From C/C++ Code to High-Performance Dataflow Circuits

Lana Josipović, *Member, IEEE*, Andrea Guerrieri, and Paolo Ienne, *Senior Member, IEEE*

***Abstract*—High-level synthesis (HLS) tools typically generate statically scheduled datapaths. Static scheduling implies that the resulting circuits have a hard time exploiting parallelism in code with potential memory dependences, with control dependences, or where performance is limited by long latency control decisions. In this work, we describe an HLS approach which generates dynamically scheduled, dataflow circuits out of imperative code. We detail a complete set of rules to transform a standard compiler intermediate representation into a high-performance dataflow circuit that is able to dynamically resolve memory dependences and adapt its behavior on the fly to particular control flow decisions and operation latencies. Compared to a traditional HLS tool, the result is a different tradeoff between performance and circuit complexity: statically scheduled circuits display the best performance per cost in regular applications, but general-purpose, irregular, and control-dominated computing tasks require the runtime flexibility of dynamic scheduling. Therefore, enabling dynamic behavior in HLS is key to dealing with the increasing computational demands of new contexts and broader application domains.**

***Index Terms*—Buffer storage, circuit optimization, dataflow computing, high-level synthesis (HLS), memory architecture.**

I. Introduction

THE USE of FPGAs in datacenters by Microsoft [10], [45] and Amazon [2], as well as the acquisition of Altera by Intel [14] signal is one of the greatest opportunities for FPGAs since they were first introduced. One of the many challenges ahead is whether software programmers will ever manage to extract enough performance out of FPGAs through modern programming paradigms. High-level synthesis (HLS) tools generate hardware designs from high-level software descriptions and they are set to play a key role in making FPGAs accessible to diverse users. While there is conspicuous research activity on this front, HLS tools almost universally rely on building datapaths that are controlled following *static schedules*—that is, the cycle when every operation is executed is fixed at synthesis time [19]. Although this approach serves well applications that are fairly regular, it tends to produce conservative and low-performance results in irregular and general-purpose code, thus limiting the usability of HLS only to particular market segments.

An alternative HLS approach is to implement *dynamic scheduling*, where decisions on when each operation should execute are made in the circuit during runtime, hence, achieving behaviors which are beyond the capabilities of statically scheduled circuits: apart from the ability to extract more parallelism when control and memory dependences are undecidable at compile time, dynamic scheduling helps to alleviate the need for complex loop transformation and the related programmer hints. Although beyond the scope of this article, dynamic scheduling also opens the door to speculative execution [34], one of the most powerful ideas in computer architecture. These opportunities to exploit parallelism while minimizing the programming effort may be critical for FPGAs to compete with modern CPUs and, ultimately, to deal with the increasing computational demands of the 21st century.

This article presents a methodology to automatically generate high-performance, dynamically scheduled circuits from C/C++ code. Our approach borrows several ideas from the asynchronous domain, but produces perfectly synchronous designs, which are directly comparable to standard HLS techniques. The remainder of this article is organized as follows. Section II explores a classic situation where the dynamic extraction of operation-level parallelism proves essential to performance. Section III details our circuit generation methodology as implemented in our open-source HLS tool. Section IV discusses the role of buffers in dataflow designs and their impact on performance. The next problem is connecting the design to memory to handle out-of-order memory accesses, which we describe in Section V. In Section VI, we provide an overview of our complete compiler flow and Section VII gives the results of the comparison of our technique with static HLS. In Section VIII, we outline what others have done to circumvent some of the problems of statically scheduled HLS, before concluding this article in Section IX. In addition to our previously published work [33], this article discusses several new aspects of the C-to-dataflow conversion (e.g., ensuring deterministic behavior) and details all concepts that are incorporated into our complete and open-source HLS framework [35].

II. Why Dynamic Scheduling?

To illustrate the limitations of standard HLS approaches, consider the code in Fig. 1(a). In this loop, there is a control flow decision (`if`), which depends on the actual data being read from arrays `a[]` and `b[]`. The operation that might take place in a specific iteration (`s = s + d`) introduces a dependency between iterations and delays the next iteration whenever the condition is true. When pipelining this loop, a typical HLS tool needs to create a static schedule—that is, a conservative execution plan for the various operations in the loop, which is valid in every possible case. Such a schedule

Manuscript received August 10, 2020; revised January 15, 2021 and May 7, 2021; accepted August 2, 2021. Date of publication August 17, 2021; date of current version June 20, 2022. The work of Lana Josipović was supported by the Google Ph.D. Fellowship in Systems and Networking. This article was recommended by Associate Editor Z. Zhang. *(Corresponding author: Lana Josipović.)*

The authors are with the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland (e-mail: lana.josipovic@epfl.ch; andrea.guerrieri@epfl.ch; paolo.ienne@epfl.ch).

