Invited Tutorial
Dynamatic: From C/C++ to Dynamically Scheduled Circuits

Lana Josipović, Andrea Guerrieri, and Paolo Ienne
École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

ABSTRACT

High-level synthesis tools, both commercial and academic, typically rely on static scheduling to produce high-throughput pipelines. However, in applications with unpredictable memory accesses or irregular control flow, these tools need to make pessimistic scheduling assumptions. In contrast, dataflow circuits implement dynamically scheduled circuits, in which components communicate locally using a handshake mechanism and exchange data as soon as all conditions for a transaction are satisfied. Due to their ability to adapt the schedule at runtime, dataflow circuits are suitable for handling irregular and control-dominated code. This paper describes Dynamatic, an open-source HLS framework which generates synchronous dataflow circuits out of C/C++ code. The purpose of this paper is to give an introductory overview of Dynamatic and demonstrate some of its use cases, in order to enable others to use the tool and participate in its development.

ACM Reference Format:
Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2020. Invited Tutorial Dynamatic: From C/C++ to Dynamically Scheduled Circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3373087.3375391

1 INTRODUCTION

Dynamatic is an academic, open-source high-level synthesis compiler that produces synchronous dynamically-scheduled circuits from C/C++ code. Dynamatic generates synthesizable RTL and delivers significant performance improvements compared to state-of-the-art commercial HLS tools in specific situations (e.g., applications with irregular memory accesses or control-dominated code). The fully automated compilation flow of Dynamatic is based on LLVM. In its current implementation, it directly targets Xilinx FPGAs. Dynamatic is customizable and extensible to target different hardware platforms and it is easy to use with commercial tools such as Vivado (Xilinx) and Modelsim (Mentor Graphics).

This paper serves as supporting material for the Dynamatic tutorial. The purpose of the tutorial is to introduce the tool and demonstrate some of its use cases. Tutorial participants can expect to (1) understand when, why, and how dynamically schedules outperform static schedules, (2) learn how to use Dynamatic to produce dynamically-scheduled circuits from high-level languages such as C/C++, and (3) understand the basic structure of the tool and its internal file formats so as to use it for research purposes or to contribute to its development.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02.
https://doi.org/10.1145/3373087.3375391

2 DYNAMICALLY SCHEDULED HLS

In this section, we provide some background on dynamic scheduling and dataflow circuits. We describe the dataflow components used in Dynamatic and give an overview of the strategy for creating functionally correct dataflow circuits out of high-level code. We then discuss the buffering of dataflow circuits as well as their out-of-order memory interfaces.

2.1 Dataflow Circuits

Latency-insensitive protocols implement dynamically scheduled dataflow circuits, built out of dataflow components which use a handshake mechanism to exchange pieces of data, conventionally referred to as *tokens*. Dyanamatic's protocol uses two signals: one indicates the availability of a new token from the source component whereas the other indicates the readiness of the target component to accept it, as indicated in Figure 1. In contrast to a predetermined, centralized controller of statically scheduled circuits, this distributed control system enables dataflow circuits to adapt the schedule at runtime to variable latencies of particular memory access patterns and control-flow decisions.

2.2 Dataflow Components

To implement latency-insensitivity, all standard datapath components (representing program instructions) communicate with their predecessors and successors using bidirectional control signals, as described in the previous section. In addition to standard functional units, dataflow circuits require specialized components which control the flow of data between components. Dynamatic employs the following dataflow components, depicted in Figure 1:

- An *eager fork (fork)* replicates every token received at the input to multiple outputs; as soon as one successor is ready to accept the token, the fork sends it to the successor; however, the fork can accept a new token only after all successors have accepted the previous one.
- A *lazy fork (lfork)* has the same functionality as the eager fork; however, it distributes a token to all successors at once (i.e., all successors must be ready for the lazy fork to send the token).
- A *join* acts as a synchronizer—its output is triggered only after all of its inputs become available.
- A *merge* is a nondeterministic component which propagates a token received on any of its input to its single output.
- A *mux* is a deterministic version of the merge; it propagates to its single output the input token selected by a control input.

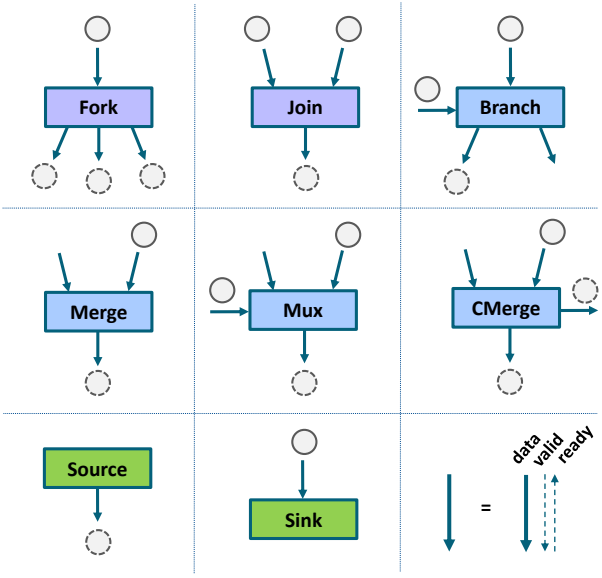

Figure 1: Dataflow components.

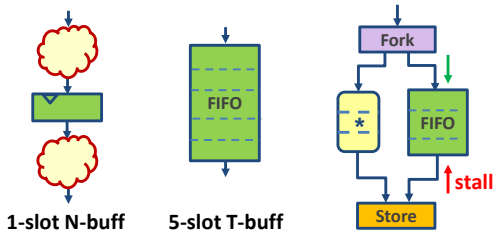

Figure 2: A nontransparent (N-buff) and a transparent (T-buff) buffer. The rightmost figure shows the role of transparent buffers (i.e., FIFOs) in mitigating backpressure.

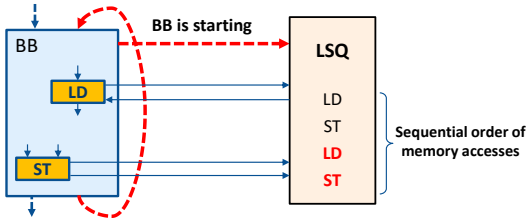

Figure 3: Connecting a dataflow circuit to a load-store queue.

