces a locking, as it otherwise produces incorrect results. Compared to the PCIe roundtrip time, the task pipelined latency of a few clock cycles is negligible (Figure 16). On the other hand, locking reduces the throughput and thus causes back pressure, which increases the latency substantially.

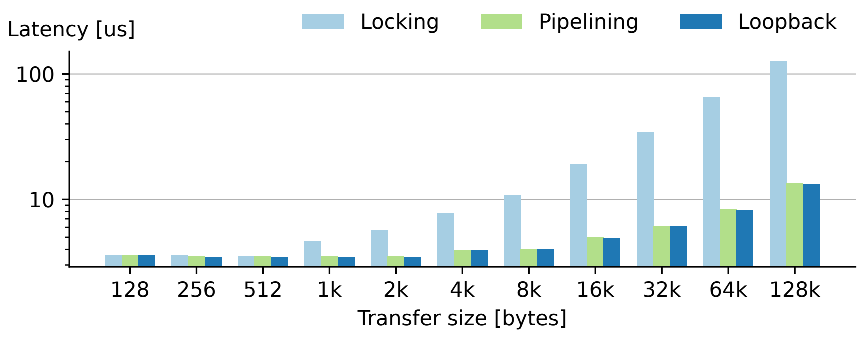


Figure 16: Latencies of the different task pipelining protection mechanisms in a feed-forward case. The throughput of the pipelined circuit is negligible compared to PCIe's overhead. Locking, on the other hand, is substantially slower.

5.2.2 *Multiple Loops*

As a second micro benchmark, we demonstrate the impact of locking loops instead of functions. The implemented program consists of three consecutive for-loops that all sequentially depend on each other (Listing 10).

```c
int64_t loop_sequence(int64_t v) {
  for (int i = 0; i < 3; ++i)
    v = (v ^ 0x1234) ^ (v >> 16);
  for (int i = 0; i < 3; ++i)
    v = (v ^ 0x1234) ^ (v >> 16);
  for (int i = 0; i < 3; ++i)
    v = (v ^ 0x1234) ^ (v >> 16);
  return v;
}
```

Listing 10: A C representation of the program we used to evaluate the impact of pipelining different loops.

Each loop iteration and thus each loop has the same latency. Therefore, the naive approach of locking the entire function is 3 times slower than locking each loop separately (Figure 18). While the pipelining approach only allows one thread per loop, it still allows one thread per loop. Therefore, the pipeline's II is a third of the function locking, which increases the pipeline's throughput accordingly (Figure 17).

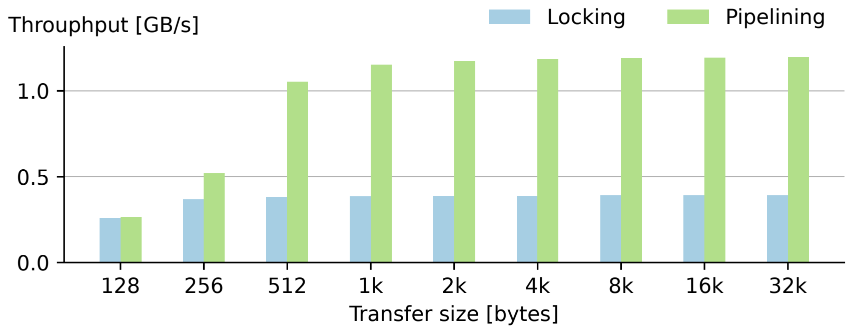


Figure 17: The throughput of the loop-wise locking, which enables some pipelining, is 3 times higher than locking the entire function. Locking the loops causes a large II and thus does not reach the loopback's throughput.