Digital Object Identifier 10.1109/TCAD.2021.3105574

```
float d, s = 0.0;
int i;
for (i=0; i<100; i++){
   d = a[i] - b[i];
   if (d >= 0)
      s += d;
}
  a[0]=1.0; b[0]=3.0;
  a[1]=4.0; b[1]=3.0;
  a[2]=2.0; b[2]=2.0;
  a[3]=4.0; b[3]=5.0;
            ⋮
```

(a)

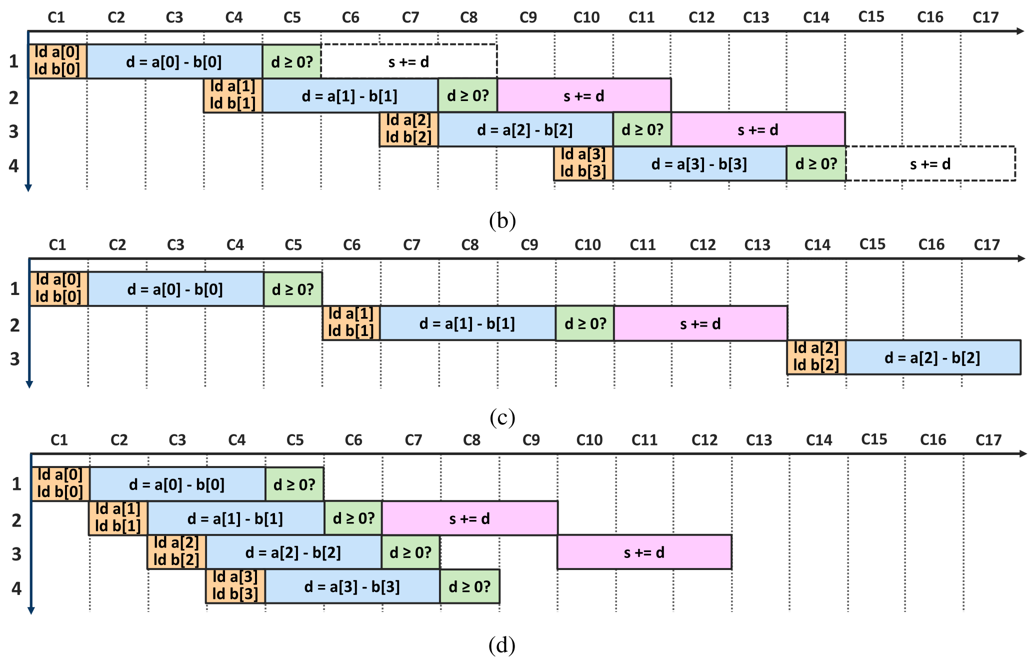


Fig. 1. Limitations of static scheduling. (a) Shows a code example where dependences cannot be determined at compile time. (b) and (c) Contrast two static schedules, possible with standard HLS tools (realized as a pipeline and a sequential state machine, respectively), with a dynamic schedule (d), achievable with our approach. The dynamic schedule achieves the best possible parallelism, which is reduced only in the presence of actual loop-carried dependences.

is shown in Fig. 1(b): in this example, the condition is true only for the second and third iterations but "space" is reserved in the schedule as if the condition was true everywhere. An alternative could be to avoid pipelining the loop and creating a sequential finite-state machine. The result could be the schedule in Fig. 1(c), where indeed cycles are spent for the addition only when needed; however, the decision of not pipelining the loop has removed one of the foremost potentials for parallelism (in this case, the memory reads, the subtraction, and the comparison are perfectly independent across iterations and could be pipelined). Obviously, a good schedule is the one in Fig. 1(d): the operations of different iterations are overlapped as much as possible and the parallelism is reduced only when the dependency is actually there (that is, when the addition is executed). Such behavior is beyond what a statically scheduled HLS tool can achieve.

This example is representative of one case where generating a schedule at synthesis time has a negative impact on performance. Another well-known situation is the presence of dependences through memory: a write in a previous iteration may address the same memory location as the read in a successive one and thus, creates a dependency imposing serialization; yet, if these two accesses address different locations, they can be executed out of order. When an HLS tool is not able to guarantee independence between two memory accesses, it must assume the worst-case scenario and thus, limit the exploitable parallelism—exactly as above but for a different reason. In recent years, many authors have been exploring workarounds to some cases of potential dependences through memory—we will discuss them in Section VIII—but dynamically scheduled circuits represent the most general solution to the problem.

A. *Dataflow Circuits*

The key to avoiding the limitations of static scheduling is to refrain from triggering the operators through a centralized preplanned controller 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. The rest of this section looks informally at one dynamically scheduled circuit paradigm to give the reader a flavor of what we want to achieve.

Fig. 2 shows a simplified version of a *dataflow circuit* [15] implementing the loop of Fig. 1(a). Besides normal datapath units, this circuit uses a few control units labeled buffer, merge, select, fork, and branch. All directed edges in the figure represent data signals accompanied by handshake control signals, which indicate the availability of a new piece of data from the source unit and the readiness of the target unit to accept it, respectively. The loop to the left of the figure shows the part of the circuit which updates the iterator `i`: at the beginning, the constant 0 is sent from the entry point. The merge node takes this value and passes it further. The buffer node is the register that holds `i` and distributes it on the next clock cycle to three consumers through the fork node; all successors must consume the value before the fork accepts a new input value. The left branch compares the incremented `i` with the loop bound; if the bound is not reached, the new value of `i` is sent back to the register by the branch node through the merge. Meanwhile, the other outputs of the first fork use `i` to access `a[]` and `b[]` and to compute the subtraction, which is propagated to the rest of the circuit.