- A *control merge (cmerge)* is a merge which, apart from the data output, has an output which indicates which of the inputs was taken by the merge.
- A *branch* implements program control-flow statements; it dispatches a token received at its single input to one of its multiple outputs based on a condition.
- A *source* is always ready to issue tokens to its single successor.
- A *sink* is always ready to consume tokens from its single predecessor.

2.3 Transforming Imperative Code into a Dataflow Circuit

This section informally describes a way to transform a standard IR into a functionally correct dataflow circuit. Formal details on correctness and liveness can be found in previous work [6].

The programs we consider are organized into sections corresponding to basic blocks (BBs), i.e., pieces of code with no conditionals. All control flow statements are implemented between the BBs and each BB contains a dataflow graph (DFG) of program instructions.

Implementing control flow. To guarantee that data is always accompanied by control, the following must hold: (1) every BB must provide data for every immediate successor BBs and exclusively to them, and (2) every BB must receive data from its immediate predecessor BBs and exclusively from them. Hence, every BB liveout must be sent to the immediate successors using branch nodes; every BB livein must be injected into a BB through a mux node, with as many data inputs as there are predecessor BBs. This strategy guarantees that every piece of data is sent correctly from BB to BB, following the control flow of the program.

In-order control network. Some operations do not have any inputs (e.g., constants); we must ensure that they are appropriately triggered and executed. Furthermore, a mux may receive multiple inputs at the same time; we need to ensure that the inputs are accepted in order of program execution. To this end, we generate an in-order control path that follows the control flow of the program through the BBs—essentially, a data-less variable which is a live-in and live-out of each and every BB. The tokens on this path are used to trigger operations without inputs as many times as their BB becomes active. This path enters each BB through a cmerge, which connects to the muxes of the same BB and indicates the ordering of the inputs from which they will receive data.

Constructing the datapath. Once the control flow is correctly handled, the BB internals are straightforward to design—each instruction is literally converted into its dataflow component (i.e., functional units with inputs and outputs accompanied by handshake signals). As every data exchange must be represented with an explicit token transfer (i.e., handshake exchange), components with multiple successors require a fork to replicate the output token into a token for each of the successors. Unused component outputs (e.g., branch outputs without successors) connect to sinks which discard the unused tokens.

We will illustrate all these features with an example in the following sections.

2.4 Buffer Insertion

Dataflow circuits require buffers which serve as registers in standard synchronous designs. As in any synchronous circuit, all combinational cycles of a dataflow circuit must contain at least one buffer. Yet, in contrast to standard registers, buffers can be placed on any channel of a dataflow circuit without compromising its functionality. However, they impact the following timing aspects: (1) *Critical path.* As standard registers, buffers can be used to break

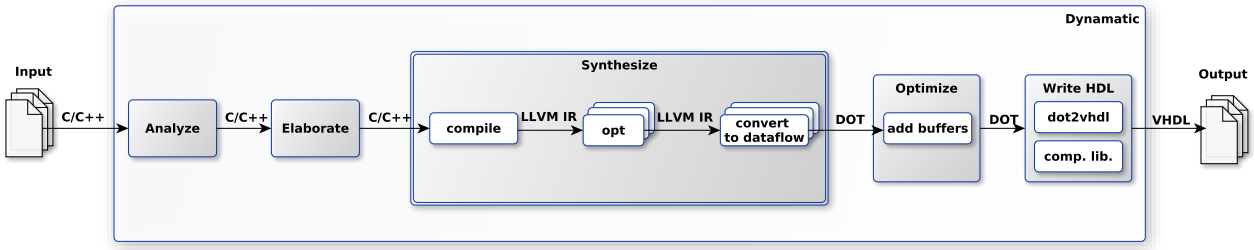

Figure 4: Dynamatic HLS compiler: software-to-hardware flow.

combinational paths of the circuit into two paths, possibly reducing the critical path of the circuit. (2) *Throughput.* In dataflow circuits, some paths may take a longer time to process data and prevent the faster paths from consuming tokens at a higher rate, hence lowering the throughput of the system. This effect can be mitigated using buffers of larger sizes (i.e., FIFOs) to accumulate tokens and relieve backpressure from the predecessor components, as illustrated in the right of Figure 2.

Dynamatic employs a performance optimization model [7] which allows for resource-optimal buffer placement and sizing, with the purpose of maximizing throughput at the desired clock frequency. The buffers are placed on the edges between components and are characterized with two properties: (1) transparency, which indicates whether a buffer adds sequential delay onto a path (a nontransparent buffer is used to break the combinational delay and implies a 1-cycle latency, whereas a transparent buffer is implemented as a pass-through element and does not increase cycle count), and (2) capacity (i.e., number of slots), which is used to regulate throughput. Examples of buffers with different properties are shown in Figure 2. The buffer placement strategy employed by Dynamatic inserts such buffers to achieve high-throughput, pipelined designs which meet the desired clock period.

2.5 Memory Interfaces

If memory dependences cannot be determined at compile time, dataflow circuits rely on load-store queues (LSQs), similar to those present in dynamically scheduled processors, to resolve the dependences dynamically, as the circuit runs. Yet, the LSQ of a dataflow circuit has a fundamental requirement: it must be given explicit information on the original program order of the memory accesses, so that it can allocate them into the queue in the right order and thus resolve them in a semantically correct way [5]. A way to provide this information is to send to the LSQ tokens which follow the actual order of execution of the basic blocks of the circuit. This ordering enables the LSQ to determine and resolve dependences as memory access arguments from different BBs arrive out-of-order. For this purpose, we use the control-only path described in Section 2.3—the path forks into the LSQ from each BB which has load and store instructions which connect to it; it triggers the allocation of BBs as soon as the control flows there (i.e., as soon as a decision has been made to enter a particular BB). This mechanism is illustrated in Figure 3.

LSQs allow dataflow circuits to execute memory accesses out-of-order and to achieve high performance in situations which static scheduling cannot efficiently handle. However, on FPGAs, LSQs imply high resource requirements as well as power and clock degradation. Hence, Dynamatic leverages compiler analysis to simplify the memory interface—whenever the compiler can disambiguate memory accesses, groups of accesses that cannot mutually conflict use separate LSQs, while accesses which certainly have no dependences with any other accesses are connected to simple memory interfaces [3].

