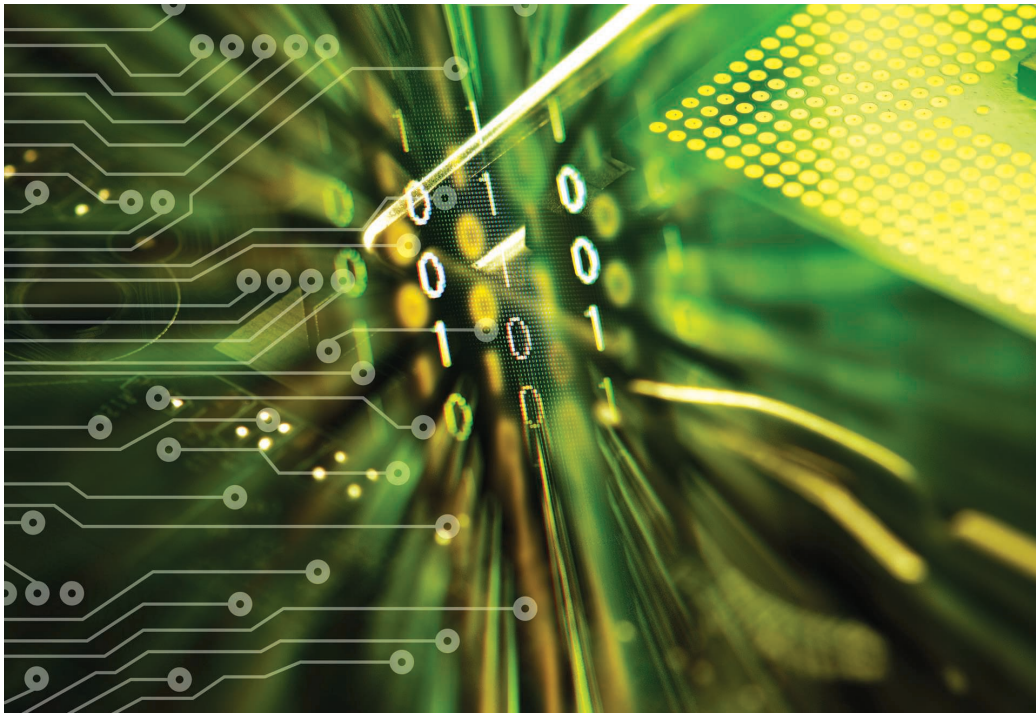
Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits

Lana Josipović, Andrea Guerrieri, and Paolo Ienne, *École Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland.*
lana.josipovic@epfl.ch, andrea.guerrieri@epfl.ch, paolo.ienne@epfl.ch

Abstract

High-level synthesis (HLS) tools generate hardware designs from high-level programming languages and should liberate designers from the details of hardware description languages like VHDL and Verilog. HLS tools typically build datapaths that are controlled using a centralized controller, which relies on a compile-time schedule to determine the clock cycle when each operation executes. Such an approach results in high-throughput pipelined designs only in cases where memory accesses are provably independent and critical control decisions are determinable during code compilation. Unfortunately, when this is not the case, current tools must make pessimistic assumptions, yielding inferior schedules and lower performance. Recent advances in HLS have explored methods to overcome the conservatism in static scheduling and to remove the inability of HLS tools to handle dynamic events. Dataflow circuits play a significant role in this context: they are built out of units that communicate using point-to-point pairs of handshake control signals and this distributed control mechanism effectively implements a dynamic schedule, adapted at runtime to particular memory and control outcomes. Dataflow circuits can exploit the same optimization opportunities as standard HLS circuits (i.e., pipelining and resource sharing), but also introduce to HLS features similar to those of modern superscalar processors (i.e., out-of-order memory accesses and speculative execution), which are key for HLS to be successful in new contexts and broader application domains.



©SHUTTERSTOCK.COM/CQ PHOTO JUY

Digital Object Identifier 10.1109/MCAS.2021.3071631
Date of current version: 24 May 2021

I. Introduction

Field Programmable Gate Arrays (FPGAs) are extremely versatile devices that can be configured into application-specific accelerators achieving high performance and energy efficiency. Recently,

FPGAs have been integrated into datacenters [2], [9], [56], packaged with processors [13], and introduced to new application domains. However, their success on the global market critically depends on the ability of software application developers to build efficient FPGA designs [30], [31], [51] by extracting sufficient performance through modern programming paradigms. *High-level synthesis* (HLS) tools enable programmers to automatically generate hardware designs from high-level software abstractions instead of writing tedious and time-consuming low-level hardware descriptions. Despite their progress and some commercial success in the last decade, HLS tools still tend to be criticized for the difficulty of extracting the desired level of performance: generating good circuits from high-level languages still requires peculiar code restructuring, expert user interaction, and extensive experimentation with the tools [39]. Moreover, current HLS techniques face a fundamental issue when handling irregular applications: because they rely on static scheduling, i.e., the cycle in which each operation executes is fixed at compile time [26], they force worst-case assumptions on memory and control dependences. Therefore, HLS tools are primarily usable by designers with hardware expertise and acceptable only for some classes of applications, but they are still unable to meet the need to accelerate emerging applications such as graph processing, data analytics, and sparse linear algebra operations.

In this article, we provide an overview of standard HLS techniques and describe some classic HLS optimizations and their tradeoffs. We then detail a fundamentally different form of HLS which produces dynamically scheduled, dataflow circuits out of high-level code. These circuits use a distributed control system that makes local scheduling decisions dynamically during runtime; data is propagated from unit to unit as soon as memory and control dependences allow it and, otherwise, it is stalled until all conditions for execution are satisfied. However, a straightforward translation of high-level code into a dataflow circuit is not sufficient to obtain circuits that are truly competitive and useful in the HLS context. We detail the methodologies to exploit the same optimization opportunities that standard HLS relies on (i.e., pipelining and resource sharing). We then discuss methods to achieve characteristics that are beyond what classic HLS can do (i.e., out-of-order memory accesses and speculation); these features introduce to dataflow circuits characteristics similar to those of modern superscalar processors. The resulting circuits achieve solutions that are, in particular cases, superior to those obtained using standard HLS techniques: similarly to the tradeoff between VLIW processors and superscalars, the performance of demanding applications is very significantly improved at an affordable cost.

II. How Does Classic HLS Work?

In this section, we describe typical high-level synthesis features and optimizations. We discuss standard scheduling techniques and illustrate the limitations they face in particular applications.

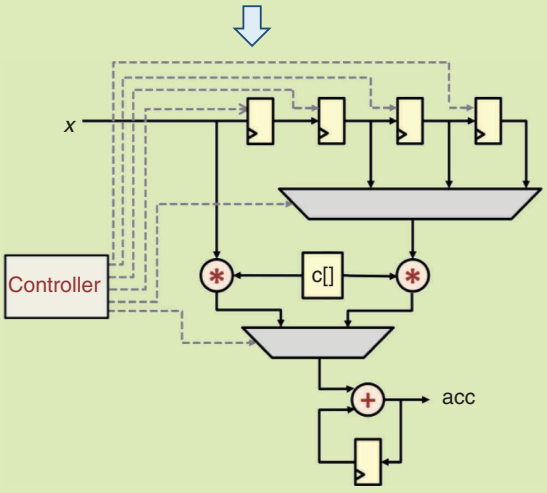
A. High-Level Synthesis

Hardware description languages (HDLs), such as VHDL and Verilog, have been used in the electronic design industry for decades to specify the details of hardware design in terms of low-level building blocks such as gates, registers, and multiplexers [3]. However, this description level requires hardware expertise and, typically, a longer time to develop the design. High-level synthesis tools allow designers to work at a higher level of abstraction by using a software language to specify the hardware functionality; this approach enables software engineers to program hardware and helps hardware engineers to speed up the design process as well as to efficiently explore the design space [51]. Although HLS can benefit both ASIC and FPGA designers, HLS tools are particularly gaining popularity in the FPGA domain, where the programming challenges of these devices are one of the biggest barriers to their mainstream adoption [3], [15].

Different HLS tools rely on different high-level representations to describe the underlying hardware; the most popular ones use C/C++ as an input language [7], [69]. Generally speaking, the user provides the input functional specification and particular design constraints such as target device, desired clock frequency, and memory interface description; the tool then automatically analyzes concurrency, inserts registers to achieve the desired frequency, generates the control and datapath logic, and maps data onto storage elements to optimize the bandwidth and resource usage [45]. The user is typically required to restructure the code and annotate it with pragmas to guide the tool in reaching the desired design point.

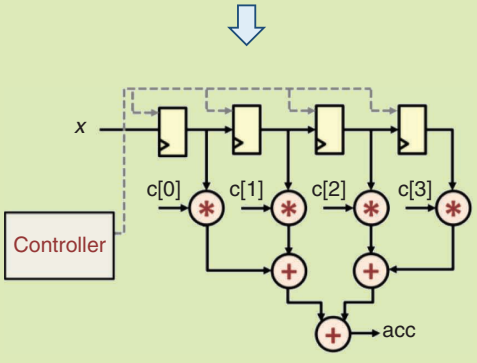
Figure 1 illustrates several out of many possibilities to specify the functionality of a simple FIR filter in C code as well as the resulting circuits produced by an HLS tool [45]; apart from the different datapaths, as shown in the figure, each design has a kernel-specific controller which triggers the datapath components at appropriate clock cycles; we will discuss this functionality in the following section. The first circuit is obtained from a typical software representation, without any hardware-specific annotations or code restructuring. The second code is manually unrolled to explicitly express available

```
acc = 0;
for (i = 3; i >= 0; i--) {
   if (i == 0) {
      shift_reg[0] = x;
      acc += x * c[0];
   } else {
      shift_reg[i] = shift_reg[i-1];
      acc += shift_reg[i] * c[i];
   }
}
y = acc;
```



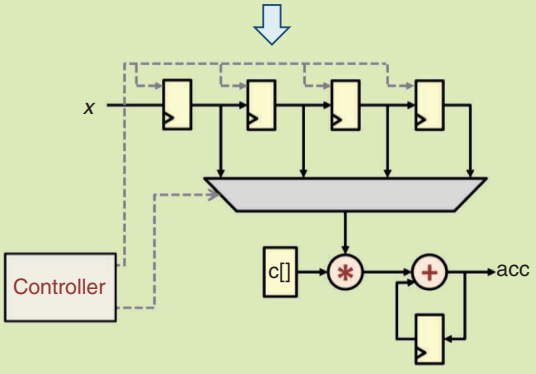
(a)

```
shift_reg[3] = shift_reg[2];
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = shift_reg[3] * c[3];
acc += shift_reg[2] * c[2];
acc += shift_reg[1] * c[1];
acc += shift_reg[0] * c[0];

y = acc;
```



(b)

```
shift_reg[3] = shift_reg[2];
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = 0;
for (i = 10; i >= 0; i--) {
   #pragma HLS pipeline
   acc += shift_reg[i] * c[i];
}
y = acc;
```



(c)

Figure 1. Design space exploration with static HLS [45]. All three codes in the figure describe the same functionality (i.e., an FIR filter); yet, the resulting HLS solutions differ in area and performance. (a) No optimization, (b) Unrolling, (c) Pipelining.

parallelism to the HLS tool—as the circuit below suggests, this design will employ multiple operators which can be used concurrently. The third design point uses a pragma to indicate to the tool that the code should be pipelined, i.e., the loop iterations should overlap for performance benefits. It is evident from the figure that the circuits differ in the number of employed resources (i.e., adders, multipliers, multiplexers); they also differ in performance, as we will discuss next.

B. Scheduling in HLS

HLS relies on a series of compiler optimizations to achieve performance- and area-efficient designs; some techniques are exploited by compilers in general (i.e., code motion, if conversion), whereas others are hardware-specific (i.e., bitwidth analysis, operation chaining). One of the key algorithms in HLS synthesis is scheduling, i.e., deciding the clock cycle in which each operation will execute. It is typically achieved through *system of difference constraints* (SDC) modeling which incorporates a variety of constraints, such as resource usage, data dependences, control dependences, and clock frequency [6], [16].