The key to a good execution of this loop is that, ideally, a new value of `i` should be used to start computing `a[i] - b[i]` on every cycle. This is the case in this circuit, contrary to a conservative statically scheduled one: the cycle on the left-hand side of Fig. 2 is completely combinational excluding the buffer and thus, a new value for `i` can be computed on every cycle. It is the right part of the circuit which can delay this: when the `if` is not taken, the result of the addition is dumped by the select node as soon as it arrives through the merge and the old value of `s` becomes immediately the new

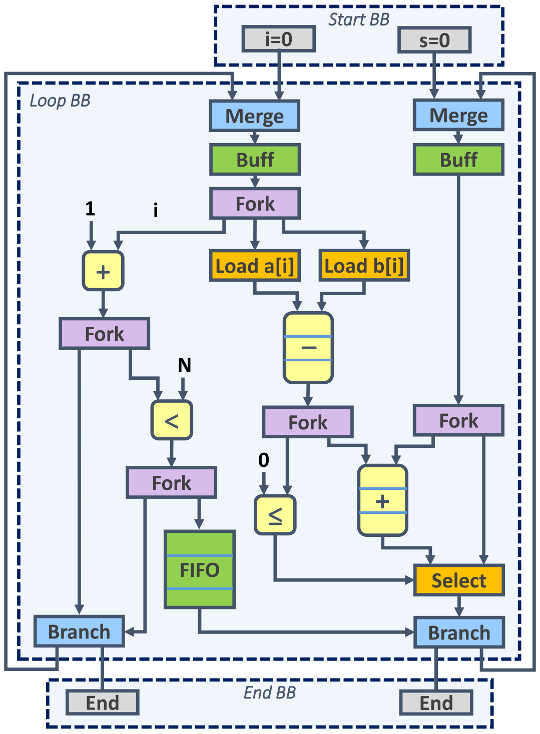


Fig. 2. Dynamically scheduled, dataflow circuit implementing the code from Fig. 1(a) and achieving the schedule of Fig. 1(d).

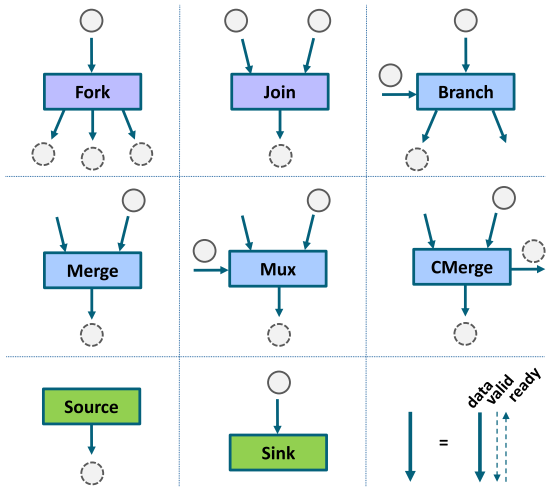


Fig. 3. Dataflow units. All data channels are paired with bidirectional control signals, which indicate the validity of data and the readiness of the successor unit to accept it.

value that is sent back to the adder on the following cycle; if, on the other hand, the result is needed, the select will wait for the sum to complete and the adder will be stalled next cycle waiting for its right operand. Ultimately, a new subtraction will not be computed and the memory accesses will not be performed due to backpressure from the adder; the top fork will not allow a new `i` to proceed on the right branch. This slows down the initiation of the loop and is exactly what the dynamic schedule in Fig. 1(d) shows.

In the rest of this article, we describe our complete methodology to automatically generate dynamically scheduled circuits, such as the one in Fig. 2, from C/C++ programs. Although the potentials of gain in terms of clock cycles saved in situations such as the one in this example are at least qualitatively clear, dynamic scheduling also costs resources and time (i.e., the area and delay of the dataflow units in Fig. 2). To evaluate these area-performance tradeoffs, we compare our circuits with those obtained using a state-of-the-art HLS tool and we show that dynamic scheduling can reap significant performance benefits in appropriate situations.

III. Synthesizing Dataflow Circuits

In this section, we first outline the dataflow units which we use in our work. We then describe the process we use to convert an arbitrary piece of code into a dataflow circuit.

A. Dataflow Units

Latency-insensitive protocols [8], [15] are a natural method to create synchronous dataflow circuits, capable of making decisions at runtime. Such circuits are built out of *units* that implement latency insensitivity by communicating with their predecessors and successors through *channels* composed of data lines and paired with handshake control signals: a *valid* signal indicates that a unit is sending a valid piece of data to its successor(s), whereas the *ready* signal informs the predecessor(s) that a unit can accept a new piece of data. A *token* [41] of data is propagated from unit to unit through a channel as soon as memory and control dependences allow it—otherwise, it is stalled by the handshake mechanism.

Fig. 3 outlines the dataflow units we use. Their gate-level descriptions can be found in prior literature [15], [29].