3 DYNAMATIC: THE BASIC FLOW

The basic flow of Dynamatic is depicted in Figure 4. Dynamatic takes as input C or C++ code and produces a synthesizable hardware description of the corresponding dataflow circuit.

The first two steps of the flow, analysis and elaboration, preprocess the C files by prechecking code correctness, adding metainformation, and formatting it for the rest of the flow. The synthesis step relies on the LLVM compiler framework [8]: the clang frontend parses the C/C++ program and produces a static single assignment intermediate representation (LLVM IR), which is then optimized using standard LLVM transformation and analysis passes. The optimized IR is then given as input to a set of our custom passes. The main pass adds dataflow components from Section 2.2, following the transformations described in Section 2.3, to produce a functionally correct dataflow circuit; other passes perform additional analysis and optimizations (e.g., memory access analysis to create the memory interfaces described in Section 2.5). The output is a dataflow graph in the form of a DOT netlist. The DOT netlist is then given as input to the optimizer which contains the buffer placement tool—it uses a MILP solver to find the optimal buffer placement and sizes for a user-defined clock period constraint, as indicated in Section 2.4. This step produces an optimized DOT netlist. Finally, the DOT describing the dataflow circuit is converted into a VHDL netlist of dataflow components. This netlist, in conjunction with a predefined library of dataflow components, can be synthesized into an FPGA bitstream.

4 USING DYNAMATIC

This section provides instructions for installation and basic usage of Dynamatic. We use a simple histogram application to demonstrate the capabilities and outputs of the tool. This application, shown in Figure 5, is representative of a case where dynamic scheduling

```c
#define  N  128

void histogram( int feature[], float weight[], float hist[], int n )
{
  for (int i=0; i<n; ++i)
  {
    int m = feature[i];
    float wt = weight[i];
    float x = hist[m];
    hist[m] = x + wt;
  }
}


int main ( void )
{

   int   feature[N];
   float weight[N];
   float hist[N];
   int n = N;

   for ( int indx = 0; indx < N; indx++ )
   {
       feature[indx] = indx+1;
       weight[indx] = indx*2;
       hist[indx] = 0;
   }

   histogram( feature, weight, hist, n );

   return 0;

}
```

Figure 5: Histogram C code.

```
************************************************************
******Dynamic High-Level Synthesis Compiler*****************
******Andrea Guerrieri - Lana Josipovic - EPFL-LAP 2019 **********
******Version 0.2 - Build 0.1 ********************************
************************************************************

Dynamatic> help
help
List of supported commands:
help         : Shows Available commands
?            : Shows Available commands
source       : Source a script file
set_project  : Set the project directory
set_top_file : Set the top level file
set_period   : Set the hardware period
set_target   : Set target FPGA
analyze      : Analyze source
elaborate    : Elaborate source
synthesis    : C synthesis
optimize     : Timing optmizations
write_hdl    : Generate VHDL
reports      : Report resources and timing
cdfg         : Show control data flow graph
status       : Report design status
simulate     : Simulation
update       : Check for updates
history      : History command list
about        : Disclaimers & Copyrights
exit         : Exit

Done
Dynamatic>
```

Figure 6: Dynamatic shell.

is useful: a possible read-after-write dependence across loop iterations will prevent any static scheduling technique to implement a pipelined circuit, whereas dataflow circuits have the ability to resolve such dependences dynamically during execution [4].

4.1 Download and Installation

Dynamatic is publicly available for download on the website `dynamatic.epfl.ch`. The website contains the link to the source code in GitHub, an auto-install package, and a virtual machine with the pre-installed and ready-to-use tool.

Automatic Installation. The automated installation procedure installs and builds the main Dynamatic flow using the latest version of the GitHub source code. The minimum requirements for installation are as follows:

- Operating system: Linux CentoS 7.6 or Ubuntu 18.04
- Memory: 4GB of RAM
- Storage space: 25 GB
- Installation time: ~2 hours

Dynamatic has dependences on the following packages, which must be installed beforehand:

- g++
- cmake
- git
- pkg-config
- dot
- cbc
- graphviz

To install and build Dynamatic, create a folder in which the tool will be installed, download the file "dhls_setup_self_extract.sh", and place it into the target folder. From the target folder, run the following command:

```
bash ./dhls_setup_self_extract.sh.
```

Virtual Machine. The virtual machine contains the entire Dynamatic toolchain, preinstalled with all of its dependences. The minimum requirements for using the virtual machine are as follows:

- Virtualization Environment: Oracle Virtual Box
- Operating system: Windows or Linux
- Memory: 4GB of RAM to dedicate to the VM
- Storage space: 80 GB

4.2 Writing the Input Code

The input to Dynamatic is a subset of the C/C++ language. The input code must contain a main function which calls a single function to be synthesized by Dynamatic. All input data to the function (e.g., function arguments, array data, etc.) must be initialized in the main.

When writing input code for the current version of Dynamatic, the user should follow these restrictions:

- Recursive calls and dynamic memory allocation are not supported.
- All function calls from the top function to synthesize (i.e., from the function called from the main) must be inlined.

Command	Description
source	Source a script file. All the commands in the file will be parsed and executed.
set_project	Set the project directory. The default value is "." (local directory).
set_top_file	Set the top level file to be synthesized. The file should be in a directory called "src".
set_period	Set the target clock period of the output circuit, expressed in nanoseconds.
set_target	Set the target FPGA. The default target is a Xilinx Kintex-7 device (7k160tfbg484).
analyze	Perform initial source file analysis; check syntax correctness and identify pragmas.
elaborate	Prepare the code for synthesis by removing comments and adding information used by the tool internals.
synthesis	Perform the synthesis from C/C++ into a dataflow circuit by invoking standard and custom LLVM passes.
optimize	Optimize the circuit; insert buffers to optimize throughput and meet clock period.
write_hdl	Translate the DOT intermediate representation into a hardware description language (VHDL).
reports	Print resource utilization and timing reports.
cdfg	Visualize the control/dataflow graph of the synthesized circuit.
status	Provide the status of the current design, e.g., *none*, *setup*, *analyzed*, *elaborated*, *synthesized*, and *optimized*.
simulate	Simulate the generated hardware by interfacing Dynamatic with a VHDL simulator.
update	Automatically search for updates and install them if available.
history	Show the list of recently used commands.
about	Print information about Dynamatic.
help	Show the help.
exit	Terminate the Dynamatic execution.

Table 1: Command list.