The three schedules in Figure 2 correspond to the circuits in Figure 1. The schedule of each design is regulated by the controller; it implements a *finite state machine* which controls the behavior of the datapath by triggering operations, enabling registers, and multiplexing values in appropriate clock cycles. The first schedule corresponds to the sequential execution of the software code: one iteration starts after the previous one has completed. The code restructuring in the second figure enables operations to execute in parallel, hence

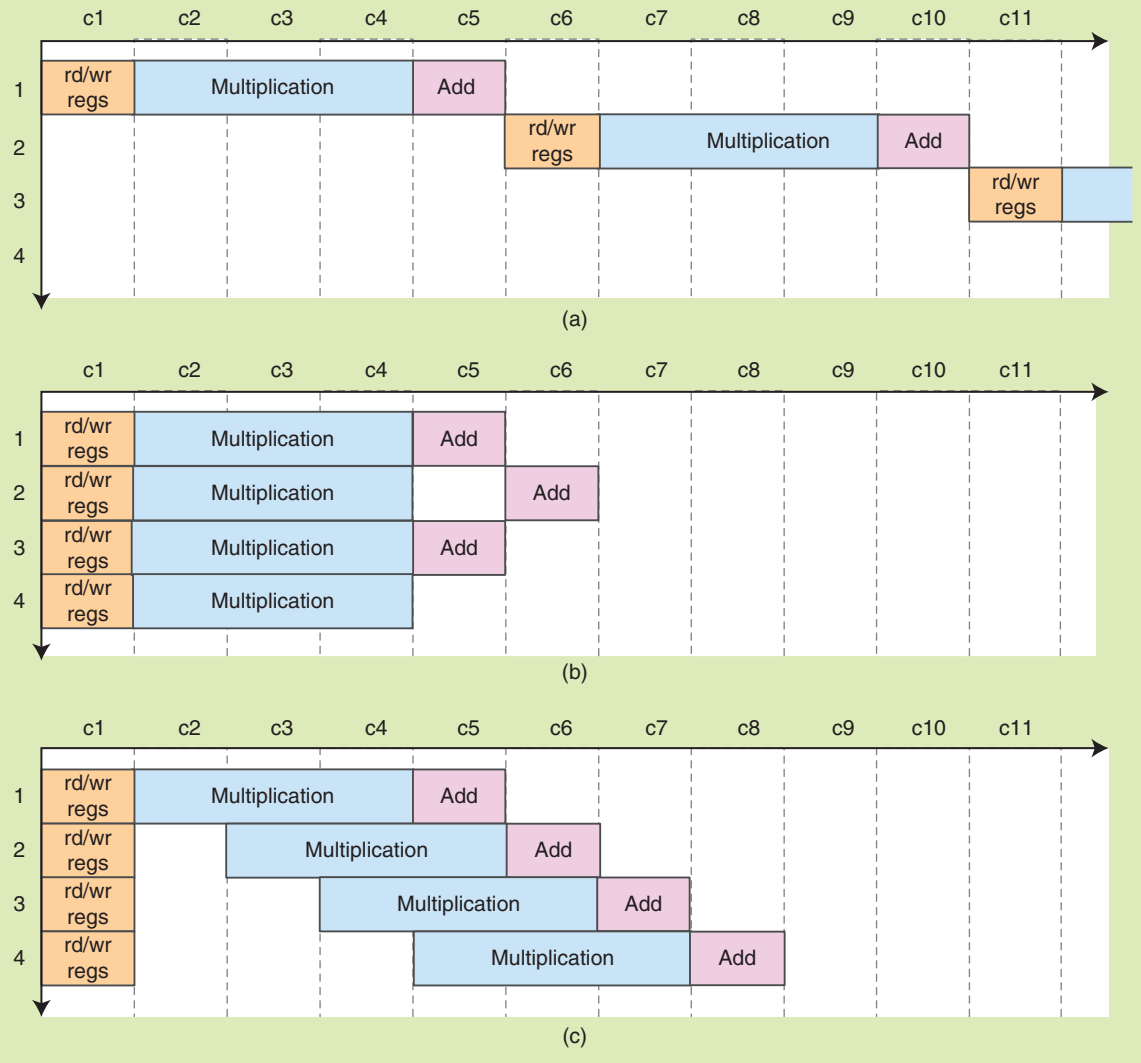


Figure 2. The schedules of the three design points from Figure 1. (a) No optimization, (b) Unrolling, (c) Pipelining.

lowering the execution time, but with the investment of additional resources. The third circuit employs loop pipelining: a new loop iteration starts on every clock cycle and the resource requirements are minimal (i.e., new data is inserted into the single adder and the single multiplier on every cycle).

Loop pipelining is one of the key performance optimizations in HLS—as the example above suggests, it allows loop iterations to overlap in the best possible manner while honoring all data and control dependences of the program. The technique originates from software pipelining techniques for *Very Long Instruction Word* (VLIW) processors, which rely on modulo scheduling algorithms to exploit instruction-level parallelism among successive loop iterations [46], [58]. A pipeline is characterized by its *initiation interval* (II), i.e., the number of clock cycles between consecutive loop iterations; the ideal II is equal to 1, as is the case for the pipelined schedule in the figure.

C. Limitations of Standard Scheduling

As in the case of VLIWs, the ability of HLS pipelining to achieve a low II is limited in the presence of resource or memory port constraints as well as long-latency loop-carried dependences. In addition, pipelining may be limited in cases where compile-time information is not sufficient to devise the best possible schedule. Examples of such situations include memory access patterns which cannot be determined at compile time, unpredictable control flow, and variable-latency operations. In such cases, standard tools must assume the presence of a dependence and produces a schedule with a conservative II.

To illustrate this limitation, consider the example in Figure 3. In this loop, there is a *possible* data dependence between the memory read of `hist[x[i+1]]` and the memory write to `hist[x[i]]` of the previous iteration. There is intrinsically no way a compiler or an HLS tool can ensure that such dependence does not exist, nor is it, in general, possible for a programmer to help the tools: in practice, the read may seldom or even never address the same value just written in memory, but there is no way to exclude a priori that this might happen. Ultimately, any HLS technique based on static scheduling hits the problem of potential dependences and needs to account for the worst-case scenario, irrespective of the actual data fetched from memory. The result is a conservative schedule valid for any possible input values, which assumes a dependence in every loop iteration

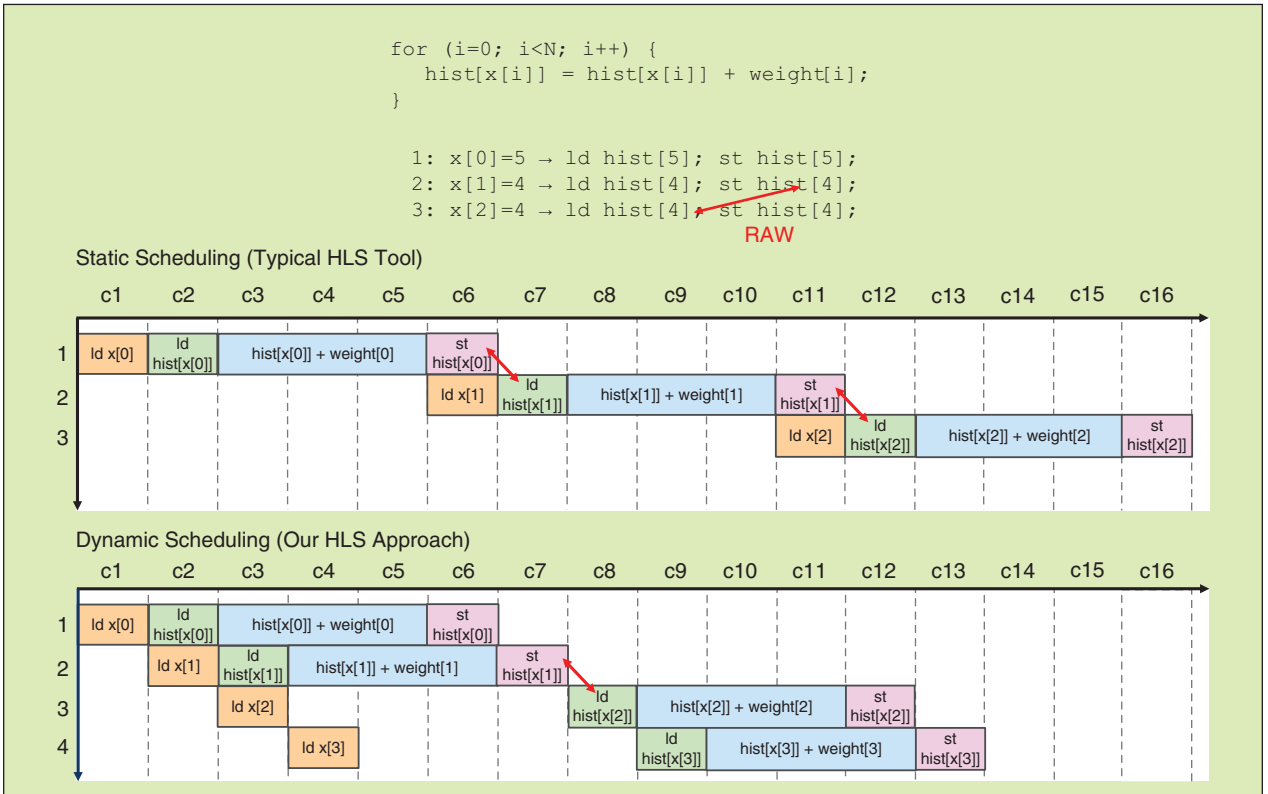


Figure 3. A static schedule created by a standard high-level synthesis tool, compared to a dynamic schedule possible with our approach. The HLS tool will conservatively assume a dependence between every loop iteration, whereas the dynamic design stalls only in the presence of an actual dependence (in this case, between iterations 2 and 3).

and postpones the read until the previous, possibly dependent write has been completed.

III. A Completely Different Way to Do HLS

The key to avoiding the limitations of static scheduling is to refrain from triggering the operations through a centralized pre-planned controller (as shown in Figure 4(a) and the examples in Figure 1), but to make scheduling decisions locally in the circuit as it runs: as soon as all conditions for execution are satisfied (e.g., the operands are available or critical control decisions are resolved), an operation starts. *Dataflow circuits* [18] are a natural method to realize such behavior. Such circuits are built out of units that implement *latency-insensitivity* by communicating with their predecessors and successors through point-to-point pairs of handshake control signals, as indicated in Figure 4(b). The data is propagated from unit to unit as soon as memory and control dependences allow it—otherwise, the handshaking mechanism stalls the data on-the-fly. This distributed control mechanism effectively implements a *dynamic schedule*, such as the bottom schedule in Figure 3: when a dependence exists (in the example, between the second and the third iteration) the dynamically scheduled circuit will stall the pipeline to prevent hazards. Otherwise, in the absence of an address collision, it will start a new iteration on every cycle and gain, in this case, up to a factor 6 in performance. Such dynamic behavior is beyond what classic static techniques can achieve [37].

In the rest of this article, after a brief digression on how the same issues have influenced computer architecture, we will describe our HLS methodology which produces dynamically scheduled dataflow circuits out of high-level code. We discuss our methodology to implement high-throughput pipelines and to identify resource sharing opportunities in the circuits we generate. We then detail the construction of a memory interface (i.e., a load-store queue) for dataflow circuits that can correctly handle memory accesses arriving out-of-order and show how to automatically customize this interface to a particular application. Further, we present a generic framework for handling speculative execution. All these features introduce to dataflow circuits characteristics similar to those of modern superscalar processors; we believe that such behavior is key for HLS to be successful in new contexts and broader application domains.

IV. Computer Architects Have Been There Already

The contrast between static and dynamic scheduling in HLS is in line with the experience in computer architecture.