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.
2) 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).
3) A *join* acts as a synchronizer—its output is triggered only after all of its inputs become available.
4) 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.
5) A *merge* is a nondeterministic unit that propagates a token received on any of its input to its single output.
6) A *mux* is a deterministic version of the merge; it propagates to its single output the input token selected by a control input.
7) A *control merge (cmerge)* is a merge which apart from the data output, has an output that indicates which of the inputs was taken by the merge.
8) A *source* can always issue a valid token to its single successor (e.g., a constant).
9) A *sink* is always ready to consume tokens from its single predecessor; the token is simply discarded in the sink.

In addition, we use 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 require *buffers* that serve as registers in standard synchronous designs—we discuss

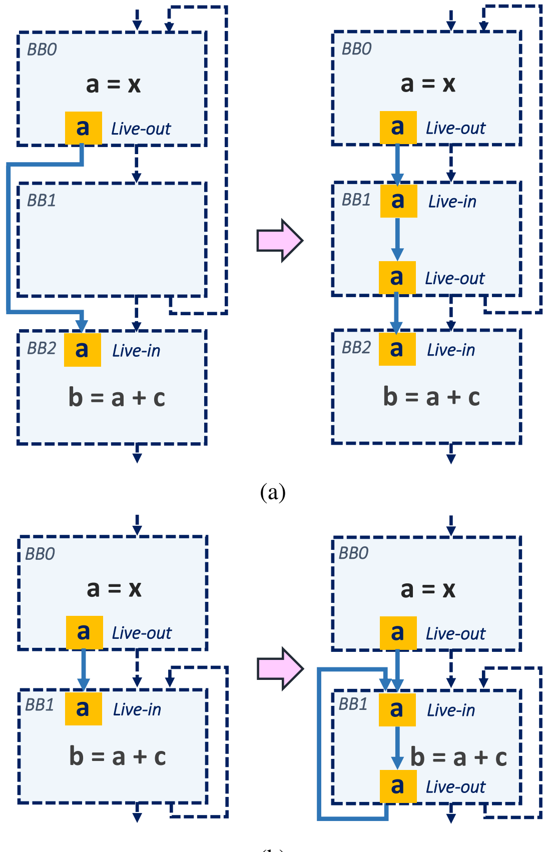


Fig. 4. Implementing control flow. The left circuits (left of Fig. 4(a) and (b)) show cases where a direct conversion of a data and control flow graph into a dataflow circuit would fail. Coupling data and control to ensure correct token transfers between BBs is shown on the right of Fig. 4(a) and (b): data are propagated exclusively from each BB to its immediate successor BBs, using merge and branch units for each BB live-in and live-out, respectively.

their properties and placement in Section IV. Finally, our circuits interface to memory using read and write ports; we will address the memory interface in Section V.

B. Implementing Control Flow

The starting point for our circuit generation is a standard compiler intermediate representation in static single assignment (SSA) form, where every variable is defined only once and *phi* nodes are used to define a variable from multiple definitions along multiple control paths [50]. The control-flow graph (CFG) of a program is organized into basic blocks (BBs), i.e., straight pieces of code separated by control flow decisions. Each BB contains a dataflow graph (DFG) of program instructions; it receives *live-in* variables from the predecessor BBs and produces *live-out* variables for the successor BBs. Transforming the DFG of each BB into a corresponding interconnect of dataflow units is relatively straightforward—we will describe this process in Section III-D, but there are a couple of problems when implementing control flow and sending values from one BB to another due to the fundamental difference between software programs and dataflow circuits.

Fig. 4 shows two examples.

1) In Fig. 4(a), the variable a is defined in BB0 and used in BB2. A typical representation in a compiler (left-hand side of the figure) propagates the desired information directly from the source to the destination block (i.e., a live-in of a BB comes from a BB which is not its immediate predecessor). This flow does not pose problems in software, as successive values of a would be stored in a register of a processor or in memory and the last value used when BB2 is reached.
2) In the example in Fig. 4(b), BB1 is the only BB in the body of a loop and uses a value a produced in BB0. The value of a does not change during the execution of BB1 and is used at every execution of BB1. Again, the representation on the left would cause no problem in a processor—the value would be stored in a register or memory and read as many times as needed. Similarly, in both cases, a standard HLS tool would devise a schedule that ensures that each value is kept in a register as long as it is needed; values are read from and written into the register in appropriate (and predetermined) clock cycles.

Directly implementing such connections in a dataflow circuit would result in incorrect behavior because every generated value is associated with a token; the number of tokens must exactly match the number of distinct uses. The cases in the left-hand side of Fig. 4(a) and (b) violate this principle if implemented literally.

1) In the first case, if the control flow was {BB0-BB1-BB0-BB1-BB2}, two new values (with the respective tokens) for a would have been generated and sent to BB2; yet, BB2 can take only a single token and requires only the most recent value. The execution would be incorrect or the circuit would not terminate because the tokens not absorbed by BB2 would create backpressure to BB0 and stop it indefinitely.
2) In the second case, BB1 would not be able to execute repeatedly due to a starving input. Assuming the control flow is {BB0-BB1-BB1}, the first execution of BB1 will consume the single data token for a and any further execution of BB1 would stall indefinitely waiting for a token.