- Global variables should be provided to functions as arguments (either independently or in an appropriate structure).
- Pointers to arrays should not be used.

Histogram Example. The program in Figure 5 is structured to be compatible with the Dynamatic frontend: the main program initializes the input data (i.e., function arguments) and calls the histogram function which will be synthesized into a dataflow circuit.

4.3 Running Dynamatic

Prior to running Dynamatic, the terminal needs to be initialized by setting the environment and shell variables. The initialization can be performed using an initialization script file (generated during the installation process and preexisting in the virtual machine) by invoking the following command:

```
source install_path/init_dhls.sh
```

The simplest way to interact with the tool is through the Dynamatic shell. To start the shell, type:

```
dynamatic
```

.

An example of the initialized shell is given in Figure 6. Table 1 reports the full commands list currently implemented in Dynamatic and usable through the shell.

```
#Author: Andrea Guerrieri - EPFL-LAP
#email:andrea.guerrieri@epfl.ch
#Dynamatic synthesis script
#ver.1.0

set_project .
set_top_file histogram.cpp
set_period 5.0
set_target 7k160tfbg484
analyze
elaborate
synthesis
optimize
write_hdl
exit
```

Figure 7: Dynamatic shell script example.

Different design steps can be automated using Dynamatic shell scripting. Figure 7 shows an example of a script which includes the commands for the complete compilation flow. The sequence of commands can be customized by the user, depending on the design objectives. The Dynamatic script can be sourced from the shell using the command **source** or passed as the first argument to Dynamatic itself, as follows:

```
dynamatic synthesis.tcl
```

```
"phi_2" [type = "Mux", bbID = 2, in = "in1?:1 in2:32 in3:32 ", out = "out1:32", delay=0.366];
"load_11" [type = "Operator", bbID = 2, op = "lsq_load_op", bbID= 2, portId= 0, in = "in1:32 in2:32", out = "out1:32 out2:32 ", delay=0.000, latency=2, II=1];
"fadd_12" [type = "Operator", bbID = 2, op = "fadd_op", in = "in1:32 in2:32 ", out = "out1:32 ", delay=0.966, latency=10, II=1];
"store_0" [type = "Operator", bbID = 2, op = "lsq_store_op", bbID= 2, portId= 0, in = "in1:32 in2:32 ", out = "out1:32 out2:32", delay=0.000, latency=0, II=1];
"cst_2" [type = "Constant", bbID = 2, in = "in1:32", out = "out1:32", value = "0x00000001"];
"fork_0" [type = "Fork", bbID = 1, in = "in1:32", out = "out1:32 out2:32 "];
"branch_0" [type = "Branch", bbID = 1,  in = "in1:32 in2?:1", out = "out1+:32 out2-:32"];
"LSQ_hist" [type = "LSQ", bbID = 0, in = "in1:0*c0 in2:32*l0a in3:32*s0a in4:32*s0d ", out = "out1:32*l0d out2:0*e ", memory = "hist", bbcount = 1, ldcount = 1,
stcount = 1, fifoDepth = 16, numLoads = "{1}", numStores = "{1}", loadOffsets = "{{0;0}}", storeOffsets = "{{1;0}}", loadPorts = "{{0;0}}", storePorts = "{{0;0}}"];

"phi_2" -> "fork_1" [color = "red", from = "out1", to = "in1"];
"load_8" -> "fadd_12" [color = "red", from = "out1", to = "in2"];
"load_11" -> "fadd_12" [color = "red", from = "out1", to = "in1"];
"fadd_12" -> "store_0" [color = "red", from = "out1", to = "in1"];
"cst_2" -> "add_15" [color = "red", from = "out1", to = "in2"];
"add_15" -> "fork_3" [color = "red", from = "out1", to = "in1"];
```

Figure 8: Snippet of the intermediate representation of the dataflow circuit in DOT format. The netlist contains a list of all dataflow components present in the design and specifies the channels (i.e., connections) between the components. Components and channels are described with additional attributes, listed in Table 2.

```
Dynamatic> synthesis
synthesis
compile histogram_elaborated.cpp ./home/dynamatic/Dynamatic/etc/llvm-6.0/bin/clang -emit-llvm -S   -c src/histogram_elaborated.cpp -o .histogram_elaborated.cpp.ll
; ModuleID = '.histogram_elaborated.cpp_mem2reg_constprop_simplifycfg_die.ll'
source_filename = "src/histogram_elaborated.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind uwtable
define void @_Z9histogramPiPfS0_i(i32* %feature, float* %weight, float* %hist, i32 %n) #0 {
block1:
  %cmp1 = icmp sgt i32 %n, 0
  br i1 %cmp1, label %block2, label %block3

block2:                                           ; preds = %block2, %block1
  %i.02 = phi i32 [ %inc, %block2 ], [ 0, %block1 ]
  %"2" = zext i32 %i.02 to i64
  %arrayidx = getelementptr inbounds i32, i32* %feature, i64 %"2"
  %"3" = load i32, i32* %arrayidx, align 4
  %"4" = zext i32 %i.02 to i64
  %arrayidx2 = getelementptr inbounds float, float* %weight, i64 %"4"
  %"5" = load float, float* %arrayidx2, align 4
  %idxprom3 = sext i32 %"3" to i64
  %arrayidx4 = getelementptr inbounds float, float* %hist, i64 %idxprom3
  %"6" = load float, float* %arrayidx4, align 4
  %add = fadd float %"6", %"5"
  %idxprom5 = sext i32 %"3" to i64
  %arrayidx6 = getelementptr inbounds float, float* %hist, i64 %idxprom5
  store float %add, float* %arrayidx6, align 4
  %inc = add nuw nsw i32 %i.02, 1
  %cmp = icmp slt i32 %inc, %n
  br i1 %cmp, label %block2, label %block3

block3:                                           ; preds = %block2, %block1
  ret void
}
```

Figure 9: LLVM intermediate representation of the histogram function in Figure 5, obtained from C code during synthesis. This IR is the starting point of the conversion into a dataflow circuit.

4.4 Synthesis

After the correctness and the format of the code have been checked in the preprocessing compiler steps (i.e., analysis and elaboration), it can be synthesized into a dataflow circuit. The synthesis step invokes clang and LLVM to transform the function into a static single assignment intermediate representation, which is the starting point for the dataflow conversion. Our custom passes transform this IR into a dataflow circuit representation, following the rules described in Section 2.3.