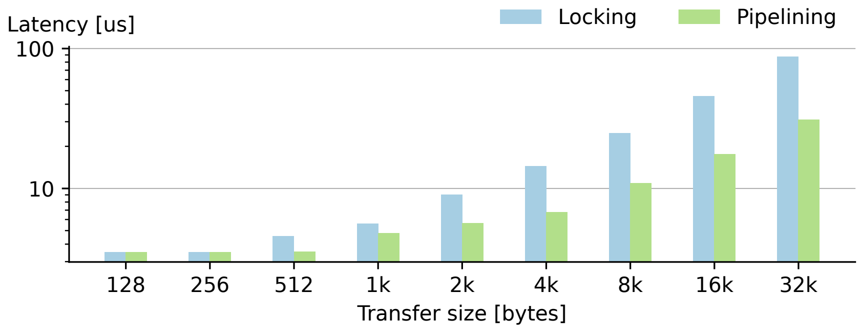


Figure 18: The latency of the loop-wise locking is 3 times lower than locking the entire function.

5.3 STREAMING USE-CASES

This section presents a set of different streaming applications we constructed, compiled, synthesized, and finally executed in hardware. All our test cases reach the maximum throughput of the PCIe connection, which is the theoretical limit. Even handwritten RTL cannot outperform the circuits generated by our HLS flow.

5.3.1 *Note on Buffering*

As already discussed (Section 2.3.3.2), the buffering problem for sequential circuits was already resolved in other works [22]. Unfortunately, the CIRCT project did not yet reimplement the optimal buffering strategy. Therefore, CIRCT only provides basic buffering strategies that do not support high throughput for any circuits.

To circumvent this issue, we implemented a custom buffering pass that does not insert additional buffers in cycles involving a register. In all other places, the strategy inserts buffers. As long as the computations involving a register can fit into a single clock cycle, this strategy will guarantee a latency of 1 clock cycle. Having a latency of just 1 cycle does not cause any throughput penalty.

The buffers inserted in feed-forward graphs are a sequence of a one-element sequential buffer to break combinational dependencies and a multi-element FIFO. While this might over-provision buffer space and thus incur a higher area usage, it yields the same throughput as the optimal buffering strategy. So, our benchmarks' throughput should not be affected by using a non-optimal buffering strategy.

5.3.2 *Statistics Computation*

The first streaming application we implemented demonstrates the bump-in-the-wire usage of our work (Figure 19). In essence, such use cases plug in a computation on an existing data stream without introducing slowdowns for the data streams. Such computations perform in flight monitoring or metadata collection, e.g., with sketch algorithms [7, 24].

This benchmark consumes a stream and returns it again while accumulating the maximal value it observed. The data type is an eight-tuple of 64-bit integers. As there are eight input values, the maximum of these eight values has to be determined first. Afterward, the remaining element is streamed into a reduction that only stores the maximum observed over time. Once an EOS signal is received, the maximum is emitted, and all state is reset.

The cycle caused by the reductions register must have low latency to reach high performance. Therefore, we use the buffering strategy that does not add additional registers in that cycle and keeps the latency at 1 clock cycle. Thus, the produced circuit can reach peak throughput (Figure 20).

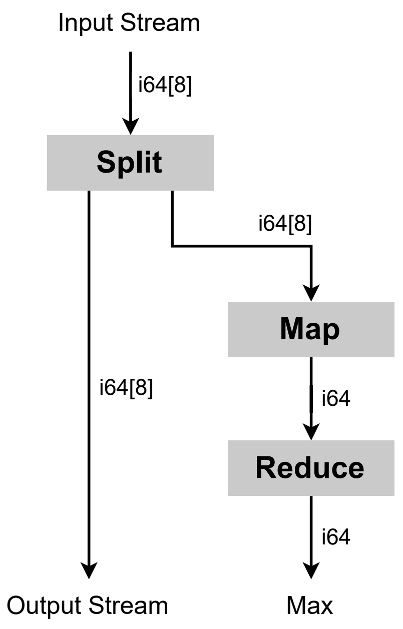


Figure 19: A streaming application that computes the maximum value of an incoming stream.