The solution to both problems is to strictly couple data propagation with control flow, as shown on the right-hand side of Fig. 4(a) and (b). The following properties must hold.

1) Every BB must provide a live-out for every live-in of all of its immediate successor BBs and exclusively to them.
2) Every BB must receive all of its live-ins from its immediate predecessor BBs and exclusively from them.

We implement these rules as shown in Algorithm 1.

1) We employ a standard liveness analysis algorithm [50] to determine the live-ins and live-outs of each BB.
2) For every BB live-in and live-out, we place a merge and a branch unit in the BB, respectively.
3) We connect all operations within a BB that use a live-in to the appropriate merge of the same BB (i.e., the merge will inject tokens into the BB to trigger the execution of its operations).
4) We connect the outputs of all branches to the inputs of the corresponding merges in the immediate successor BBs. In Fig. 4(a), this strategy results in merges for a in BB1 and BB2 and branches for a in BB0 and BB1. In Fig. 4(b), BB0 has a branch for a and BB1 has a merge and a branch.

Algorithm 1: Implementing Control Flow

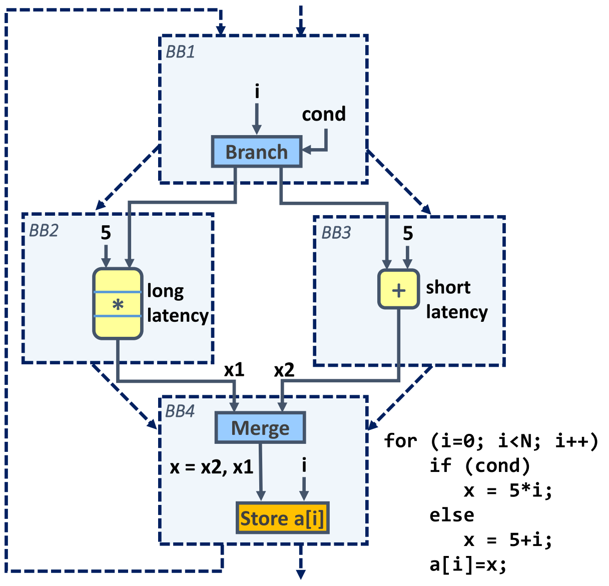
```
// Input: CFG (control-flow graph)
// Input: DFG (SSA-based dataflow graph)
// Output: DFG (dataflow graph with coupled
// data propagation and control flow)

// Determine live-ins and live-outs of each BB
$liveIns, liveOuts = LivenessAnalysis\ (CFG)$

// Place merge for every live-in in every BB
foreach $bb \in CFG$ do
    foreach $li \in liveIns\ (bb)$ do
        $mg = CreateMerge(bb, li, DFG)$
        // Connect all operations within the BB
        // that use the live-in to the
        corresponding merge
        foreach $op \in operations\ (bb)$ do
            if $li \in predecessors\ (op)$ then
                $Connect\ (op, mg)$

// Place branch for every live-out in every BB
foreach $bb \in CFG$ do
    foreach $lo \in liveOuts\ (bb)$ do
        $br = CreateBranch\ (bb, lo, DFG)$
        // Connect branch to corresponding merges
        // in successor BBs
        foreach $bb_{succ} \in successors\ (bb)$ do
            $mg = FindMerge\ (lo, bb_{succ})$
            $Connect\ (br, mg)$
```

This strategy guarantees that every piece of data is sent correctly from BB to BB, following the control flow of the program. Each BB contains as many merge units as it has incoming variables and as many branch units as it has outgoing variables. Some merges correspond to SSA *phi* nodes—they propagate into the BB a value chosen from one of the distinct predecessor values (based on the control flow), whereas other merges propagate a single value (coming from different control flow directions) to honor the rules above. This is the case, for instance, for the merge for variable a in BB1 of Fig. 4(b). All branches of a BB share the same condition and send tokens to the same successor BB based on a control flow decision.

Our strategy allows tokens to independently proceed from one BB to the succeeding BB (i.e., there is no synchronization of tokens at the BB output) and ensures correctness by propagating all tokens strictly following the control flow. However, in particular cases, this approach may be overly conservative: a throughput-critical token (e.g., a token carrying the loop iterator) may be unnecessarily prevented from quickly propagating through the BBs due to a long-latency BB condition. Although outside the scope of this article, systematically determining when data can bypass a certain BB (i.e., when a token propagation is independent of a particular BB condition) would further simplify the dataflow network and may, in particular applications, improve the achievable throughput.

C. Ensuring Determinism

Although different operations in a dataflow circuit may execute out-of-order, tokens arrive to each individual operator *strictly in order*. Yet, there is one particular case in which this property may not hold and which we discuss in this section.



```
for (i=0; i<N; i++)
    if (cond)
        x = 5*i;
    else
        x = 5+i;
    a[i]=x;
```

Fig. 5. Nondeterministic behavior at SSA *phi* nodes. The token entering BB4 is produced either by BB2 or BB3; since these values are produced independently, the merge in BB4 may receive its inputs out-of-order.