Dynamatic employs an intermediate representation for describing the synthesized dataflow circuits—the specification is based on the DOT language from Graphviz [1]. A dataflow circuit is represented by a digraph where each node corresponds to a component and each edge corresponds to a channel. Components have input and output ports; channels are unidirectional and connect an output port from one component to an input port of another component. Every port connects to exactly one input or output channel, which explicitly models a single token transfer from one component to another.

The components and channels can be annotated with attributes in the DOT netlist. These attributes represent component and channel properties which are needed to generate a correct VHDL netlist as well as information which is exploited by certain optimization steps (i.e., buffer placement). The attribute descriptions are summarized in Table 2. An absence of a numerical attribute assumes a zero value.

A DOT netlist can be visualized using the following command:

```
dot -Tpng file.dot > file.png
```

This command outputs a graphical representation of the control/dataflow graph, with components grouped into basic blocks and connected via channels.

Histogram example. Dynamatic synthesis invokes clang and LLVM to transform the histogram function into a static single assignment intermediate representation, reported in Figure 9. This IR is the starting point for the dataflow conversion. Our custom passes transform this IR into a dataflow circuit representation and output it in DOT format—a snippet is shown in Figure 8 and the complete control/dataflow graph is depicted in Figure 11. As in the LLVM IR, the operations are organized into three basic blocks (green rectangles in Figure 11); Dynamatic transforms the graph to propagate data into each block from its immediate predecessor blocks, as described in Section 2.3; the channels between different BBs are depicted in blue. Each LLVM operation is translated into an equivalent dataflow component and components with multiple successors in the same BB are followed by forks. The yellow channels and the components they connect represent the control-only

Attribute	Description
type	Type of dataflow component (e.g., fork, merge).
in, out	Lists of input and output ports. Each port has a suffix representing its bitwidth (e.g., in1:32 indicates a 32-bit input); zero-width ports are control-only.
delay	Combinational delay of a component or a port in ns.
latency	Latency (in clock cycles) of pipelined components.
II	Initiation interval (in clock cycles) of pipelined units.
slots	Number of buffer slots.
transparent	Buffer transparency (true or false).
bbID	Index of the BB that the component belongs to.
op	Operation type of an LLVM arithmetic or memory instruction (i.e., add, sub, load).
conditional suffixes	Suffix '?' represents the condition port. '+' and '-' indicate true and false ports, respectively. These attributes describe conditional components (e.g., branch, LLVM select, mux).
memory attributes	Attributes for memory operations and interface; used for generating application-specific memory interfaces.
value	Constant value (in hexadecimal).

Table 2: Attributes used in the intermediate representation in DOT format.

network Dynamatic inserts; it follows the control flow through the BBs, triggers certain constants, and communicates with the memory interface.

In the figure, the memory interface nodes are depicted as `LSQ_hist`, `MC_feature`, and `MC_weight`. As discussed in Section 2.5, the LSQ is used to dynamically resolve dependences between the reads and writes of array `hist`. As memory hazards are not possible for the other two arrays, they do not need an LSQ but connect directly to their BRAMs through the MC nodes. The connections between the memory operations and the interfaces they connect to are shown in green in the figure.

4.5 Optimizing Performance

Circuits produced in the previous step can have combinational cycles and some paths may cause backpressure. The optimization step inserts buffers to meet the clock period constraint, break all combinational loops, and relieve backpressure from the forks to achieve a pipelined design, as described in Section 2.4. This step will output a new DOT netlist—the netlist will indicate the channels where buffers are inserted and all buffers will be described with their slot count and transparency using the corresponding attributes from Table 2.

Histogram example. The circuit produced in the previous step has several combinational cycles across the control-flow loop through BB2. Furthermore, there are some paths that cause backpressure and prevent pipelining: for instance, the floating-point adder `fadd_12` takes 10 cycles to process a piece of data (see latency attribute in Figure 8); the delayed data arrival to `store_0` causes the store to

```
Dynamatic> optimize
Optimize
buffers shab 5 0.0 cbc 1 20 ./reports/histogram_elaborated.dot
./reports/histogram_elaborated_bbgraph.dot
./reports/histogram_elaborated_optimized.dot
./reports/histogram_elaborated_bbgraph_optimized.dot 0
==================
READING BB DOT FILE
==================
Reading graph name...
Reading set of nodes...
Reading set of edges between nodes...
Setting BB frequencies...
BB1 : 1
BB2 : 128
BB3 : 1

Adding elastic buffers with period=5 and buffer_delay=0
=====================
ADDING ELASTIC BUFFERS
=====================
Extracting marked graphs
-------------------------
Iteration 1
Storing CFDFC and corresponding Marked Graph...
-------------------------
******************
Covered Frequency = 127, Total Frequency = 129, Coverage = 0.984496
******************

Creating MILP variables...
----------------------
Adding buffer in branch_0:out1 -> phi_2:in2 | slots: 1, trans: 0 | BB1 -> BB2
Adding buffer in branch_1:out1 -> phi_n0:in2 | slots: 1, trans: 0 | BB1 -> BB2
Adding buffer in branchC_4:out1 -> phiC_1:in1 | slots: 1, trans: 0 | BB1 -> BB2
Adding buffer in branchC_5:out2 -> phiC_2:in2 | slots: 1, trans: 0 | BB2 -> BB3
Adding buffer in load_8:out1 -> fadd_12:in2 | slots: 1, trans: 1 | BB2 -> BB2
Adding buffer in add_15:out1 -> fork_3:in1 | slots: 2, trans: 0 | BB2 -> BB2
Adding buffer in fork_2:out2 -> store_0:in2 | slots: 11, trans: 1 | BB2 -> BB2
Adding buffer in branch_3:out1 -> phi_n0:in3 | slots: 2, trans: 0 | BB2 -> BB2
Adding buffer in branchC_5:out1 -> phiC_1:in2 | slots: 2, trans: 0 | BB2 -> BB2
Adding buffer in LSQ_hist:out1 -> load_11:in1 | slots: 1, trans: 0 | BB0 -> BB2
Adding buffer in MC_feature:out1 -> load_5:in1 | slots: 1, trans: 0 | BB0 -> BB2
Adding buffer in MC_weight:out1 -> load_8:in1 | slots: 1, trans: 0 | BB0 -> BB2
-----------------

*** Throughput achieved in sub MG 0: 1.00 ***
```

Figure 10: Optimize step (buffer insertion).

stall the address coming from `fork_2` and prevents this fork from issuing new tokens quickly.