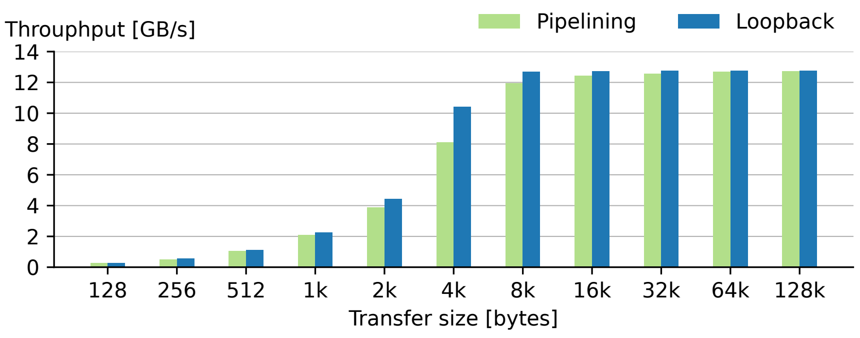


Figure 20: The throughput of the maximum computation. As desired, the produced circuits reach peak throughput and thus do not cause any back-pressure on the link.

5.3.3 *Query Offloading*

As a second use case, we evaluate the usage of the streaming work for query offloading tasks. Data transfer is starting to become a bottleneck for data processing applications that work with substantial amounts of data. Networks and buses have limited bandwidth. Therefore, dropping unneeded data as soon as possible is beneficial, i.e., close to the data source. This concept is similar to projection and predicate pushdown in database query optimization [33]. In large systems, selections and projections (in streaming terms maps and filters) can be performed close to the storage [32].

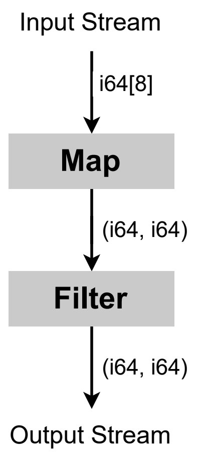


Figure 21: A streaming application that semantically performs a select and project database query.

```
SELECT (t0 + t1) AS a, (t4 + t5) AS b
  FROM stream IF a <= b;
```

Listing 11: A pseudo SQL query which is equivalent to the stream computation.

The implemented streaming program (Figure 21, which is equivalent to Listing 11), performs a projection (map) followed by a selection (filter) and thus only produces outputs that match the desired properties. The generated circuits reach an II of one. Therefore, their throughput, once saturated, reaches the peak of 100 Gbit/s (Figure 22).

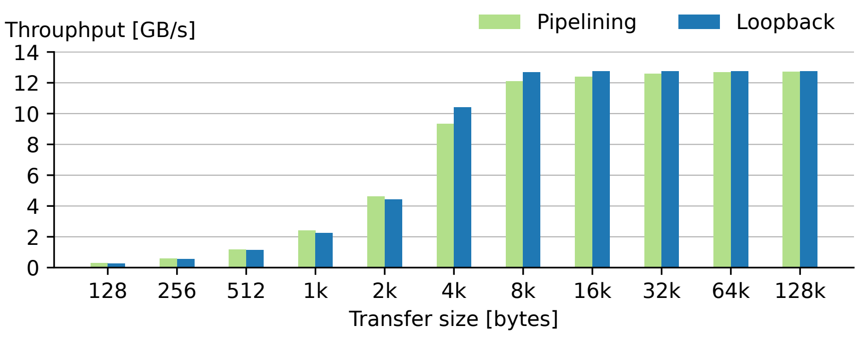


Figure 22: The throughput of the query style computation. As desired, the generated circuits reach peak throughput and thus do not cause any back-pressure on the link.