The execution of our dataflow circuits is triggered by injecting a single token for each input (i.e., program argument) into the start BB. The tokens propagate through the BBs, following the control flow of the program—the BBs are triggered in exactly the same order as the software execution of the unmodified original program. When a single value propagates through the BBs, a token will always enter each BB from its single active predecessor—once the token enters through a merge, no other source can reinject a token into the merge until the merge itself produces a token; hence, there is nothing that can interfere with the token ordering at the BB input. Tokens will never reorder inside a BB as it contains only straight and unconditional dataflow computation.

However, the situation is different in BB entry points where a value is redefined (i.e., when a token enters a BB through a merge which corresponds to SSA *phi* node)—as each input represents a distinct and, potentially, uncorrelated value, the input tokens may arrive in an order different than specified in the original program. An example of such a case is illustrated in Fig. 5, which shows the CFG and a simplified datapath of the code in the figure. Assuming that the control flow is {BB1-BB2-BB4-BB1-BB3-BB4} (determined by the condition cond in BB1), the iterator from BB1 will first be sent to BB2 to compute the value of x1. This value takes multiple cycles to compute, but the iterator can quickly propagate through BB4 and BB1 (the iterator path is omitted from the figure; as described in the previous section, it follows the control flow of the program). It will then enter BB3, which will trigger the short computation producing x2—this value may arrive to the merge in BB4 before the value of x1; the merge would send the values to the store out of order, which would then produce incorrect results. In standard HLS, this problem is addressed through static scheduling, which enforces in-order execution and dictates *phi* input ordering.

To ensure that tokens never enter a BB out of order, we replace every merge that corresponds to an SSA *phi* node with a mux unit described in Fig. 3. We generate an in-order control

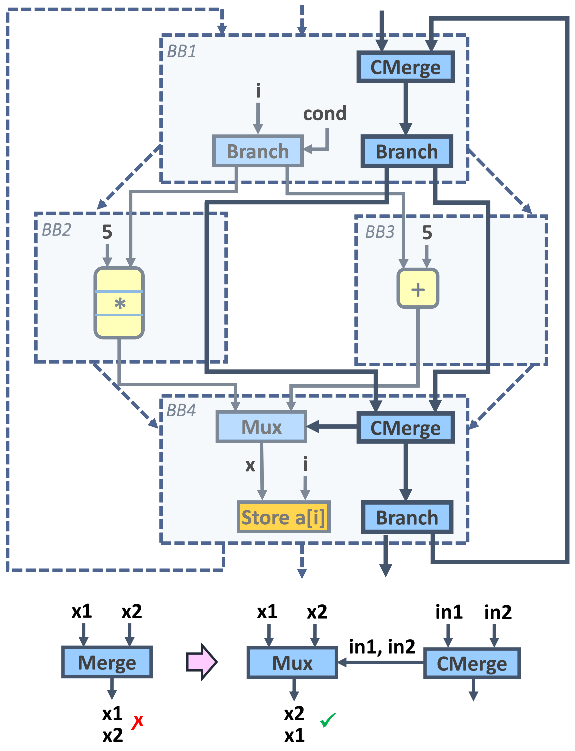


Fig. 6. Ensuring determinism. We extend the circuit from Fig. 5 with a specialized in-order control network that follows the control flow of the program—the cmerges of this network communicate with the muxes of the same BB to indicate the correct input ordering.

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. 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. The extended circuit from Fig. 5 is shown in Fig. 6: in the previously discussed control flow sequence, the cmerge in BB4 would first receive a value on input `in1`, coming from BB2, and then on input `in2`, coming from BB3—it would indicate this ordering to the mux, which would then not accept the value of `x2` before it has received the value of `x1`. This way of building dataflow circuits implies the following properties.

1) *Determinism:* The strict ordering of BBs reflected in the in-order control path guarantees that the execution is race free.
2) *One Token Per Loop:* On a cyclic path, there can be only a single token at a time (a token enters a BB on a cycle through a merge; as this BB determines the next control flow decision, it is the only block that can send the token back into the merge).
3) *Strict Token Ordering on a Path:* If there are multiple tokens on an acyclic path, they could only be injected into it by repeatedly forking at every passage the single token of a cycle and the cyclic propagation of this token is determined by the in-order control flow decision.

D. 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

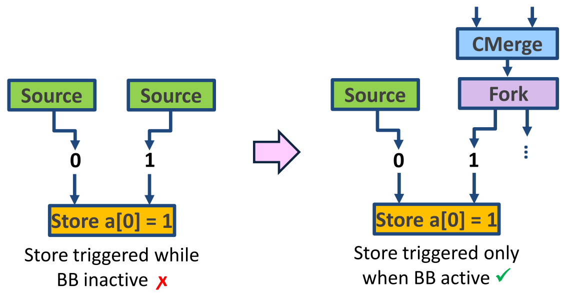

Source
Source
0
1
Store a[0] = 1
Store triggered while BB inactive ✗
CMerge
Source
Fork
0
1
Store a[0] = 1
Store triggered only when BB active ✓

Fig. 7. Triggering constants. Setting constants as always valid (e.g., using a source) may incorrectly trigger operations even when their execution is not determined by the control flow (in this example, the store would constantly store data to memory). In such cases, at least one constant should be connected to the in-order control network, thus ensuring that the constant is triggered only when its BB is active.

and outputs accompanied by handshake signals). When a unit has more than one direct successor, we place a fork to replicate the token into an individual token for each of the successors (i.e., for each point-to-point data transfer). Unused unit outputs (e.g., branch outputs without successors) connect to sinks that discard the unused tokens.