Practically all high-end application processors in our computing devices and data centers are *superscalar out-of-order* processors [33]. Their architecture is usually similar to the one shown in Figure 5(a). The key idea is that *reservation stations* enable out-of-order execution: they hold back fetched and decoded instructions until

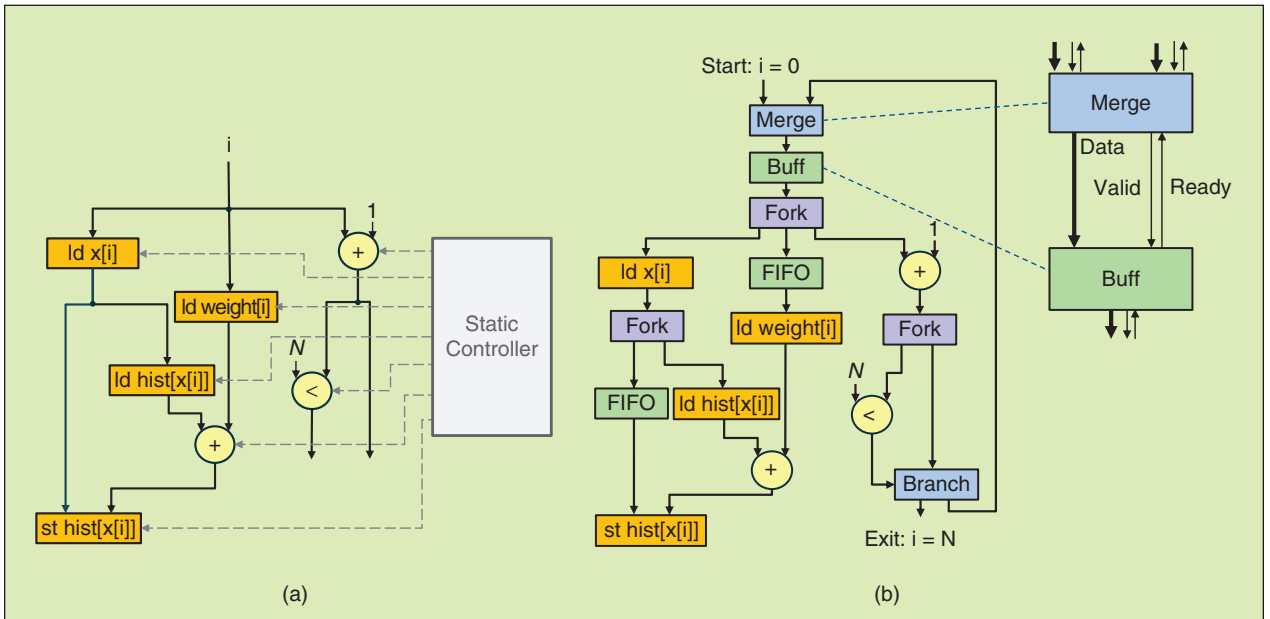


Figure 4. A statically (Figure 4(a)) and a dynamically (Figure 4(b)) scheduled circuit. The static circuit has a pre-planned controller which determines the time when each operation will execute. In contrast, the dynamically scheduled circuit contains a distributed control system which enables decision-making at runtime and offers greater flexibility and performance. The circuits in the figure correspond to the code from Figure 3.

all of their operands are available and let ready instructions advance; instructions are delayed if a load misses in the cache or a rare dependence through memory actually occurs. As with our HLS technique, there is no schedule planned in advance: the schedule develops dynamically as operands become available. Note that the *load-store queue* in the figure is essentially the same component that we employ in our circuits, as we will detail in Section VII-A. Modern processors can also execute instructions speculatively (e.g., even if the outcome of a preceding branch is yet unknown or if the existence of a dependence through memory has not yet been ascertained)—again, a feature that our HLS technique also sports, as we will discuss in Section VII-B. In these processors, the *reorder buffer* is the key component to implement speculative execution and squash results of incorrectly executed instructions.

Another class of processors which exploit instruction-level parallelism are VLIW processors (a term introduced by Josh Fisher in 1983 [28]), illustrated in Figure 5(b). In VLIWs, the problem of deciding how early instructions can be executed and which ones can be issued in parallel is left completely to the compiler: the hardware simply fetches at once groups of operations which can be performed together (the *very long instructions*) and executes them without checking for operand availability.

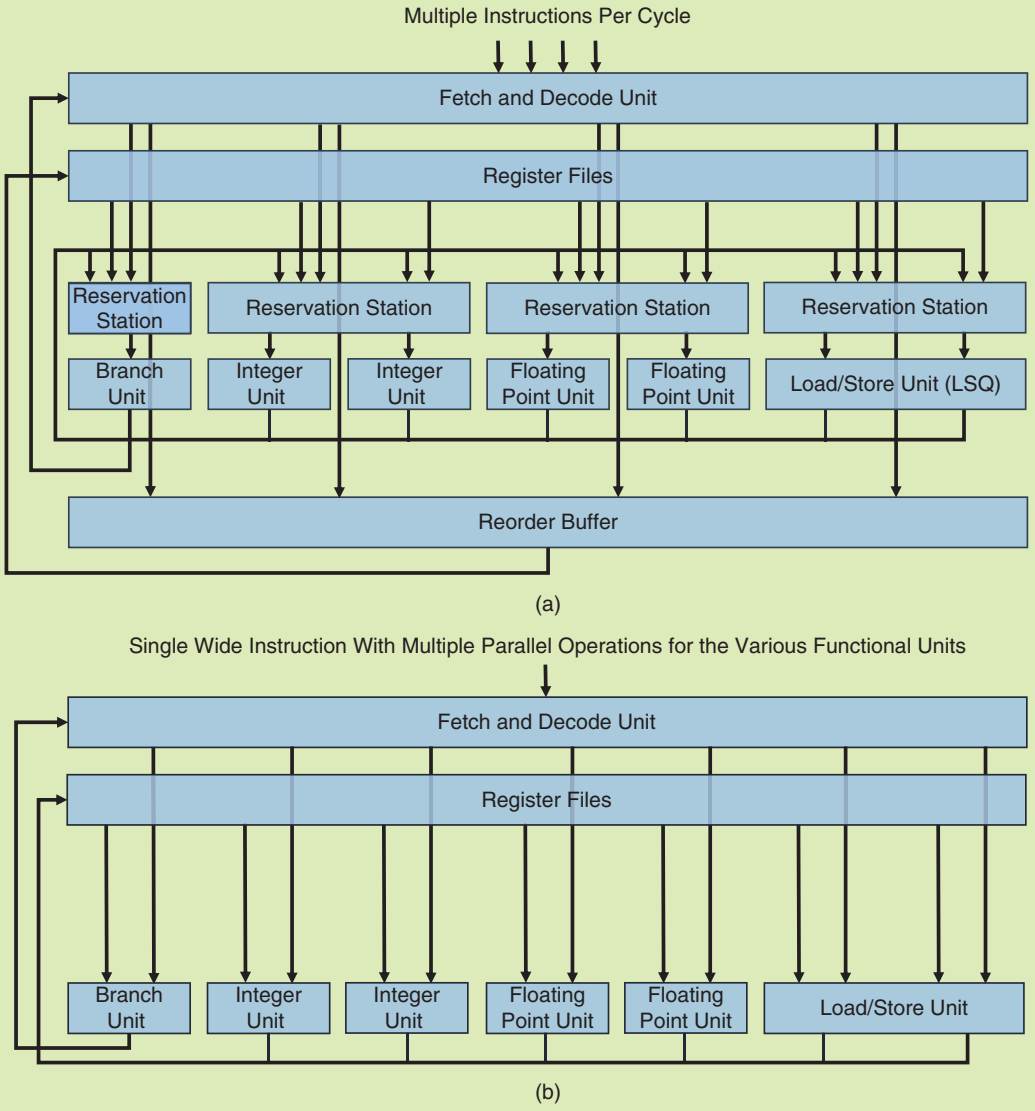


Figure 5. A superscalar and a VLIW processor. Whereas the VLIW processor (Figure 5(b)) executes instructions without checking for dependences, thus relying on the compiler to have prepared a correct schedule, the superscalar processor (Figure 5(a)) contains hardware which supports out-of-order behavior; the schedule is developed dynamically during execution.

The program is effectively a schedule computed statically by the compiler, exactly as in the case of statically-scheduled HLS. The appeal is clear: these processors do not need all the complex hardware that superscalar out-of-order processors require to make runtime decisions.

Researchers quickly figured out that VLIW processors need very sophisticated compilers. Today, a large body of literature exists on VLIW compilation techniques but such techniques often require either complex heuristics to drive the optimizations or pragmas from the programmers. Conversely, dynamically scheduled out-of-order processors achieve good levels of parallelism on-the-fly and without extensive code preparation, yet at the price of more complex hardware. Many of the key transformations to exploit fine-grain parallelism between operations in statically scheduled HLS derive directly from VLIW compilation techniques, such as trace scheduling, software pipelining, and modulo scheduling [27], [46], [58]. But, exactly as HLS tools producing statically scheduled circuits, VLIWs suffer when handling code with irregular memory or control dependences.

The dichotomy in computer architecture may tell us something about the future of dynamically scheduled HLS. In the mid-1990s, Hewlett-Packard and Intel partnered to develop the first (and, to date, only) general-purpose VLIW processor: Itanium. Servers using Itanium shipped mostly in the 2000s and were a major commercial failure [23]. Today, VLIWs thrive exclusively in markets with extremely regular and predictable applications, and where it is acceptable for skilled developers to tune code manually. They are, for instance, in our smartphones where they run complex digital signal processing applications. However, general-purpose, irregular, and control-dominated computing tasks require the runtime flexibility of dynamic scheduling. Even in cost-sensitive devices such as smartphones, none of the processors which run operating systems (e.g., Android and iOS) are VLIWs.

Today, with FPGAs moving to data centers and facing broader application classes, HLS tools might have to satisfy the needs of general-purpose markets as well. Apart from the advantage of exploiting parallelism in cases where static scheduling cannot, the ability of dynamic scheduling to find an acceptable solution without the programmer's help may be critical in a future where HLS will not be driven by hardware designers (available to study the generated circuits and to restructure the input code) but by higher-level code generation tools (e.g., Delite [31]) and, ultimately, by software programmers.

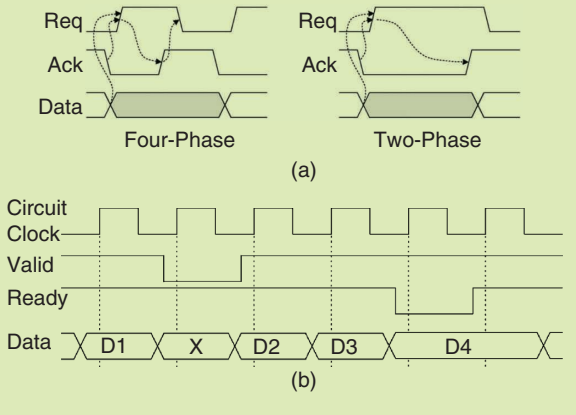


Figure 6. (a) Asynchronous and (b) synchronous latency-insensitive protocols. We here consider synchronous dataflow circuits. This figure is adapted from the work by Cortadella et al. [19].

V. From High-Level Code to a Dynamically Scheduled Circuit

In this section, we outline our HLS methodology which produces dynamically scheduled circuits out of C/C++ code. We first provide an overview of the latency-insensitive design paradigm; we then discuss the dataflow primitives we use and, finally, we describe our HLS conversion strategy.

A. Latency-Insensitive Protocols

Latency-insensitive protocols [8], [18] implement dynamically scheduled dataflow circuits. These circuits are built out of dataflow *units* which exchange pieces of data (referred to as *tokens* [50]) through *channels* composed of data lines and paired with handshake control signals: a *valid* signal indicates the availability of a piece of data and the *ready* signal indicates the readiness of a unit to accept new data. 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.

The latency-insensitive communication strategy originates from the asynchronous circuit domain. Figure 6(a) illustrates two commonly used asynchronous protocols which employ a pair of request and acknowledge signals to regulate data transfers. In the 4-phase protocol, a communication cycle involves four events (i.e., rising and falling edges) and the handshake signals return to zero at the end of each data transfer [29]. In the two-phase protocol, each cycle involves only two events (i.e., either rising or falling edges of the handshake signals) [61].

In the rest of this article, we consider a *synchronous* latency-insensitive protocol: the initiation and completion of data transfers are indicated by the value of the handshake signals at the rising clock edges [18], as illustrated in Figure 6(b). Our perfectly synchronous designs are therefore compatible with traditional VLSI

and FPGA flows and directly comparable to standard HLS techniques.

B. Dataflow Units

In addition to standard functional units, dataflow circuits require specialized units which control the flow of data between units, as illustrated in Figure 7:

- 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 *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 *merge* is a nondeterministic unit that propagates a token received on any of its input to its single output.
- A *control merge (cmerge)* is a merge that, apart from the data output, has an output which indicates which of the inputs was taken by the merge.
- A *mux* is a deterministic version of the merge; it propagates to its single output the input token selected by a control input.
- A *source* can always issue a valid token to its single successor (e.g., a constant).
- A *sink* is always ready to consume tokens from its single predecessor; tokens are simply discarded in the sink.

In addition, like any circuit produced by HLS, dataflow circuits employ any functional unit the code requires, such as integer and floating-point units. Units that require multiple operands contain a join to trigger the operation only when all inputs are available. Our circuits will require *buffers* which serve as registers in standard synchronous designs—we will discuss their properties and placement in Section VI-A. Finally, our circuits will interface to memory using read and write ports, yet, interfacing to memory is challenging due to the out-of-order nature of our system; we will address this issue in Section VII-A.

C. C-to-Dataflow Conversion

This section informally describes a way to transform a standard static single assignment (SSA) intermediate representation [64] into a functionally correct dataflow circuit. We detail the formal correctness and liveness properties in our previous work [40].

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. Each BB receives *live-in* variables from the predecessor BBs and produces *live-out* variables for the successor BBs. This typical compiler intermediate representation propagates a piece of data directly from a source operation to any number of its consumers. In a dataflow circuit, the data and control must be strictly coupled and the number of tokens must exactly match the number of distinct uses, i.e., an operation should be triggered exactly the number of times that it executes in the original program.