The optimization report is given in Figure 10, which indicates the types and sizes of instantiated buffers. Note that the aforementioned path from `fork_2` to `store_0` now contains a large transparent buffer (i.e., a FIFO) which accumulates address values while the floating-point adder computes, hence alleviating backpressure and enabling pipelined execution. Small nontransparent buffers break combinational cycles and ensure that the period constraint (in this example, set to 5 ns, as indicated in the report) is met.

4.6 VHDL Output

The final DOT output of the previous step can be translated into a functional, high-performance dataflow circuit. The write_hdl command produces the resulting VHDL netlist and generates application-specific memory components. Together with the Dynamatic component library, these files form a complete hardware design; its bitstream can be produced using standard tools (i.e., Vivado) and it can be functionally verified using simulation environments such as ModelSim [9].

Application-specific outputs. The VHDL component netlist obtained in the final compilation step is a direct translation of the DOT netlist into an HDL description: all components are translated into VHDL component instantiations and all channels are translated

into VHDL signal connections. Every channel is represented with three signals: a data signal and a handshake pair. All arguments of the implemented function, as well as signals which connect to memory, are specified as the top entity input and output ports, as shown in Figure 13. The current version of Dynamatic assumes dual-port BRAM memory and creates the interfaces accordingly; if a memory requires an LSQ, it is custom-generated using the memory attributes from Table 2, which specify the LSQ depth, port counts and types, as well as access ordering information which the LSQ relies on for correct execution, as indicated in Section 2.5.

RTL component library. The Dynamatic flow is device-independent—the produced VHDL netlist, as well as our HDL implementations of the dataflow components, are usable with any FPGA or for ASIC design. All dataflow components are fully parametrizable to arbitrary bitwidths and, in most cases (i.e., as far as component functionality permits), the number of inputs and outputs. The arithmetic units employed by Dynamatic currently target the Kintex-7 family of Xilinx FPGAs; we extract them from the Vivado environment [10] and instantiate using Xilinx component libraries. To use these components, the user must have a valid Vivado license. All components employ a custom wrapper with handshake signals to be compatible with the rest of our dataflow components. Future releases of Dynamatic will extend the component library to target other devices families and vendors as well.

Histogram example. The translation of the histogram DOT representation into a VHDL netlist produces the report in Figure 12; it outlines all the components instantiated in the VHDL netlist. As explained in the previous section, the netlist top entity in Figure 13 contains BRAM ports for all three arrays of the original code as well as a simple port for the argument `n`. In addition, the entity has ports which represent the start and the end point of the control-only path—they trigger circuit execution and indicate execution termination, respectively.

An example of an instantiated component as well as its connections to other components in the VHDL netlist is given in Figures 14 and 15. Each data input (`dataInArray`) and output (`dataOutArray`) port is accompanied by a pair of ports for the handshake signals: signals arriving on `pValidArray` ports correspond to valid signals of the predecessors and `readyArray` signals indicate the availability of the adder inputs to accept data. Similarly, together with the output data, the adder outputs a validity signal on the `validArray` port; the signal will stay active until the adder receives the confirmation from the successor that the data has been accepted using the `nReadyArray` port.

Apart from the VHDL netlist, this step will generate an LSQ for the array `hist`. This component will be customized to the description in the DOT file; in this case, the LSQ depth will be equal to 16, it will connect to a single BB, one load port, and one store port (as the attributes `fifoDepth`, `bbcount`, `ldcount`, and `stcount` in Figure 8 indicate). The rest of the attributes are used for configuring the LSQ internals and ensuring correct access ordering.

5 ADVANCED TOPICS

Dynamatic can be easily customized or extended with new features. This section outlines some extension possibilities.

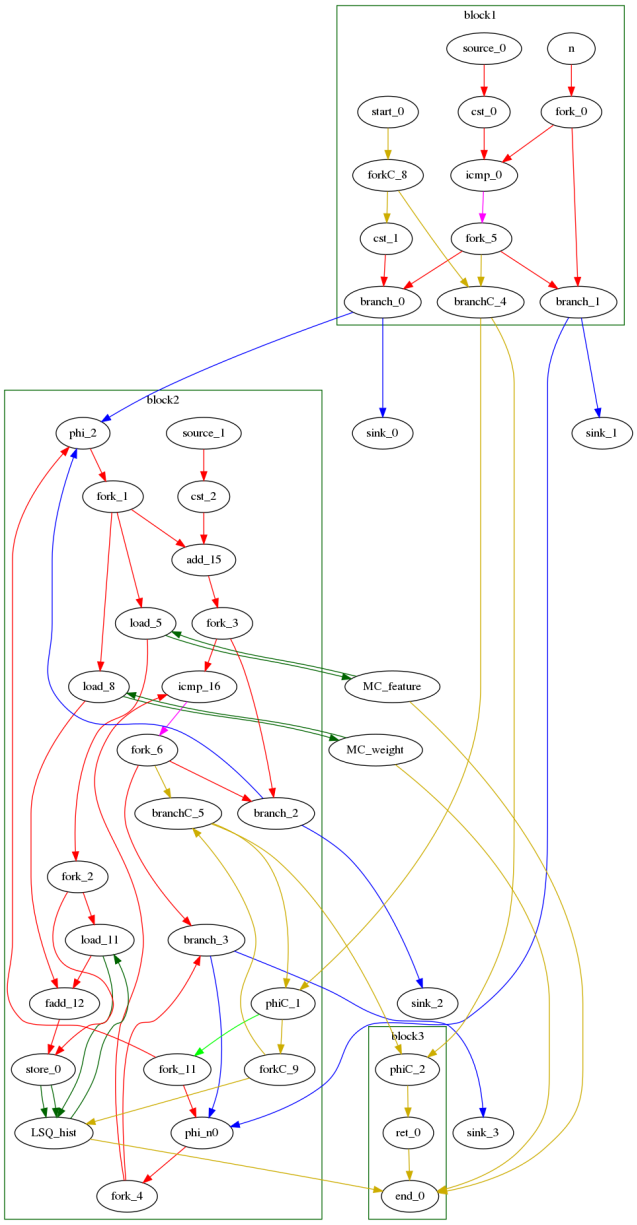

Figure 11: Intermediate representation of a dataflow circuit, organized as a CDFG of dataflow components grouped into BBs. The circuit corresponds to the histogram example.