Some units (e.g., constants) do not have any inputs; we must ensure that they are appropriately triggered and executed. Keeping units without inputs always active (e.g., by setting a source as their input) may result in incorrect behavior, as they could trigger operations that are not supposed to execute. An example is shown in Fig. 7: a store with a constant address and data would constantly send data to memory, regardless of the number of store executions specified by the program. Another case is that of a branch with a constant data input and a constant condition, which would constantly trigger a merge of some successor BB, even if this is not decided by the control flow. For this purpose, we exploit the in-order control network described in Section III-C and used to ensure determinism—we fork the token from this network and use it to trigger operations with no inputs only and as many times as their BB becomes active, as shown on the right-hand side of Fig. 7. Whenever a constant is an input to a unit that is triggered only when the BB is active (i.e., at least one of the unit predecessors is a live-in of the BB), the connection of the constant to the control path can be omitted and it can be triggered by a source instead, which reduces the complexity of the dataflow network. This is the case for the constants in BB2 and BB3 in Fig. 5—the computational units will receive a data input and trigger the computation only if the corresponding BB becomes active, so both constants can be triggered with a source. Similarly, only one of the constant inputs to the store in Fig. 7 requires a connection to the control network.

IV. Buffers in Dataflow Circuits

The circuits produced by the compilation strategy described in the previous section do not contain any buffers (i.e., registers). In this section, we discuss buffer properties and their importance in obtaining high-performance dataflow circuits.

A. Buffer Properties

Dataflow circuits require *buffers* that serve as registers in standard synchronous designs. Buffers store either *tokens* (i.e., valid data) or *bubbles* (i.e., invalid data). A buffer can

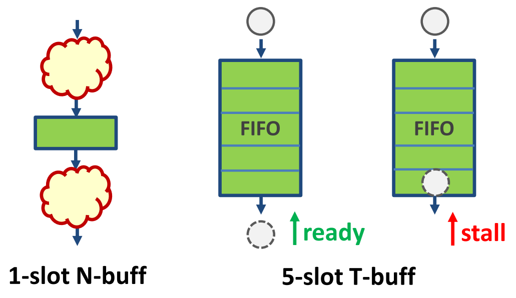


Fig. 8. Buffer properties. The figure contrasts a 1-slot nontransparent buffer, which breaks the combinational path and can store a single token, with a 5-slot transparent buffer, which can send data combinationally from input to output or store up to five tokens if the successor is not ready to take them.

hold a token or a bubble—each time a token moves forward, a bubble moves in the opposite direction, similar to electrons and holes in semiconductors [26]. Every cycle of our circuit will always contain *at most one token* (see Section III-C), whereas bubbles can be freely allocated by adding buffers. The buffers are characterized with two properties.

1) *Transparency* 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 (therefore, potentially damaging throughput), whereas a transparent buffer is implemented as a pass-through element and does not increase cycle count (but may deteriorate the combinational delay due to the bypass multiplexer at its output)
2) *Capacity* (i.e., number of slots) is used to regulate throughput; these properties are illustrated in Fig. 8: a single-slot nontransparent buffer is equivalent to a register in a standard synchronous circuit; a common FIFO of size N with a combinational path between input and output is here, an N-slot transparent buffer.

B. Buffers and Circuit Functionality

Dataflow systems use distributed handshake signals to control the flow of data in the datapath. These signals implicitly take care of stalling early data items when they need to synchronize with later items [23]. Although buffers shift the values in time, their presence or absence does not affect the functional correctness of the system, as any consumer of multiple values synchronizes the corresponding valid tokens. Hence, contrary to registers in traditional synchronous designs, buffers can be placed on *any* channel of the dataflow circuit—due to its latency insensitivity, this insertion will not compromise the functionality [5], [33], but may impact timing and throughput.

C. Buffers and Avoiding Deadlock

The following conditions are necessary to ensure deadlock-free execution of dataflow systems.

1) Each combinational cycle must be broken with at least one nontransparent buffer; this requirement is analogous to that in standard synchronous circuits, where each combinational cycle needs to be broken using a register.
2) Each cycle in must contain at least one token and one bubble [16]; this requirement ensures that a token and a bubble can always exchange places and tokens can propagate through the cycle. As our circuit generation strategy guarantees that each cycle will have exactly one token, our combinational cycles will require at least two buffer slots to accommodate for the token and (at least) one bubble.