Implementing Control Flow

To guarantee that data is always accompanied by control, the following must hold: (1) every BB must provide data exclusively to its immediate successor BBs and (2) every BB must receive data exclusively from its immediate predecessor BBs. 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 at the BB entry may receive multiple inputs at the same time; we need to ensure

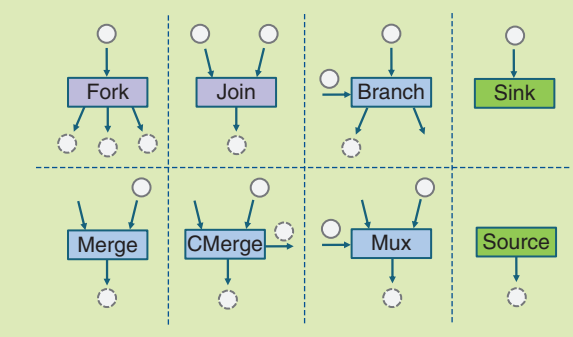


Figure 7. Dataflow units. Although not shown in the figure, all data channels are paired with bidirectional control signals, which indicate the validity of data and the readiness of the successor unit to accept it.

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, as illustrated in Figure 8. Whenever a mux is guaranteed to receive its inputs in order (e.g., when there is a single value propagating through the CFG and a token can only enter a BB from its single active predecessor), it may be disconnected from the cmerge and replaced with a simpler merge unit.

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 unit (i.e., a functional unit with inputs and outputs accompanied by handshake signals). As every data exchange must be represented with an explicit token transfer (i.e., handshake exchange), units with multiple successors require a fork to replicate the output token into a token for each of the successors. Unused unit outputs (e.g., branch outputs without successors) connect to sinks which discard the unused tokens.

VI. Bringing HLS Optimizations to Dataflow Circuits

In this section, we present techniques that make the circuits produced with our HLS approach competitive to standard HLS solutions: we first describe how to pipeline dataflow circuits and then discuss how to save resources through time-multiplexing of functional units.

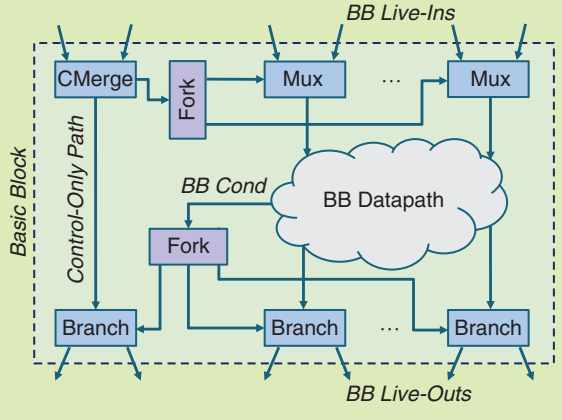


Figure 8. Basic block organization. Every live-in enters a BB through a mux and every live-out exits the BB through a branch. A control-only network (left of the figure) regulates the ordering of tokens at the mux inputs.

A. Pipelining

Dataflow circuits are naturally capable of pipelining, as the fine-grain handshake mechanism allows certain operations to run ahead and, consequently, enables executions of different operations to overlap. Yet, pipelining is not always possible due to *backpressure*: some paths take a longer time to consume a token and prevent potentially quick and independent paths from processing tokens at a high rate. This issue is illustrated on the left of Figure 9(a), showing the dataflow circuit implementing the code in the bottom of the figure: the fork could, in principle, issue tokens to the load on every cycle, but the path to the store stalls the first token until the multiplication completes, hence preventing new tokens from issuing to the load and limiting loop pipelining; the achieved schedule will, essentially, correspond to a nonpipelined schedule of a static HLS tool. Classic pipelining algorithms that standard tools exploit are not applicable in the absence of a static schedule; the solution here is to systematically identify and resolve backpressure to achieve the same pipelining effect.

Just like standard synchronous circuits, dataflow circuits require buffers, i.e., registers, which break combinational paths and, possibly, reduce the critical path of the circuit. Yet, in contrast to standard circuits, buffers can be placed on any channel (i.e., between any two dataflow units) without compromising the circuit functionality. This property can be exploited to mitigate backpressure by inserting buffers into the paths that create stalls and lower system throughput, as illustrated on the right of Figure 9(a). Buffers used for regulating throughput typically have a larger capacity (i.e., they are implemented as FIFOs with multiple data slots) to hold all tokens issued by the predecessor before the successor is ready to accept them—in this example, the buffer requires 3 slots to constantly consume tokens from the fork; without the backpressure on the fork, the iterator loop can issue a new token on every cycle and achieve a perfect pipeline with an II equal to 1.

To optimize the performance of dataflow circuits by strategically placing and sizing buffers, we have developed a mixed-integer linear programming model [43] based on Petri net theory [50]. This model allows for resource-optimal buffer placement and sizing, with the purpose of maximizing throughput of the performance-critical loops at the desired clock frequency.

B. Resource Sharing

Standard HLS tools perform scheduling in conjunction with resource allocation and sharing [70]; depending

on the optimization objective, they trade off area and performance by deciding the cycle in which each operation executes and allocating units accordingly. The top of Figure 9(b) shows two possible schedules for the code in the figure. The first schedule is unconstrained in resources; by scheduling both multiplications in the same cycle, it employs two multipliers to achieve the ideal loop pipeline with an initiation interval of 1. The second schedule enforces a resource constraint of one multiplier; each multiplication must be scheduled on every second cycle and causes an increase of the II to 2.

Dataflow circuits face the same optimization objectives and area-performance trade-offs. Yet, in the absence of a predetermined schedule, it is challenging to determine which operations can share a functional unit without a performance penalty. Intuitively, one could rely on statistical information on unit utilization to decide what to share, as illustrated in the bottom of Figure 9(b): if the two multipliers are fully utilized (i.e., fully occupied with tokens), sharing would damage throughput; if they are only half-utilized, one could employ a single multiplier instead that would always be filled with tokens. Yet, this approach on its own may still compromise performance because the execution of some operations may be delayed with respect to their execution in the original circuit. More critically, although sharing seems to imply only some trivial circuitry, time-multiplexing units in dataflow circuits may cause deadlock by blocking certain data transfers and preventing operations from executing. Hence, a crucial step in making dataflow circuits resource-competitive with standard HLS is to systematically identify good sharing opportunities in an absence of a predetermined schedule, but also to build a sharing mechanism that always results in functional dataflow circuits.

The performance optimization model described in the previous section can be used directly to determine the flow of data through each functional unit and provides us with information on unit utilization; we can exploit this information to decide which units to share. Furthermore, the performance-limiting delays caused by sharing can be resolved through appropriate buffering, achieved by the same optimization technique. Employing a local scheduler at each unit input to regulate the multiplexing of the incoming tokens can avoid unit starvation and ensure the absence of deadlock [17], [32].

VII. Introducing Features of Superscalar Processors to HLS

The optimizations from the previous section enable dataflow circuits to achieve high-throughput pipelines and to share resources, just like standard HLS circuits

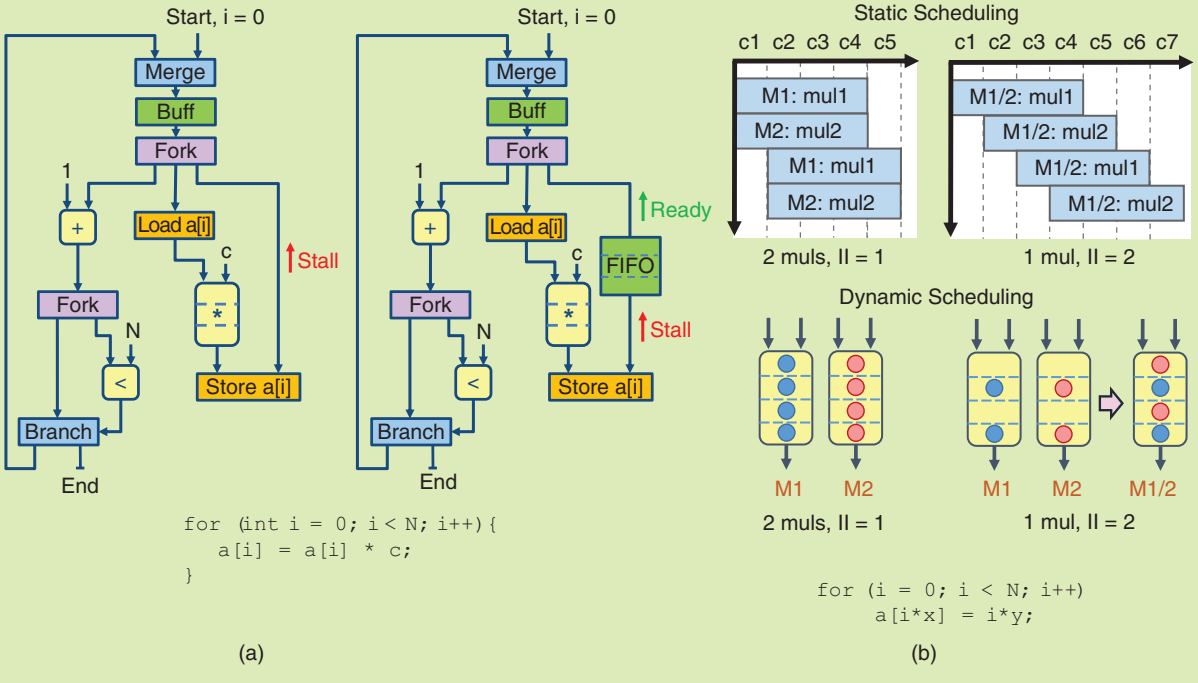


Figure 9. Figure 9(a) illustrates the role of buffer (i.e., FIFO) insertion in resolving backpressure and pipelining a dataflow circuit. Figure 9(b) contrasts resource sharing in static and dynamic scheduling.

can do. Yet, their ability to adapt the schedule at runtime allows them to support features which are regularly exploited by superscalar processors and beyond the capabilities of standard HLS, such as out-of-order memory interfaces and speculative execution—we describe these features in this section.

A. An Out-of-Order Memory Interface

One of the key enablers of dataflow pipelines lies in the ability to execute memory accesses in an order different than the one specified in the original program. As illustrated in Figure 3, pipelining in cases where memory dependences cannot be determined at compile time may be critical for good performance and is the key to the superiority of dataflow computation to statically scheduled HLS designs.

Out-of-order behavior has been exploited in out-of-order processors for decades [54], [55], [59]: load-store queues are used to ensure that all memory dependences are honored, while independent memory requests may execute out-of-order for performance benefits. Dataflow circuits require the same functionality, but a processor LSQ cannot be employed directly because of a fundamental difference between the two systems, as illustrated in Figure 10: In a processor, the notions of fetching and decoding instructions immediately convey the correct sequential order of requests at the memory interface. In contrast, dataflow circuits lack such notions and the information of the original sequential program order is lost; an LSQ receiving memory requests out of order would not be able to decide which reorderings are legal. Therefore, to employ an LSQ and truly benefit from out-of-order execution, dataflow circuits require an alternative way to perform allocation and to convey the correct order of memory requests to the LSQ [38].

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 in-order control path described in Section V-C—this path forks into the LSQ from each BB whose loads and stores are connected to the LSQ; whenever control flows into this BB (i.e., as soon as a decision has been made to enter a particular BB), it triggers the allocation of its memory accesses to the LSQ. This mechanism is illustrated in Figure 11; it ensures that the LSQ can correctly handle memory accesses arriving in arbitrary order while still respecting data dependences.