5.3.4 *Interaction with External Components*

Some functionalities cannot be implemented with the stream feature set implemented in this work. Therefore, it can be necessary

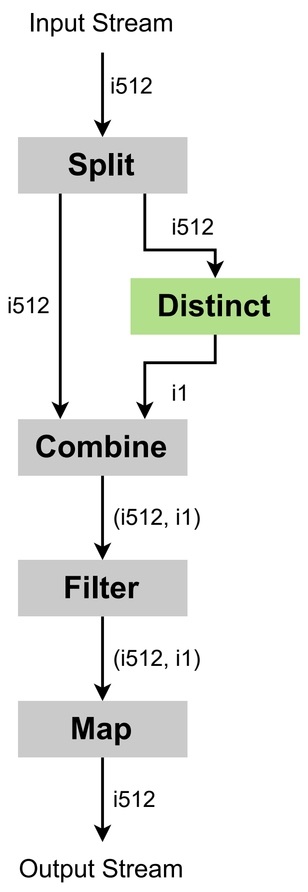


Figure 23: A streaming application that performs a distinct query with the usage of an external component (green).

to interface with external components. To demonstrate this capability (described in Section 4.4), we implemented a stream application that uses an external component that detects duplicates. The external functionality can be plugged in to build an SQL distinct query (Figure 23, equivalent to Listing 12).

```
SELECT DISTINCT FROM stream;
```

Listing 12: A SQL distinct query which is equivalent to the stream computation.

This streaming application, which ran a line rate in hardware Figure 24, demonstrates that our work can interact with external components. These results show that the streaming abstraction, with some additional components, can execute complex queries directly on the data streams without causing noticeable latency. The handling of variable latencies while enabling interfacing with arbitrarily complex hardware components makes this work a promising candidate for streaming use cases.

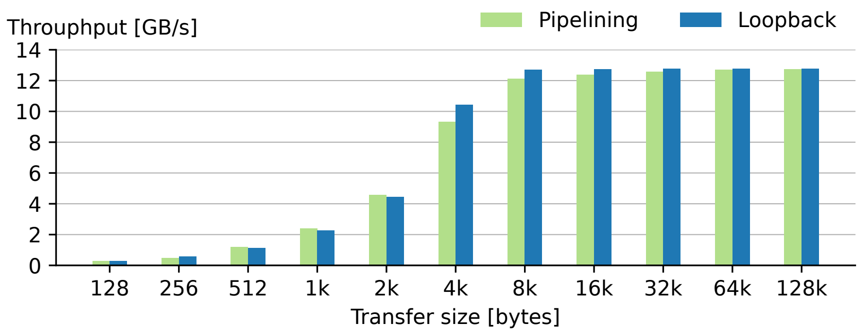


Figure 24: The throughput of the distinct computation. As desired, the produced circuits reach peak throughput and thus do not cause any back-pressure on the link.

6 CONCLUSION

This work presented an extension to existing DHLS flows that allows it to model pipelined computations. The discussion of the DHLS extension, in the form of a safety transformation, clearly points out existing problems when one wants to reach high performance. Reducing the II of circuits with pipelining is an effective way of increasing the throughput of a hardware circuit while also increasing the temporal utilization. Apart from providing throughput improvements for existing DHLS flows, the task pipelining transformation lays the foundation for alternative HLS approaches to reuse the DHLS dataflow abstraction. To implement this extension, substantial parts of CIRCT's `handshake` dialect and the surrounding transformations were repaired, modified, and extended. All these changes summed up to 9000 added and 4000 removed lines of code and were upstreamed in 59 commits to the CIRCT GitHub repository.

We demonstrate the reusability by providing a general-purpose streaming abstraction that targets this DHLS abstraction. The streaming abstraction is implemented in a general-purpose compiler framework. Therefore, it can be targeted by other compilation flows and target different devices, e.g., CPUs.

Furthermore, the streaming abstraction demonstrates that constructing a DSL for HLS can bring benefits to HLS compilers. In contrast to C-style HLS, the input language does not implicitly limit the kinds of inputs it can handle. It provides no way of expressing programs that do not match its execution model. This explicit encoding helps both the developers and the compiler. The developers will not e surprised by drastic optimization blockers, while the compiler can skip potentially expensive analysis.