5.1 Extending the Flow

Dynamatic can be extended with new features at any point of its flow—we here provide some insights into where to integrate new optimizations.

Adding pragmas. Dynamatic has custom pragma support—pragmas defined by the user can easily be added to the existing flow. The pragmas specified in the input code are checked in the preprocessing steps (i.e., analyze and elaborate) and propagated to the rest of the flow. The user can access the list of pragmas from

```
Dynamatic> write_hdl

*********************************************************************
******Dynamic High-Level Synthesis Compiler **********************
******Andrea Guerrieri - Lana Josipovic - EPFL-LAP 2019 **********
******Dot to VHDL Generator**************************************
*********************************************************************
Parsing ./reports/histogram_elaborated_optimized.dot

Report Modules
+-----------------------------------------------------------------------------+
|   Node_ID|               Name|         Module_type|    Inputs|   Outputs|
+-----------------------------------------------------------------------------+
|         0|                  n|               Entry|         1|         1|
|         1|              cst_0|            Constant|         1|         1|
|         2|             icmp_0|            Operator|         2|         1|
|         3|              cst_1|            Constant|         1|         1|
|         4|              phi_2|                 Mux|         3|         1|
|         5|             load_5|            Operator|         2|         2|
|         6|             load_8|            Operator|         2|         2|
|         7|            load_11|            Operator|         2|         2|
|         8|            fadd_12|            Operator|         2|         1|
|         9|            store_0|            Operator|         2|         2|
|        10|              cst_2|            Constant|         1|         1|
|        11|             add_15|            Operator|         2|         1|
|        12|            icmp_16|            Operator|         2|         1|
|        13|              ret_0|            Operator|         1|         1|
|        14|             phi_n0|                 Mux|         3|         1|
|        15|             fork_0|                Fork|         1|         2|
|        16|             fork_1|                Fork|         1|         3|
|        17|             fork_2|                Fork|         1|         2|
|        18|             fork_3|                Fork|         1|         2|
|        19|             fork_4|                Fork|         1|         2|
|        20|           branch_0|              Branch|         2|         2|
|        21|           branch_1|              Branch|         2|         2|
|        22|             fork_5|                Fork|         1|         3|
|        23|           branch_2|              Branch|         2|         2|
|        24|           branch_3|              Branch|         2|         2|
|        25|             fork_6|                Fork|         1|         3|
|        26|          LSQ_hist|                 LSQ|         4|         2|
|        27|       MC_feature|                  MC|         4|         2|
|        28|        MC_weight|                  MC|         4|         2|
|        29|           end_0|                Exit|         4|         1|
|        30|          start_0|               Entry|         1|         1|
|        31|          forkC_8|                Fork|         1|         2|
|        32|         branchC_4|              Branch|         2|         2|
|        33|             phiC_1|           CntrlMerge|         2|         2|
|        34|          forkC_9|                Fork|         1|         2|
|        35|         branchC_5|              Branch|         2|         2|
|        36|             phiC_2|               Merge|         2|         1|
|        37|             sink_0|                Sink|         1|         0|
|        38|             sink_1|                Sink|         1|         0|
|        39|             sink_2|                Sink|         1|         0|
|        40|             sink_3|                Sink|         1|         0|
|        41|           source_0|              Source|         0|         1|
|        42|           source_1|              Source|         0|         1|
|        43|          fork_11|                Fork|         1|         2|
|        44|         Buffer_1|                TEHB|         1|         1|
|        45|         Buffer_2|              Buffer|         1|         1|
|        46|         Buffer_3|                Fifo|         1|         1|
|        47|         Buffer_4|              Buffer|         1|         1|
|        48|         Buffer_5|              Buffer|         1|         1|
|        49|         Buffer_6|              Buffer|         1|         1|
|        50|         Buffer_7|              Buffer|         1|         1|
|        51|         Buffer_8|              Buffer|         1|         1|
|        52|         Buffer_9|              Buffer|         1|         1|
|        53|        Buffer_10|              Buffer|         1|         1|
|        54|        Buffer_11|              Buffer|         1|         1|
|        55|        Buffer_12|              Buffer|         1|         1|
+-----------------------------------------------------------------------------+
Generating ./reports/histogram_elaborated_optimized.vhd
Generating LSQ 0 component...
java -jar -Xmx7G /home/dynamatic/Dynamatic/bin/lsq.jar --target-dir . --spec-
file ./reports/histogram_elaborated_optimized_lsq0_configuration.json
Elaborating design...
Done elaborating.
```

Figure 12: Component report after envoking the write_hdl command, which converts the DOT netlist into an equivalent VHDL netlist.

an LLVM pass and use it to modify the dataflow circuit according to the information present in the pragma.

Adding LLVM passes. The pass which converts LLVM IR into a dataflow circuit is based on LLVM—it relies on LLVM information to transform the CDFG. Once this pass outputs the dataflow graph (in DOT form), the following steps do not interact with LLVM any further. Hence, all LLVM-related optimizations should be inserted before this point of the flow. One might insert an LLVM pass before the conversion to dataflow, or build a pass which interacts with the

```
-- ------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.customTypes.all;
-- ------------------------------------------------------
entity histogram_elaborated is
port (
        clk:  in std_logic;
        rst:  in std_logic;
        start_in:  in std_logic_vector (0 downto 0);
        start_valid:  in std_logic;
        start_ready:  out std_logic;
        end_out:  out std_logic_vector (0 downto 0);
        end_valid:  out std_logic;
        end_ready:  in std_logic;
        n_din : in std_logic_vector (31 downto 0);
        n_valid_in : in std_logic;
        n_ready_out : out std_logic;
        hist_address0 : out std_logic_vector (31 downto 0);
        hist_ce0 : out std_logic;
        hist_we0 : out std_logic;
        hist_dout0 : out std_logic_vector (31 downto 0);
        hist_din0 : in std_logic_vector (31 downto 0);
        hist_address1 : out std_logic_vector (31 downto 0);
        hist_ce1 : out std_logic;
        hist_we1 : out std_logic;
        hist_dout1 : out std_logic_vector (31 downto 0);
        hist_din1 : in std_logic_vector (31 downto 0);
        feature_address0 : out std_logic_vector (31 downto 0);
        feature_ce0 : out std_logic;
        feature_we0 : out std_logic;
        feature_dout0 : out std_logic_vector (31 downto 0);
        feature_din0 : in std_logic_vector (31 downto 0);
        feature_address1 : out std_logic_vector (31 downto 0);
        feature_ce1 : out std_logic;
        feature_we1 : out std_logic;
        feature_dout1 : out std_logic_vector (31 downto 0);
        feature_din1 : in std_logic_vector (31 downto 0);
        weight_address0 : out std_logic_vector (31 downto 0);
        weight_ce0 : out std_logic;
        weight_we0 : out std_logic;
        weight_dout0 : out std_logic_vector (31 downto 0);
        weight_din0 : in std_logic_vector (31 downto 0);
        weight_address1 : out std_logic_vector (31 downto 0);
        weight_ce1 : out std_logic;
        weight_we1 : out std_logic;
        weight_dout1 : out std_logic_vector (31 downto 0);
        weight_din1 : in std_logic_vector (31 downto 0));
end;
```