Although LSQs enable dataflow circuits to achieve high performance in situations that static scheduling cannot efficiently handle, they imply high resource requirements as well as power and clock degradation when implemented on an FPGA. Hence, it is beneficial to

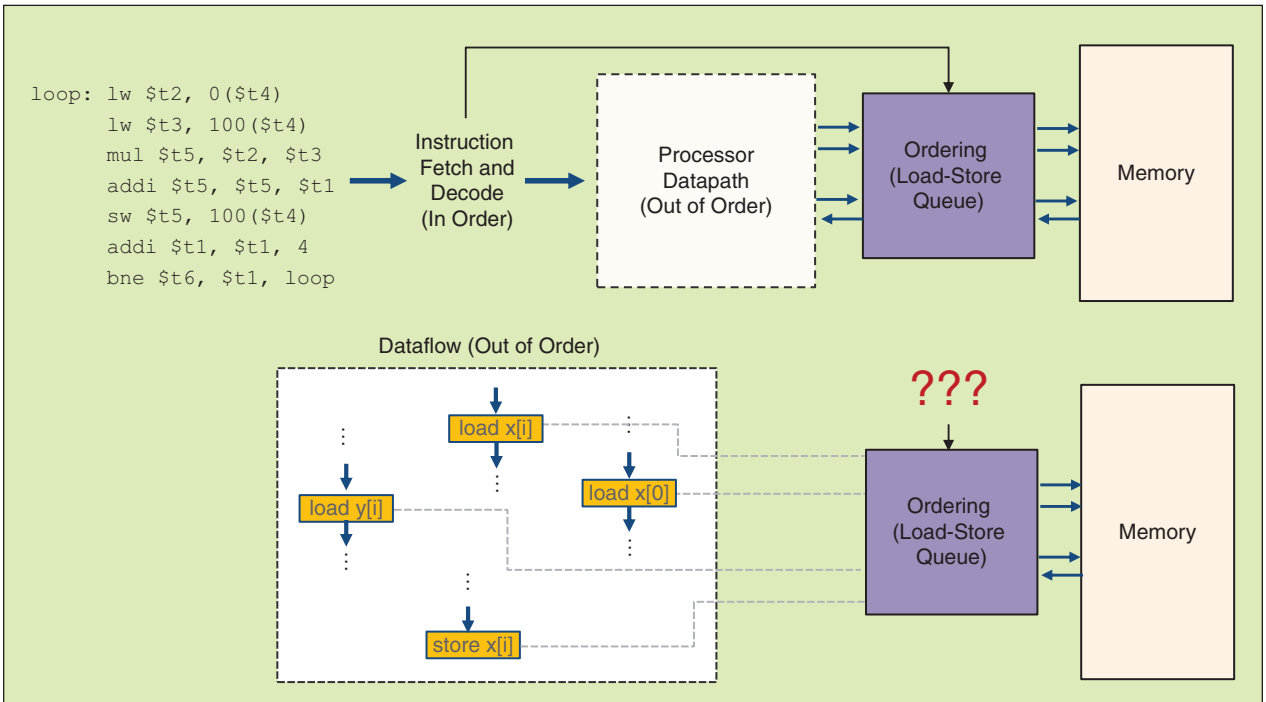


Figure 10. Conveying program order to the memory interface. Program order is crucial to reorder memory accesses at the memory interface. In contrast to a processor, a dataflow system has no notion of fetching and decoding instructions to convey this order.

leverage compiler analysis to simplify the memory interface whenever possible–whenever the compiler can disambiguate memory accesses, groups of accesses that cannot mutually conflict can use separate LSQs, while those which certainly have no dependences with any other access can connect to simple memory interfaces.

The code in Figure 12 shows a loop with multiple memory accesses which are analyzed and optimized using different memory analysis techniques. The information about the memory accesses available in each analysis step is illustrated on the top of the figure: the green dashed edges indicate a possible dependence among accesses which the memory interface must appropriately handle (i.e., if two accesses are certainly or possibly dependent, they require an LSQ). The memory interfaces obtained in different analysis steps are illustrated below. Without any memory analysis (case a) to reason about actual memory dependences, all accesses must connect to a single, large LSQ. By exploiting alias analysis (case b) and determining the memory access patterns using polyhedral techniques (case c), one can determine that some accesses (i.e., those accessing different arrays and those targeting different memory locations, respectively) cannot conflict (i.e., the presence of certain dependences is excluded); the memory interface can be simplified by employing multiple smaller LSQs and removing some LSQs altogether, as the second and the third memory configurations in the figure suggest. Finally, our specialized dataflow analysis [36] can determine that the load and the store to x[i] naturally occur in the correct order as the load produces the data for the store—one can omit the LSQ completely without compromising correctness (case d).

Employing these memory analysis techniques enables us to obtain an optimized memory interface configuration in an out-of-order dataflow system: we profit from the LSQ whenever it is truly useful (such as the situation in Figure 3); otherwise, we remove or simplify it to save resources.

B. Speculation

As in computer architecture, dynamic scheduling paves the way to one of the most powerful ideas in computing: executing some operations before one has the certitude that they are actually needed or that it is correct to execute them. Speculation can significantly improve the execution of loops where the condition on the loop continuation takes very long to compute by predicting very early whether it makes sense to execute *tentatively* another iteration. Similarly, speculation can further improve the problem of memory dependences, not only by reordering accesses when there is no dependence detected, but also by assuming independence early on and reverting back if the prediction was wrong.

The example in Figure 13 illustrates the need to enable speculative execution in HLS. A standard, nonspeculative schedule allows a new loop iteration to start only after the condition to exit the loop (which, in this example, takes multiple cycles to compute) has been checked; therefore, the loop cannot be pipelined. In contrast, a speculative system would achieve the lower schedule which tentatively starts a new loop iteration on every clock cycle, before the loop condition is known. The ability to implement speculation depends on reliable mechanisms to revert state changes due to wrongly executed operations and discard mispeculated values—in processors, this functionality is entrusted to reorder buffers and store queues [33]. In the example in Figure 13, the speculatively computed values from iterations 4 and 5 must be discarded and the result from the third iteration must be returned.

Figure 14 gives a sense of our strategy to implement generic forms of speculation in dataflow circuits [41]. The idea is that some units might be allowed to issue *speculative tokens*—pieces of data which might or might not prove correct and which will combine with other (nonspeculative) tokens, resulting in more speculative tokens traveling through a delimited region of the circuit. In other words, speculative tokens trigger some computations which might have to be squashed and possibly repeated with the correct nonspeculative tokens.

Speculation is triggered by a *speculator*, i.e., a special version of a regular dataflow unit which, besides its standard functionality, can also inject tokens into the circuit before receiving any at its input(s). The most

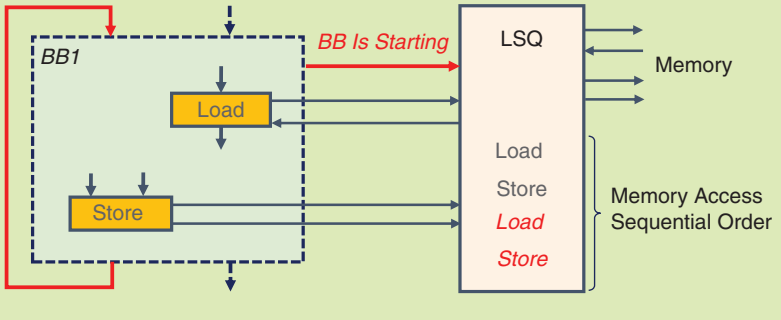


Figure 11. The load-store queue required for correct out-of-order memory accesses. In addition to load and store ports, the LSQ requires a specialized signal indicating the start of each BB in the program-determined order.

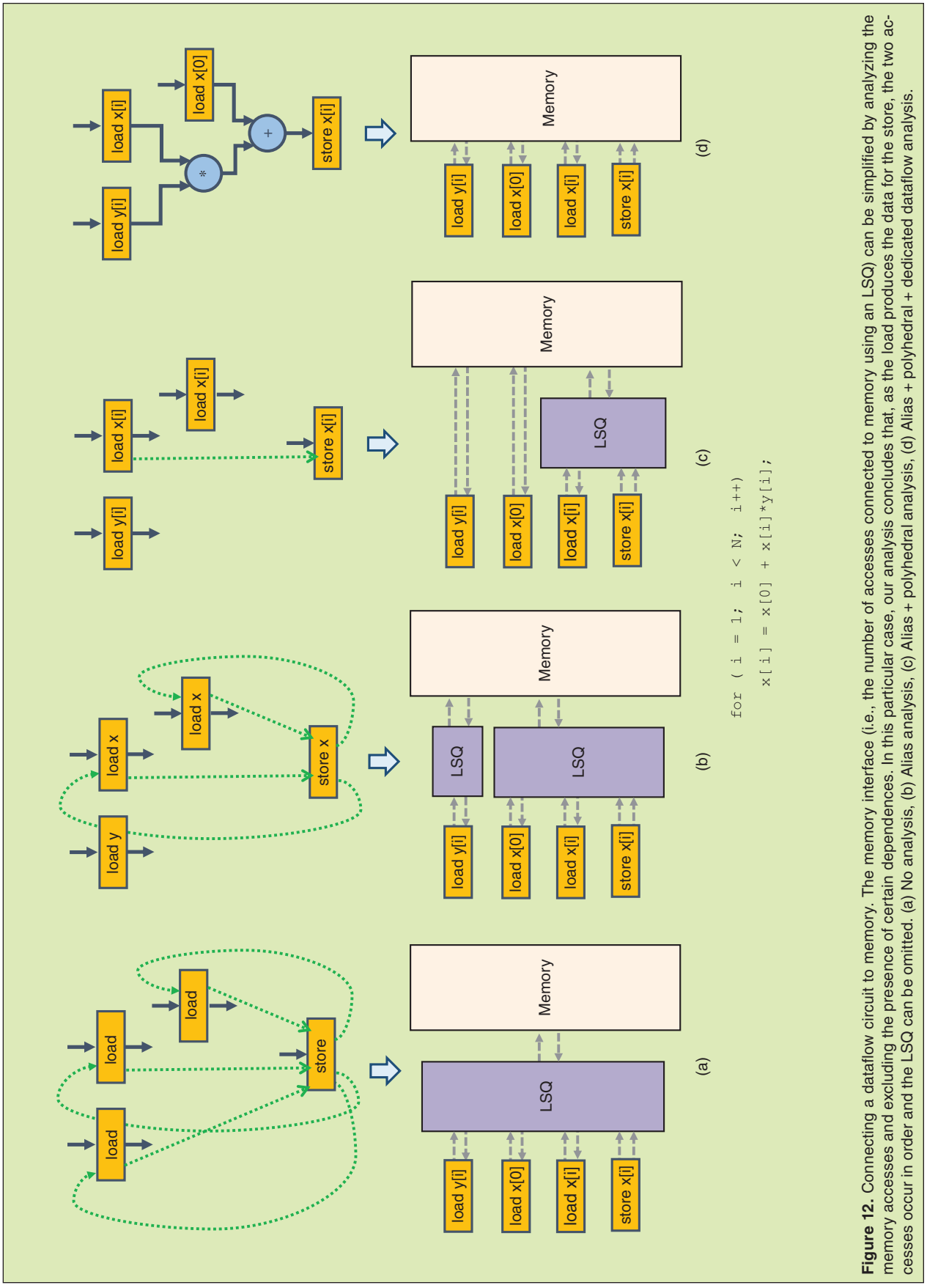


Figure 12. Connecting a dataflow circuit to memory. The memory interface (i.e., the number of accesses connected to memory using an LSQ) can be simplified by analyzing the memory accesses and excluding the presence of certain dependences. In this particular case, our analysis concludes that, as the load produces the data for the store, the two accesses occur in order and the LSQ can be omitted. (a) No analysis, (b) Alias analysis, (c) Alias + polyhedral analysis, (d) Alias + polyhedral + dedicated dataflow analysis.

natural example is that of a branch node which receives the value to dispatch but not the condition; a branch speculator could predict the missing condition and send tentatively the value through one of its outputs. If, after issuing a speculative token, the speculator eventually receives the same data which it assumed speculatively (e.g., the condition it predicted), all is fine and execution was probably sped up; if, on the other hand, the data it eventually receives does not match the prediction, the speculative work must be discarded and the speculator must perform its function correctly (e.g., resend the value on the other branch output).

The speculative region in Figure 14 is bound on its input and output sides with specialized units which implement a squash-and-replay mechanism: *Save* units on the region inputs save a copy of all regular tokens which may combine with a speculative token and reissue them if the previous computation is squashed. *Commit* units at the region output let propagate further speculative results which turn out to be correct and simply squash misspeculated values. Because commit units must differentiate speculative from nonspeculative tokens (the former ones need explicit commit information before propagating, while the latter ones can always go ahead), all channels between the speculator and the commit units must be enriched with a control signal which indicates the speculativeness of the token being passed, as Figure 14 suggests.

Our generic framework can implement broad classes of speculation, which is beyond the capabilities of statically scheduled HLS; in the example of Figure 13, our technique achieves a perfect pipeline shown in the bottom schedule. As we will demonstrate in the following section, our speculation approach is of significant performance advantage in situations where waiting for a key execution decision is particularly time-consuming.

VIII. Evaluation

In this section, we compare our dynamically scheduled circuits with a commercial, statically-scheduled HLS tool.