Finally, the generated circuits can be plugged into existing systems due to supporting AXI4 interfaces and showing peak throughput. Both properties are essential for systems integration, as there is no integration cost in terms of interfacing or performance. Still, the abstraction simplifies stream programming for FPGAs drastically.

6.1 FUTURE WORK

This section lists potential future work that builds on top of the contributions of this thesis.

6.1.1 *Building a Frontend*

This work defines an abstraction that encodes streaming applications, but it lacks a non-verbose input language. Even though MLIR dialects have pretty printing, their representations remain somewhat verbose and thus harder to write than necessary. This abstraction should be connected to a DSL or an existing language or framework to simplify developing streaming applications.

6.1.2 *Other compilation Backends*

During this work, we focused only on compiling the streaming abstraction to RTL to execute programs in FPGAs. As the abstraction is part of the MLIR ecosystem, there should be no fundamental problems in adding other backends. While streaming might not be perfectly suited to run on CPUs, there are use cases where this is still desirable. Different backends could also be interesting when large stream applications should be executed on heterogeneous systems that have both FPGAs and CPUs, e.g., Enzian [8].

6.1.3 *Extending the Streaming Abstraction*

The implemented abstraction does not claim feature completeness, as the set of stream operators is large. For example, operations that create sliding windows would be helpful but semantically not that simple. On the other hand, splitting and combining operations that only emit, respectively consume, from one of N streams would be simple to implement.

Apart from adding language features, some optimizations could be implemented. Optimizations on a higher level can eliminate unneeded computations or increase performance. While this was not necessary for our workloads, there might be cases where area constraints are a concern.

BIBLIOGRAPHY

[1] Frances E. Allen. "Control Flow Analysis." In: *SIGPLAN Not.* 5.7 (1970), 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: https://doi.org/10.1145/390013.808479.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: Constructing Hardware in a Scala Embedded Language." In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: Association for Computing Machinery, 2012, 1216–1225. ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. URL: https://doi.org/10.1145/2228360.2228584.

[3] 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 (2013). ISSN: 1539-9087. DOI: 10.1145/2514740. URL: https://doi.org/10.1145/2514740.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine." In: *IEEE Data Eng. Bull.* 38 (2015), pp. 28–38.

[5] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. "Combining Dynamic & Static Scheduling in High-Level Synthesis." In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 288–298. ISBN: 9781450370998. DOI: 10.1145/3373087.3375297. URL: https://doi.org/10.1145/3373087.3375297.

[6] Jianyi Cheng, John Wickerson, and George A. Constantinides. "Dynamic C-Slow Pipelining for HLS." In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022, pp. 1–10. DOI: 10.1109/FCCM53951.2022.9786096.

[7] Monica Chiosa, Thomas B. Preußer, and Gustavo Alonso. "SKT: A One-Pass Multi-Sketch Data Analytics Accelerator." In: *Proc. VLDB Endow.* 14.11 (2021), 2369–2382. ISSN: 2150-8097. DOI: 10.14778/3476249.3476287. URL: https://doi.org/10.14778/3476249.3476287.

[8] David Cock et al. “Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research.” In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’22. Lausanne, Switzerland: Association for Computing Machinery, 2022, 434–451. ISBN: 9781450392051. DOI: 10.1145/3503222.3507742. URL: https://doi.org/10.1145/3503222.3507742.

[9] The CIRCT Community. *CIRCT: Circuit IR Compilers and Tools*. https://circt.llvm.org/. Accessed: 2022-09-21.

[10] The LLVM Community. *LLVM Loop Terminology (and Canonical Forms)*. https://llvm.org/docs/LoopTerminology.html. Accessed: 2022-10-12.

[11] J. Cortadella, M. Kishinevsky, and B. Grundmann. “Synthesis of synchronous elastic architectures.” In: *2006 43rd ACM/IEEE Design Automation Conference*. 2006, pp. 657–662. DOI: 10.1145/1146909.1147077.