Figure 13: Top entity interface.

dataflow pass. An example of such a pass is the Dynamatic pass for memory analysis [3]—it is an independent pass which uses LLVM and Polly [2] to analyze memory accesses. The main pass then uses these results to appropriately design the memory interfaces and the results of this pass are reflected in the output DOT netlist.

Modifying the dataflow graph. Once the first part of the flow produces a functional dataflow circuit, it can be further modified independently of the LLVM framework. An example of such modification is the buffer placement—it inputs the DOT file from the previous step, appropriately adds buffers, and produces another DOT file, which is then sent to the backend of the framework. A user can easily insert additional optimization steps (either before or after the buffer placement) in the same manner—this approach circumvents the need to modify the rest of the codebase and the modifications can be easily interfaced with the rest of the framework through

```
fadd_12: entity work.fadd_op(arch) generic map (2,1,32,32)
port map (
        clk => fadd_12_clk,
        rst => fadd_12_rst,
        dataInArray(0) => fadd_12_dataInArray_0,
        dataInArray(1) => fadd_12_dataInArray_1,
        pValidArray(0) => fadd_12_pValidArray_0,
        pValidArray(1) => fadd_12_pValidArray_1,
        readyArray(0) => fadd_12_readyArray_0,
        readyArray(1) => fadd_12_readyArray_1,
        nReadyArray(0) => fadd_12_nReadyArray_0,
        validArray(0) => fadd_12_validArray_0,
        dataOutArray(0) => fadd_12_dataOutArray_0
);
```

Figure 14: Component instantiation in the VHDL netlist.

the DOT format. Apart from the ability to implement custom optimization passes, modifying the DOT files is useful for manual explorations (e.g., changing arithmetic units, inserting buffers or modifying their properties, changing the memory interface type).

5.2 Adding Custom Components

As previously mentioned, Dynamatic is accompanied by a library of arithmetic units. It is straightforward for a user to add new components to the library.

Like any other dataflow component, any arithmetic unit requires a wrapper with handshake signals to communicate with the rest of the components. Our library provides a wrapper in which one can simply place a new arithmetic unit; the wrapper requires information on the component latency to correctly propagate the control through the unit. Once the component has been created, it is trivial to add it to the DOT netlist or to replace an existing component with the new one as well as to set its timing parameters (i.e., latency, delay, II). It is important to note that changes to components or their timing may impact the timing properties of the design—it is therefore recommended to perform such modifications prior to buffer placement, so that they are taken into account when buffering the design.

6 CONCLUSIONS

This paper provides an overview of the Dynamatic framework and some of its use cases. The Dynamatic team invites the community to try out the first generation of our HLS tool and to contribute to its development. For downloading the framework, more examples, and further documentation, please visit the Dynamatic website: `dynamatic.epfl.ch`.

```
load_8_clk <= clk;
load_8_rst <= rst;
fadd_12_pValidArray_1 <= load_8_validArray_0;
load_8_nReadyArray_0 <= fadd_12_readyArray_1;
fadd_12_dataInArray_1 <= load_8_dataOutArray_0;
MC_weight_pValidArray_0 <= load_8_validArray_1;
load_8_nReadyArray_1 <= MC_weight_readyArray_0;
MC_weight_dataInArray_0 <= load_8_dataOutArray_1;

load_11_clk <= clk;
load_11_rst <= rst;
fadd_12_pValidArray_0 <= load_11_validArray_0;
load_11_nReadyArray_0 <= fadd_12_readyArray_0;
fadd_12_dataInArray_0 <= load_11_dataOutArray_0;
LSQ_hist_pValidArray_1 <= load_11_validArray_1;
load_11_nReadyArray_1 <= LSQ_hist_readyArray_1;
LSQ_hist_dataInArray_1 <= load_11_dataOutArray_1;

fadd_12_clk <= clk;
fadd_12_rst <= rst;
store_0_pValidArray_0 <= fadd_12_validArray_0;
fadd_12_nReadyArray_0 <= store_0_readyArray_0;
store_0_dataInArray_0 <= fadd_12_dataOutArray_0;
```

Figure 15: Signal connections (corresponding to channels between dataflow components) in the VHDL netlist.

REFERENCES

[1] Graphviz graph visualization software. *https://www.graphviz.org/*.
[2] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, pages 1–6, Chamonix, Apr. 2011.
[3] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne. Shrink it or shed it! Minimize the use of LSQs in dataflow designs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, Tianjin, Dec. 2019. To appear.
[4] L. Josipović, P. Brisk, and P. Ienne. From C to elastic circuits. In *Proceedings of the 51st Annual Asilomar Conference on Signals, Systems, and Computers*, pages 121–25, Pacific Grove, Calif., Nov. 2017.
[5] L. Josipović, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):125:1–125:19, Sept. 2017.
[6] L. Josipović, R. Ghosal, and P. Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 127–36, Monterey, Calif., Feb. 2018.
[7] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella. Buffer placement and sizing for high-performance dataflow circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, Calif., Feb. 2020.
[8] The LLVM Compiler Infrastructure. *http://www.llvm.org*.
[9] Mentor Graphics. *ModelSim*.
[10] Xilinx Inc. *Vivado High-Level Synthesis*.