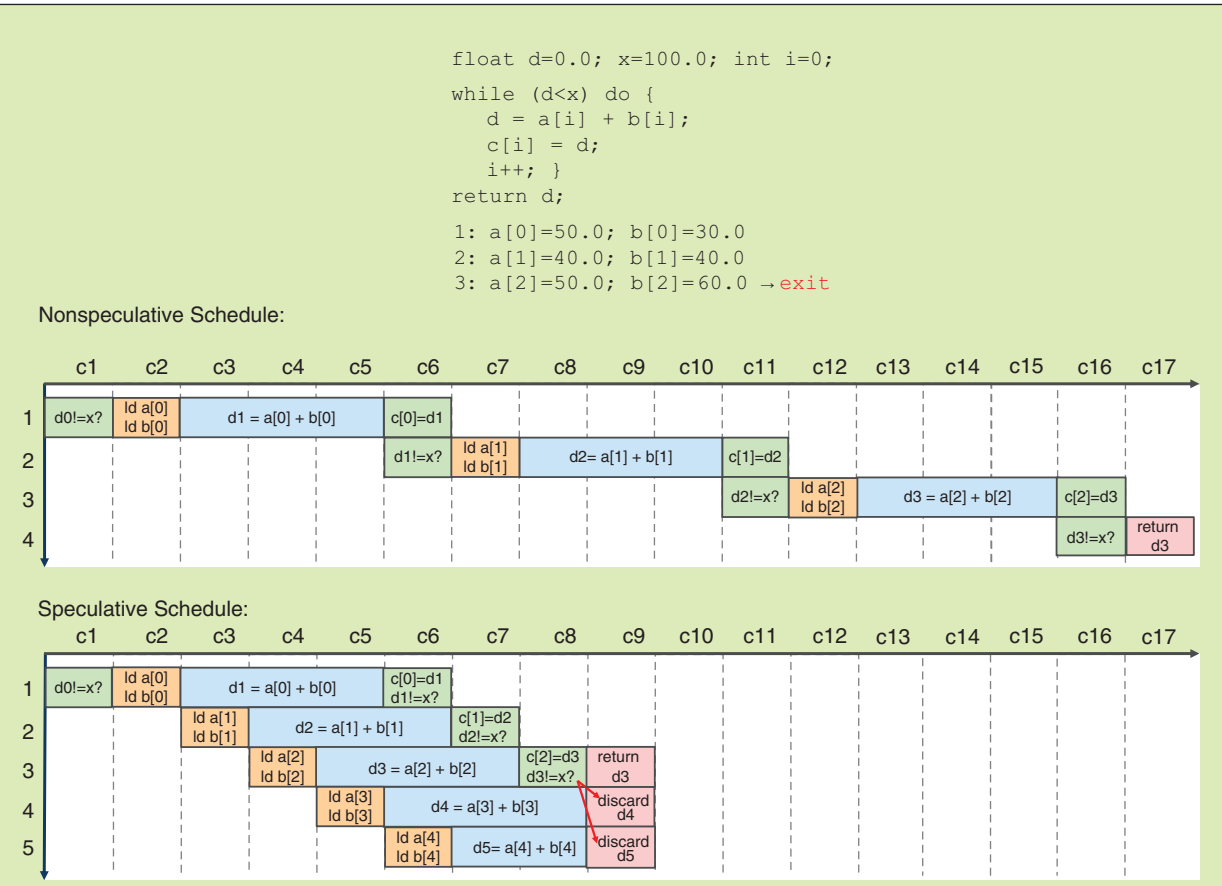


Figure 13. A nonspeculative schedule, compared to a schedule achieved by a system supporting speculation. In this example, the condition to execute another loop iteration takes multiple cycles to compute. Hence, a nonspeculative circuit needs to wait for the condition, whereas the speculative circuit tentatively starts another iteration and then discards the newly computed values if they are later on determined unneeded.

Our complete HLS tool and the benchmarks we explore in this section are publicly available at dynamatic.epfl.ch.

A. Dynamatic HLS Compiler

The C-to-dataflow conversion and the optimizations we discussed in the previous sections are implemented in Dynamatic, our open-source HLS compiler [42]. Dynamatic takes as input C or C++ code and produces a synthesizable hardware description of the corresponding dataflow circuit. The synthesis step relies on the LLVM compiler framework [48]: 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 units from Section V-B following the transformations described in Section V-C to produce a functionally correct dataflow circuit; other passes perform additional analysis and optimizations (e.g., buffer placement as described in Section VI-A and memory access analysis to create the memory interfaces described in Section VII-A). The result is a dataflow circuit netlist which can be directly converted into a VHDL netlist of dataflow units; together with a predefined dataflow unit library, it can be synthesized into an FPGA bitstream.

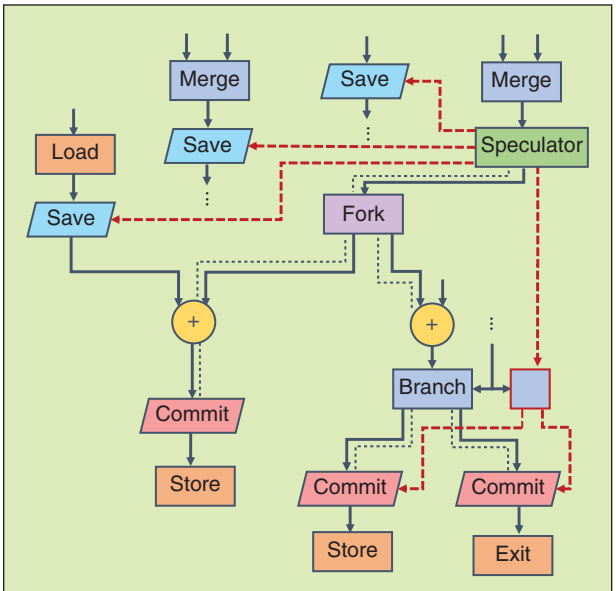


Figure 14. A region of a dataflow circuit implementing our speculative execution paradigm. The speculator initiates speculative execution by injecting tokens tentatively, save units capture required inputs of the region to enable a correct replay in case of misspeculation, and commit units prevent speculative tokens from affecting irreversibly the architectural state, such as memory. Speculative tokens are marked explicitly using an additional bit (represented by the dotted line). A dataflow control circuit (in red, dashed line) between the speculator and the save and commit units carries information about speculative events (start, commit, squash, etc.).

B. Methodology and Benchmarks

To demonstrate the benefits of using dynamic scheduling in HLS, we compare our circuits with designs generated by Vivado HLS [69], a state-of-the-art commercial HLS tool. For a fair comparison, we employ the same arithmetic units and use the same RAMs as Vivado to connect to memory.

We simulate the designs in ModelSim [49] and use a set of test vectors for functional verification. We obtain the average loop initiation interval (II) from the simulation and the clock period (CP) from the post-routing timing analysis to calculate the total execution time. Placing and routing the designs using Vivado gives us the resource usage (i.e., the number of CLB slices, with the corresponding LUT and FF count, as well as the number of DSP units).

Our benchmarks are simple kernels which represent typical cases where static scheduling is known to run into its fundamental limits while dynamic scheduling should make a significant difference. We also consider two simple kernels where static scheduling is fully successful, to show that dynamically scheduling achieves virtually the same result with acceptable overheads.

- *Histogram and Matrix Power* have memory access patterns that cannot be determined at compile time—there may be read-after-write dependences between the stores and the loads from the following iterations.
- *If loop add and If loop mul* have a potential dependence across iterations which depends on the runtime-determined condition (i.e., the condition is determined based on data fetched from memory which is unknown during compilation).
- *Backtrack and Newton-Raphson* have long-latency, data-dependent conditions for starting a new loop iteration and could benefit from branch prediction.
- *FIR and MatVec* are regular kernels that do not have any memory or control dependences.

C. Results: Comparison with Static HLS

Figure 15 shows our results relative to those from Vivado HLS (results to the left or below the red square, which represents all Vivado designs, are better). Table I details our results.

Avoiding conservative assumptions on memory and control dependences results in a significant improvement of the throughput and, consequently, execution

time in all of the corresponding benchmarks. On the *FIR* and *MatVec* benchmarks, static HLS techniques produce highly optimized pipelines because memory accesses can be disambiguated at compile time. The static HLS tool depends on techniques such as modulo scheduling [58] to restructure and pipeline the loop, whereas we effortlessly compile the LLVM IR into a dataflow circuit as-is: although both the static and dynamic design achieve the ideal II of 1, these are the only cases where our results are Pareto-dominated by the static results due to the increase in CP (caused by the additional dataflow logic that we insert into the circuit).

The overhead in slices of the dynamic designs, notable across all benchmarks, is partially due to the control logic that the dataflow circuits contain and which allows them to achieve the latency-insensitivity which we desire. It is immediately visible from Figure 15 that the circuits requiring an out-of-order memory interface demand significant additional resources. It should be emphasized that the resource and timing overhead could be minimized by implementing the LSQs as hard-macros, in the same way as other memory hierarchy components might be in the future (e.g., caches and TLBs). In contrast to the expensive memory interface, our speculation mechanism brings no significant area overhead, yet successfully accelerates all of the corresponding benchmarks by speculating on critical control decisions.

IX. Perspectives

In this section, we evoke some of the most important areas where dynamically scheduled HLS could improve in the future; we outline some research avenues beyond the scope of classic C-based HLS which may benefit from the techniques described in this work.

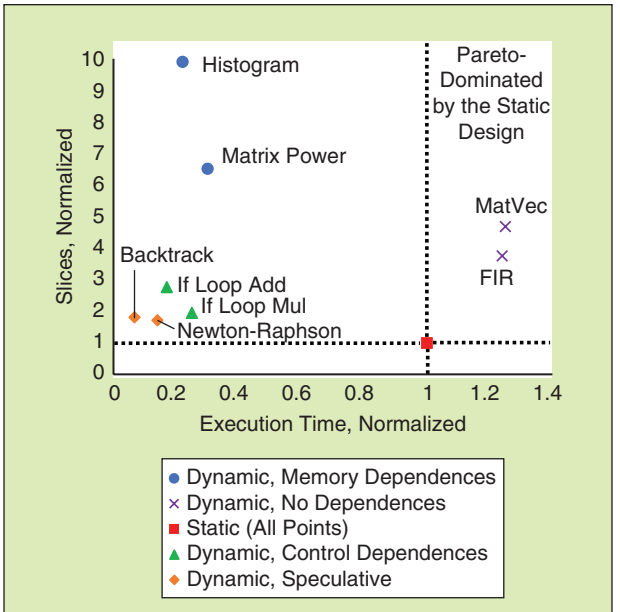


Figure 15. Resource utilization and execution time of the dynamically scheduled designs, normalized to the corresponding static designs produced by Vivado HLS.

A. Partial Schedule Rigidification

One optimization aspect which is immediately manifest when looking at the circuits we generate is that we allow latency insensitivity through any unit and on any path. Although, in some cases, this is exactly the strength of our methodology and the reason for its superiority over standard HLS techniques, in many cases it is an expensive overkill: many computational paths may be constructed with fixed-latency components (ALUs, floating-point operators, etc.) and never really profit from the flexibility of dataflow computation. There may be

Table I.
Dynamically scheduled results (our dataflow circuits) contrasted to statically scheduled results (Vivado HLS). The slice count for the kernels with the LSQ is shown as slices of kernel + slices of LSQ.

	II_{avg}		CP (ns)		Exec. Time (us)		Slices		LUTs		FFs		DSPs	
Benchmark	**STAT**	**DYN**	**STAT**	**DYN**	**STAT**	**DYN**	**STAT**	**DYN**	**STAT**	**DYN**	**STAT**	**DYN**	**STAT**	**DYN**
Histogram	13.0	2.1	3.5	4.9	45.5	10.1	129	220 + 1073	254	4294	510	2033	2	2
Matrix power	13.0	2.7	3.4	4.9	16.8	5.0	200	295 + 1020	340	4463	735	2055	5	5
If loop add	10.0	1.1	3.2	5.0	32.0	5.5	141	393	315	960	525	1318	2	4
If loop mul	7.0	1.1	3.2	5.2	22.4	5.5	177	348	334	892	655	1127	5	5
FIR	1.0	1.0	2.9	3.6	2.9	3.6	47	178	83	463	176	526	3	3
MatVec	1.0	1.0	3.2	4.0	2.9	3.6	63	298	129	843	221	631	3	3
Backtrack	21.0	1.0	3.7	5.1	76.2	5.1	175	320	353	774	625	956	5	7
Newton-Raphson	8.0	1.0	5.4	5.5	4.3	0.6	201	348	585	1181	636	603	9	9

optimizations that rip-off, under some conditions, complex control paths from the corresponding datapaths and replace them with simpler, customized control structures. One could see this as a *selective rigidification* of the schedule where dynamism is not really needed.