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

[13] Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. “Compositional Dataflow Circuits.” In: *ACM Trans. Embed. Comput. Syst.* 18.1 (2019). ISSN: 1539-9087. DOI: 10.1145/3274280. URL: https://doi.org/10.1145/3274280.

[14] Joseph A. Fisher. “Very Long Instruction Word Architectures and the ELI-512.” In: *SIGARCH Comput. Archit. News* 11.3 (1983), 140–150. ISSN: 0163-5964. DOI: 10.1145/1067651.801649. URL: https://doi.org/10.1145/1067651.801649.

[15] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. “Hardware system synthesis from Domain-Specific Languages.” In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927454.

[16] ETH Systems Group. *Heterogeneous Accelerated Compute Cluster*. https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html. Accessed: 2022-10-14.

[17] Theo Haerder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery.” In: *ACM Comput. Surv.* 15.4 (Dec. 1983), 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[18] *Intel High-Level Synthesis (HLS) Compiler*. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. Accessed: 2022-03-31.

[19] A. Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations." In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[20] Lana Josipovic, Philip Brisk, and Paolo Ienne. "An Out-of-Order Load-Store Queue for Spatial Computing." In: *ACM Trans. Embed. Comput. Syst.* 16.5s (2017). ISSN: 1539-9087. DOI: 10.1145/3126525. URL: https://doi.org/10.1145/3126525.

[21] Lana Josipović, Radhika Ghosal, and Paolo Ienne. "Dynamically Scheduled High-Level Synthesis." In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '18. Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, 127–136. ISBN: 9781450356145. DOI: 10.1145/3174243.3174264. URL: https://doi.org/10.1145/3174243.3174264.

[22] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. "Buffer Placement and Sizing for High-Performance Dataflow Circuits." In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, 186–196. ISBN: 9781450370998. DOI: 10.1145/3373087.3375314. URL: https://doi.org/10.1145/3373087.3375314.

[23] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. "Do OS abstractions make sense on FPGAs?" In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 991–1010. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/roscoe.

[24] Amit Kulkarni, Monica Chiosa, Thomas B. Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. "HyperLogLog Sketch Acceleration on FPGA." In: *CoRR* abs/2005.13332 (2020). arXiv: 2005.13332. URL: https://arxiv.org/abs/2005.13332.

[25] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[26] 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.

[27] Johannes Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. "Transformations of High-Level Synthesis Codes for High-Performance Computing." In: *IEEE Transactions on Parallel and Distributed Systems* 32 (May 2021), pp. 1014–1029. DOI: 10.1109/TPDS.2020.3039409.

[28] ARM Ltd. *AMBA AXI-Stream Protocol Specification*. https://developer.arm.com/documentation/ihi0051/latest. Accessed: 2022-09-21.

[29] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. "A Compiler Infrastructure for Accelerator Generators." In: *CoRR* abs/2102.09713 (2021). arXiv: 2102.09713. URL: https://arxiv.org/abs/2102.09713.

[30] Morten Borup Petersen. *A Dynamically Scheduled HLS Flow in MLIR*. Tech. rep. 2022.

[31] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, 12–27. ISBN: 0897912527. DOI: 10.1145/73560.73562. URL: https://doi.org/10.1145/73560.73562.

[32] Muthian Sivathanu, Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Database-Aware Semantically-Smart Storage." In: *FAST*. Vol. 5. 2005, p. 18.

[33] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I*. USA: Computer Science Press, Inc., 1988. ISBN: 088175188X.

[34] AMD Xilinx. *Alveo U55C High Performance Compute Card*. https://www.xilinx.com/products/boards-and-kits/alveo/u55c.html. Accessed: 2022-10-14.

[35] AMD Xilinx. *Vitis Unified Software Platform*. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/nmc1504034362475.html. Accessed: 2022-09-21.

[36] AMD Xilinx. *Vitis Unified Software Platform*. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html. Accessed: 2022-09-21.

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets." In: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. Boston, MA: USENIX Association, June 2010. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.