The challenge in performing rigidification is to automatically identify which units and paths may be rigid, without compromising performance or circuit correctness. We already exploited Petri net theory to obtain information on the flow of tokens through the dataflow graph—this information may be critical to identify units through which data always flows at a constant rate. These units do not require handshake logic—instead, they could be triggered using a local, predetermined scheduler which ensures that data is received and dispatched at appropriate time intervals; multiple independent schedulers could eventually be merged into a single finite-state machine which would control the entire rigid portion of the circuit, as illustrated in Figure 16. The result would be a hybrid statically and dynamically scheduled circuit which enables the programmer to exploit the 'best of both worlds' [11], depending on the properties of the code: in regular applications, the final result would qualitatively correspond to a statically scheduled circuit; dynamism would remain only in places where it is actually required for performance benefits and at a significantly reduced area overhead.

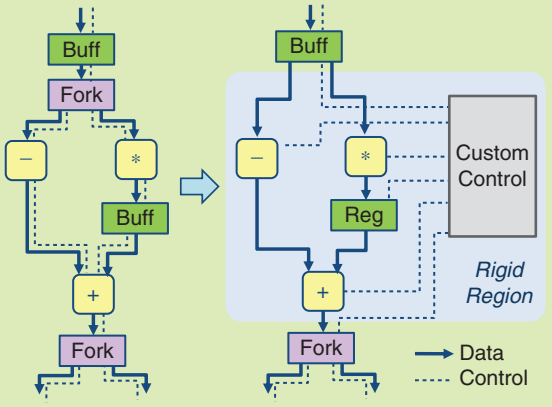


Figure 16. Dataflow circuit rigidification. To simplify the dataflow circuit, the distributed handshake control logic could be replaced with customized control structures whenever dynamism is not required.

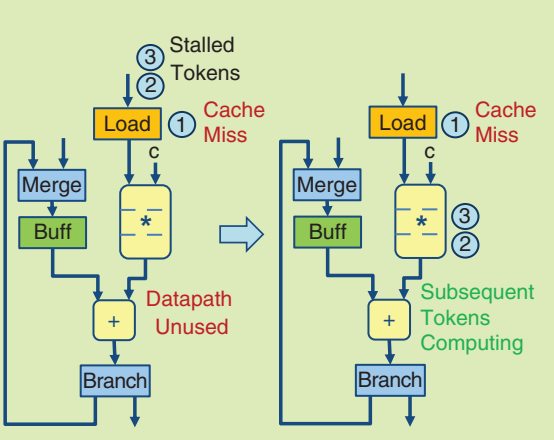


Figure 17. Multithreaded execution. Instead of a single thread which always issues tokens in order into noncyclic paths, one could devise a system where tagged tokens execute out-of-order to increase performance and hardware utilization.

B. Multithreaded Execution

Our current approach targets standard sequential C-based synthesis: there is a single execution thread, i.e., a single token enters through the starting point, propagates through the BBs following the control flow, and exits through the final BB; pipelining is achieved by repeatedly issuing tokens in order into noncyclic paths. Yet, this type of circuit construction may result in limited parallelism and datapath usage in cases where pipelining is not possible (e.g., loop-carried dependences) or when multiple tokens on a noncyclic path are stalled waiting for a long-latency event related to some preceding token (e.g., a cache miss).

Many standard HLS approaches support kernel replication to enable multiple parallel executions on independent copies of the datapath [14], hence fully exploiting the spatial parallelism of the device—the same is perfectly possible with dataflow circuits and could be achieved by high-level transformations (i.e., different input language or intermediate-level kernel replication) in dynamically scheduled HLS. In addition, some authors have looked into pipelining multiple threads on a single kernel (i.e., allowing one thread to execute on the datapath before the previous thread has completed) [62]. Such *multithreading*, analogous to simultaneous multithreading in superscalar processors [66], [67], allows for maximal hardware reuse as resources can be shared among multiple threads. Just like superscalars, dataflow circuits are naturally suited to accommodate such behavior; it could be implemented by inserting multiple *tagged tokens* into the circuit and allowing them to reorder on both cyclic and noncyclic paths, as illustrated in Figure 17. Enabling such multithreaded pipelines requires the creation of an efficient tagging mechanism which allows out-of-order execution wherever it is beneficial and reorders tokens appropriately when needed; similar mechanisms have been explored in dataflow architectures [22] and could be leveraged in this context

as well. Furthermore, such a system would require additional guarantees on the absence of deadlock and appropriate buffering to accommodate the desired number of tokens on each dataflow path. Enabling dataflow kernel replication as well as multithreaded execution on a single kernel has the potential to significantly improve parallelism and resource utilization, hence bringing a completely new optimization dimension to dataflow design.

C. Reconfigurable Dataflow Architectures

So far, we have only attempted to map our dataflow circuits to standard FPGAs—a natural alternative to explore are coarser *reconfigurable arrays*, whose limited flexibility as well as word-oriented nature promise efficiency in area, timing, and energy [35]. The absence of a centralized controller and the systematic pairing of data with handshake signals makes dataflow circuits particularly well-suited for such architectures: each array tile would be composed out of one or more dataflow primitives and the interconnect between tiles would carry data bundled with its control signals, as suggested in Figure 18.

One of the major challenges is to design an array that is structurally adequate for the computational patterns and interconnects which typically appear in dataflow circuits. Intuitively, circuits obtained from high-level code share some representative properties (e.g., BB organization with merges and branches at the inputs and outputs, respectively) which can be exploited to customize the array tiles. Our existing compilation flow can be used to translate a program into a functional netlist of hardware primitives; we would need to develop custom place-and-route techniques which exploit array-specific transformations and optimizations to map these netlists onto the underlying architecture and to enable efficient architectural exploration.

X. Related Work

In this section, we outline what others have done to circumvent some of the problems of statically scheduled HLS and we contrast our work with other dataflow-oriented approaches.

A. Towards Dynamic Scheduling

Recent advances in HLS have explored methods to overcome the conservatism in static scheduling and to remove the inability of HLS tools to handle dynamic events. Several techniques [1, 47] generate multiple schedules which are dynamically selected during runtime, once the values of all parameters are known; they rely on the capabilities of current HLS tools by replicating the source code and dynamically selecting which copy of the code needs to be executed. The drawback of these approaches is that they apply to only some very particular cases of dependences through memory; they are also affected by the area (or reconfiguration) overhead of synthesizing two or more versions of an accelerator and the cost of switching between them.

Tan et al. [63] describe an approach called ElasticFlow to apply loop pipelining on a particular class of irregular loop nests with no inter-iteration dependences in the outer loops. In their approach, multiple pipeline instances of a dynamic-bound inner loop are scheduled to execute in parallel. Dai et al. [20] propose methods for pipeline flushing by performing static scheduling for multiple initiation intervals of the pipeline to resolve different possible resource collisions; they later developed application-specific dynamic hazard detection circuitry [21] and have shown the ability of speculation but with stringent constraints (i.e., the approach lacks generality in the ability to revert arbitrarily the state after failed predictions). Nurvitadhi et al. [53] perform automatic pipelining, assuming that the datapath is already partitioned into pipeline stages. The underlying methodology in all these techniques is still based on static scheduling adapted to enable some level of dynamic behavior, which limits the achievable performance improvements only to some particular cases. We think that this body of recent work points to the importance of the ultimate

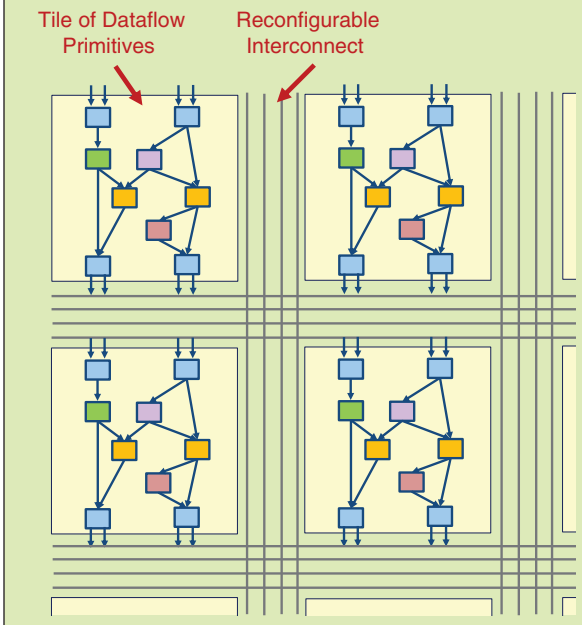


Figure 18. A reconfigurable dataflow array, with tiles composed out of fixed interconnects of dataflow primitives. All connections carry data bundled with its control signals.

solution to the limits of static scheduling: embracing general forms of dynamic scheduling.

B. Dynamic Scheduling in HLS

Different authors exploited latency-insensitive protocols [8], [18], [25] to construct synchronous and asynchronous dataflow circuits. Elastic circuits [18] are probably the best-studied form of latency insensitivity, but the original paradigm used in most of the papers by Cortadella et al. is too restrictive for HLS. Several approaches [10], [34] extended the SELF protocol [18] with constructs similar to the branch and merge which we use in this work. Kam et al. [44] show the ability of elastic circuits to create dynamic pipelines, but do not provide generic transformations to create such circuits out of high-level descriptions. Cheng et al. [12] describe circuits as networks of processes in which hardware accelerators exchange data via dynamic communication channels; similarly, standard HLS tools [69] can interconnect with handshakes various datapaths from nested loops and functions. We are interested in exploring dynamicity on a finer grain (i.e., the schedule of individual datapaths). Efforts in the asynchronous domain, such as Balsa [24] and Haste/TiDE [52], applied syntax-driven approaches for mapping a program into a structure of handshake components [60]; a synchronous backend for Haste/TiDE has later been developed. Putnam et al. [57] also explored synthesizing dataflow-like circuits from high-level specifications. Townsend et al. [65] used a functional programming intermediate representation as a starting point for synthesizing dataflow networks. Dataflow circuits, with their handshake signals, bring to mind Bluespec and its firing rules [68]. However, all these approaches provide little information on some critical conversion aspects and features which are at the heart of this work; to our best knowledge, these approaches have never been contrasted to modern HLS tools.

The efforts closest to ours are the work by Huang et al. [35] and Budiu et al. [4], [5]). Huang et al. generated dataflow circuits from C code, to be mapped to a coarse-grain reconfigurable array [35]. Their circuit generation approach differs from ours in two aspects: (1) They use a single branch node at the output of each basic block, which forces them to synchronize all the basic block outputs and, consequently, prevents loop iterations from overlapping (i.e., loops are not pipelined). (2) Their approach does not employ an LSQ at the memory interface and, thus, all memory accesses which cannot be disambiguated at compile time need to be conservatively sequentialized ("The memory dependence is implemented by creating a lockstep between the corresponding […] memory ports" [35]). Budiu et al. described a compiler for generating *asynchronous circuits* from C code [4], [5]. Although their final circuits are fundamentally different from ours (our circuits are *perfectly synchronous* and avoid the traditional difficulties associated with asynchronous designs), the generation strategy is similar to ours. Unfortunately, the exact methodology is never described in full detail and examples across different papers by the same authors do not seem perfectly consistent; although they also employ an LSQ to handle memory dependences, their allocation policy is more conservative than what we described in Section VII-A: they serialize memory accesses whose dependences cannot be resolved statically ("we insert a token edge between two instructions only if their points-to sets overlap and they do not commute" [5]). Both the approach by Budiu et al. and by Huang et al. largely limit the benefits of dynamic scheduling; although Budiu et al. maintain the LSQ, Huang et al. omit it, most likely due to its seemingly limited value. Our LSQ, with its group allocation policy, enables spatial architectures to fully exploit memory access parallelism; our results show that our strategy achieves highly optimized dynamic scheduling.

XI. Conclusions

High-level synthesis (HLS) tools enable hardware generation from high-level software code; because they provide a higher level of abstraction for accelerator design, their role in the future of reconfigurable computing is critical. Despite their recent commercial success and ability to successfully accelerate certain types of applications, standard HLS tools still heavily rely on manual code optimization, code restructuring, and extensive trial and error with configuration parameters and code annotations. In addition, these tools force worst-case assumptions in irregular applications, where data and control dependences cannot be statically resolved; they also provide only limited support for novel optimizations such as speculative execution. In this article, we described a dynamically scheduled form of HLS which produces dataflow circuits from imperative code. Compared to a commercial HLS tool, the result is a different trade-off between performance and circuit complexity, much as superscalar processors represent a different trade-off compared to VLIW processors: when static HLS exploits the maximum parallelism available, our technique achieves similar results with minimal degradation in cycle time and resources; when static HLS misses some key performance optimization opportunities, our circuits seize them by reordering memory accesses, dynamically resolving control dependences, and speculating on critical control decisions, achieving significant performance improvements with the investment of more resources. The ability of our approach to find these design points without requiring significant code restructuring by the programmer is likely to be extremely important in the future where HLS will

be used by software developers with limited hardware design expertise. We therefore believe that this avenue of HLS has the potential to open new doors for reconfigurable computing and its applications.

Lana Josipović (S'16) received a BSc (2013) and MSc (2015) in Electrical Engineering and Information Technology from the University of Zagreb, Croatia. In 2021, she received a PhD in Computer and Communication Sciences from EPFL, Switzerland. Her research interests include high-level synthesis, compilers, and reconfigurable computing. During her PhD, she developed Dynamatic, an open-source high-level synthesis tool that produces dynamically scheduled circuits from C/C++ code. She is a recipient of the Google PhD Fellowship in Systems and Networking, Google Women Techmakers Scholarship, Best Paper Award at FPGA'20, and Best Paper Award Nominations at FPGA'18 and CASES'17.

Country of residence: Switzerland

Andrea Guerrieri received his M.Sc. degree in Electronic Engineering from Politecnico di Torino, Italy, in 2015. In 2006, he started working on embedded systems in PB Elettronica, Italy, where he became Principal Engineer responsible for the development of the company's flagship products. In 2017, he joined the Processor Architecture Laboratory at Ecole Polytechnique Federale de Lausanne, Switzerland, where he leads and participates in research projects in collaboration with industry. Recent projects involve reconfigurable SoCs, exploiting dynamic partial reconfiguration of FPGAs for future space missions and planet observation. He is also a co-developer of Dynamatic, an open-source dynamically scheduled high-level synthesis tool.

Country of residence: Switzerland

Paolo Ienne (S'90, M'96, SM'10) received the *laurea* degree in Electrical Engineering from *Politecnico di Milano*, Italy, in 1991 and the Ph.D. degree in Computer Science from EPFL, Switzerland, in 1996. Since 2000, he a Professor in the School of Computer and Communication Sciences, EPFL. His research interests include computer and processor architecture, FPGAs and reconfigurable computing, electronic design automation, and computer arithmetic. Some of his articles have received the Best Paper Awards at prestigious venues (including at the FPGA, FPL, CASES, and DAC conferences) and several others have been nominated. Ienne has served as general, programme, and topic chair of renown international conferences, serves on the steering committee of the ARITH, FPL, and FPGA conferences, and is regularly member of several program committees. He was an associate editor of the ACM TODAES and is an associate editor of ACM CSUR and ACM TACO. Ienne has published over 200 articles in peer reviewed journals and international conferences. He is a Senior Member of the IEEE and a Member of the ACM.

Country of residence: Switzerland

References

[1] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proc. 50th Design Automation Conf.*, Austin, TX, June 2013, pp. 1–10. doi: 10.1145/2463209.2488796.

[2] "Amazon EC2 F1 instances." Amazon.com. 2017.

[3] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Commun. ACM*, vol. 56, no. 4, pp. 56–63, Apr. 2013. doi: 10.1145/2436256.2436271.

[4] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *Proc. IEEE Int. Symp. Performance Analysis Syst. Softw.*, Austin, TX, Mar. 2005, pp. 177–186.

[5] M. Budiu and S. C. Goldstein, "Pegasus: An efficient intermediate representation," Carnegie Mellon University, Tech. Rep. CMU-CS-02-107, May 2002.

[6] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *Proc. 23rd Int. Conf. Field-Programmable Logic Appl.*, Munich, Sept. 2014, pp. 1–8.

[7] A. Canis et al., "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 13, no. 2, pp. 24:1–24:27, Sept. 2013. doi: 10.1145/2514740.

[8] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1059–1076, Sept. 2001. doi: 10.1109/43.945302.

[9] A. M. Caulfield et al., "A cloud-scale acceleration architecture," in *Proc. 49th Int. Symp. Microarchitecture*, Taipei, Taiwan, Oct. 2016, pp. 1–13.

[10] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "xMAS: Quick formal modeling of communication fabrics to enable verification," *IEEE Design Test Comput.*, vol. 29, no. 3, pp. 80–88, June 2012. doi: 10.1109/MDT.2012.2205998.

[11] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proc. 28th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Seaside, CA, Feb. 2020, pp. 288–298.

[12] S. Cheng and J. Wawrzynek, "Synthesis of statically analyzable accelerator networks from sequential programs," in *Proc. Int. Conf. Comput.-Aided Design*, Austin, TX, Nov. 2016, pp. 126–133.

[13] D. Chiou, "Intel acquires Altera: How will the world of FPGAs be affected?" in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 2016, Monterey, p. 148.

[14] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Technol.*, Kyoto, Dec. 2013, pp. 270–277.

[15] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011. doi: 10.1109/TCAD.2011.2110592.

[16] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proc. 43rd Design Automation Conf.*, San Francisco, July 2006, pp. 433–438.

[17] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky, "Elastic systems," in *Proc. 10th ACM/IEEE Int. Conf. Formal Methods Models Codesign*, July 2010, pp. 149–158.

[18] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proc. 43rd Design Automation Conf.*, San Francisco, July 2006 pp. 657–662.

[19] J. Cortadella, M. G. Oms, M. Kishinevsky, and S. S. Sapatnekar, "RTL synthesis: From logic synthesis to automatic pipelining," *Proc. IEEE*, vol. 103, no. 11, pp. 2061–2075, Nov. 2015. doi: 10.1109/JPROC.2015.2456189.
[20] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," in *Proc. 51st Design Automation Conf.*, San Francisco, June 2014, pp. 1–6.
[21] S. Dai et al., "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proc. 25th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, Feb. 2017, pp. 189–194.
[22] R. Desikan, S. Sethumadhavan, D. Burger, and S. W. Keckler, "Scalable selective re-execution for EDGE architectures," in *Proc. 11th Int. Conf. Architectural Support Program. Languages Oper. Syst.*, Boston, Oct. 2004, pp. 120–132.
[23] J. C. Dvorak, "How the Itanium killed the computer industry," Jan. 2009.
[24] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *Comput. J.*, vol. 45, no. 1, pp. 12–18, Jan. 2002. doi: 10.1093/comjnl/45.1.12.
[25] S. A. Edwards, R. Townsend, and M. A. Kim, "Compositional dataflow circuits," in *Proc. 15th ACM-IEEE Int. Conf. Formal Methods Models Syst. Design*, Vienna, Sept. 2017, pp. 175–184.
[26] M. Fingeroff, *High-Level Synthesis Blue Book*, 1st ed. Xlibris Corp., 2010.
[27] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. 30, no. 7, pp. 478–490, July 1981. doi: 10.1109/TC.1981.1675827.
[28] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. 10th Annu. Int. Symp. Comput. Architecture*, Stockholm, June 1983, pp. 140–150. doi: 10.1145/800046.801649.
[29] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 4, no. 2, pp. 247–253, June 1996. doi: 10.1109/92.502196.
[30] N. George et al., "Automatic support for multi-module parallelism from computational patterns," in *Proc. 24th Int. Conf. Field-Programmable Logic Appl.*, London, Sept. 2015, pp. 1–8.
[31] N. George et al., "Hardware system synthesis from domain-specific languages," in *Proc. 23rd Int. Conf. Field-Programmable Logic Appl.*, Munich, Sept. 2014, pp. 1–8.
[32] J. Hansen and M. Singh, "Multi-token resource sharing for pipelined asynchronous systems," in *Proc. Design, Automation Test Europe Conf. Exhib.*, Dresden, Mar. 2012, pp. 1191–1196.
[33] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Mateo, CA: Morgan Kaufmann, 2011.
[34] G. Hoover and F. Brewer, "Synthesizing synchronous elastic flow networks," in *Proc. Design, Automation Test Europe Conf. Exhib.*, Munich, Mar. 2008, pp. 306–311.
[35] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in *Proc. 21st ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, Feb. 2013, pp. 171–180.
[36] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne, "Shrink it or shed it! minimize the use of LSQs in dataflow designs," in *Proc. IEEE Int. Conf. Field Programmable Technol.*, Tianjin, Dec. 2019, pp. 197–205.
[37] L. Josipović, P. Brisk, and P. Ienne, "From C to elastic circuits," in *Proc. 51st Annu. Asilomar Conf. Signals, Syst., Comput.*, Pacific Grove, Nov. 2017, pp. 121–125.
[38] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 16, no. 5s, pp. 125:1–125:19, Sept. 2017. doi: 10.1145/3126525.
[39] L. Josipovic, N. George, and P. Ienne, "Enriching C-based high-level synthesis with parallel pattern templates," in *Proc. 26th IEEE Int. Conf. Field Programmable Technol.*, Xian, Dec. 2016, pp. 177–180.
[40] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proc. 26th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, Feb. 2018, pp. 127–136.
[41] L. Josipović, A. Guerrieri, and P. Ienne, "Speculative dataflow circuits," in *Proc. 27th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Seaside, Feb. 2019, pp. 162–171.
[42] L. Josipović, A. Guerrieri, and P. Ienne, "Dynamatic: From C/C++ to dynamically scheduled circuits," in *Proc. 28th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Seaside, Feb. 2020, pp. 1–10.
[43] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *Proc. 28th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Seaside, Feb. 2020, pp. 186–196.
[44] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *Proc. 27th Int. Conf. Comput.-Aided Design*, Nov. 2008, pp. 434–441.
[45] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs," May 2018, arXiv:1805.03648.
[46] M. S. Lam. "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. 1988 ACM Conf. Program. Language Design Implementation*, Atlanta, GA, June 1988, pp. 318–328.
[47] J. Liu, S. Bayliss, and G. A. Constantinides. "Offline synthesis of online dependence testing: Parametric loop pipelining for HLS," in *Proc. 23rd IEEE Symp. Field-Programmable Custom Comput. Mach.,* Vancouver, May 2015, pp. 159–162.
[48] The LLVM Compiler Infrastructure, 2018. http://www.llvm.org
[49] "ModelSim." Mentor Graphics, 2016.
[50] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989. doi: 10.1109/5.24143.
[51] R. Nane et al., "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016. doi: 10.1109/TCAD.2015.2513673.
[52] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the Haste/TiDE design flow," in *Proc. 15th Int. Symp. Asynchronous Circuits Syst.*, Chapel Hill, N.C., May 2009, pp. 185–194.
[53] E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. L. Lu, "Automatic pipelining from transactional datapath specifications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 3, pp. 441–454, Mar. 2011. doi: 10.1109/TCAD.2010.2088950.
[54] I. Park, C. L. Ooi, and T. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proc. 36th Int. Symp. Microarchitecture*, San Diego, Dec. 2003, pp. 411–422.
[55] M. Pericàs et al., "A two-level load/store queue based on execution locality," in *Proc. 35th Int. Symp. Comput. Architecture*, Beijing, June 2008, pp. 25–36. doi: 10.1145/1394608.1382171.
[56] A. Putnam et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. 41st Int. Symp. Comput. Architecture*, Minneapolis, June 2014, pp. 13–24.
[57] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures," in *Proc. 16th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, Feb. 2017, pp. 173–178.
[58] B. R. Rau, "Iterative modulo scheduling," *Int. J. Parallel Program.*, vol. 24, no. 1, pp. 3–64, Feb. 1996. doi: 10.1007/BF03356742.
[59] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. "Late-binding: Enabling unordered load-store queues," in *Proc. 34th Int. Symp. Comput. Architecture*, San Diego, June 2007, pp. 347–357.
[60] J. Sparsø, "Current trends in high-level synthesis of asynchronous circuits," in *Proc. 16th IEEE Int. Conf. Electron., Circuits, Syst.*, Yasmine Hammamet, Dec. 2009, pp. 347–350.
[61] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, June 1989. doi: 10.1145/63526.63532.
[62] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded pipeline synthesis for data-parallel kernels," in *Proc. Int. Conf. Comput.-Aided Design*, San Jose, Nov. 2014, pp. 718–725.
[63] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "ElasticFlow: A complexity-effective approach for pipelining irregular loop nests," in *Proc. 34th Int. Conf. Comput.-Aided Design*, Austin, TX, Nov. 2015, pp. 78–85.
[64] L. Torczon and K. Cooper, *Engineering a Compiler*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 2011.
[65] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proc. 26th Int. Conf. Compiler Construction*, Austin, TX, Feb. 2017, pp. 76–86.
[66] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proc. 23rd Annu. Int. Symp. Comput. Architecture*, Philadelphia, May 1996, pp. 191–202.
[67] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22nd Annu. Int. Symp. Comput. Architecture*, Santa Margherita Ligure, May 1995, pp. 392–403.
[68] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *Proc. 9th Int. Conf. Formal Methods Models Codesign*, Cambridge, MA, July 2009, pp. 171–180.
[69] "Vivado High-Level Synthesis," Xilinx Inc, 2018.
[70] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *Proc. 32nd Int. Conf. Comput.-Aided Design*, San Jose, CA, Nov. 2013, pp. 211–218.