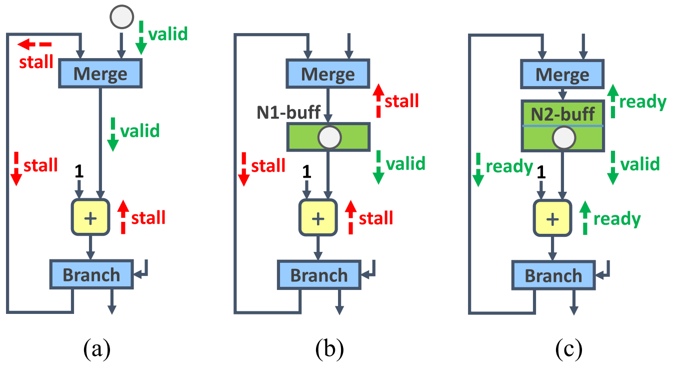


Fig. 9. Adding buffers. A combinational cycle without buffers or with a single buffer slot will cause deadlock, as the token will not be able to propagate through the cycle. At least two buffer slots are necessary to ensure deadlock-free execution. (a) No buffer on cycle: deadlock. (b) 1-slot buffer on cycle: deadlock. (c) 2-slot buffer on cycle: no deadlock.

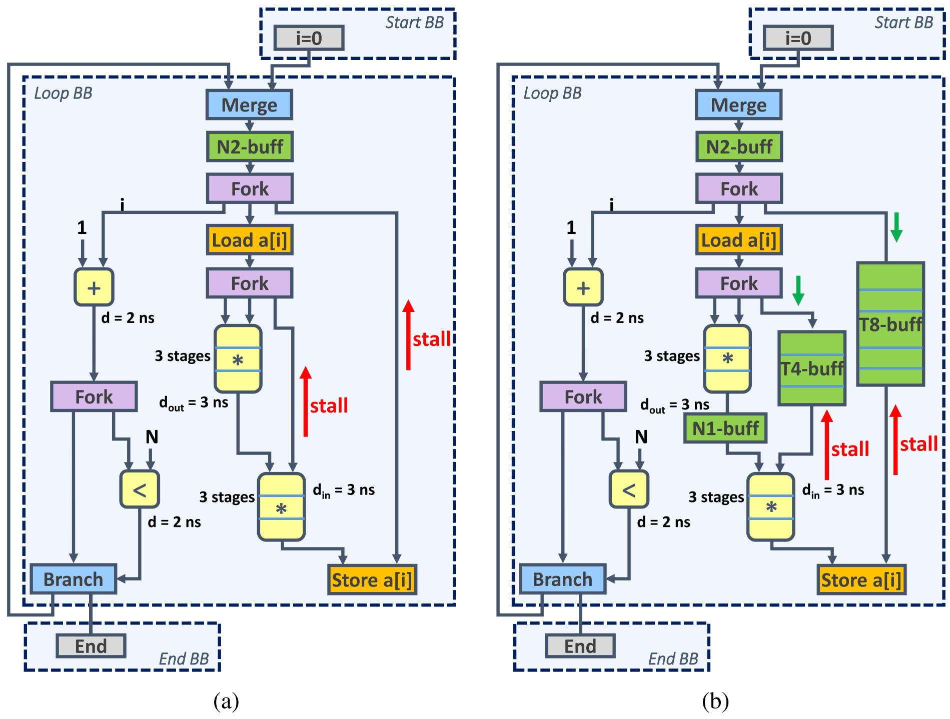
Fig. 9 contrasts a combinational cycle of a dataflow circuit without a buffer, with a single-slot nontransparent buffer (satisfying the first requirement above), and with a two-slot nontransparent buffer (satisfying both of the requirements above). In the first case, a token at the input of the merge cannot propagate through the cycle due to the combinational relationship of the valid and ready handshake signals on the cycle. Adding a buffer breaks the combinational path and enables the token to enter the cycle, but there is no empty buffer slot (i.e., bubble) for the token to loop back through the merge. A 2-slot buffer (N2-buff) ensures deadlock-free execution as a token and a bubble can always exchange places.

D. Buffers and Performance

The circuit in Fig. 10(a) satisfies the correctness properties described in the previous sections, but it fails to address two important performance aspects.

1) *Critical Path:* The buffers are placed without any consideration for the combinational delays of the nodes (all nonzero delays are indicated in the figure) and therefore, do not control the critical path in any way.
2) *Throughput:* Some paths may take a longer time to process data and prevent the faster paths from consuming tokens at a higher rate. This effect may restrict loop pipelining—even if the need for another iteration can be decided very fast, new tokens may not be able to trigger the following loop operations because tokens from the previous iterations are stalled in the loop units.

In Fig. 10(a), the token carrying the array value a is forked into two pipelined multipliers, but the lower multiplier cannot accept the token until the upper multiplier is done computing (i.e., after three clock cycles). Similarly, the store can accept the iterator from the fork only after the two chained 3-stage multipliers produce a result. Because of backpressure in these paths, the iterator cannot be issued through the loop at a high rate, which lowers the loop initiation interval (II).



```
int i = 0;
for (i = 0; i < N; i++)
   a[i]=a[i]*a[i]*a[i];
```

(c)

Fig. 10. Circuit in (a), implementing the code from (c), is functionally correct and deadlock free, but it is not optimized for performance. The optimized circuit in (b) has buffers placed strategically: nontransparent buffers (i.e., N1- and N2-buff) control the critical path and larger capacity transparent buffers (i.e., T4- and T8-buff) in the slow paths mitigate backpressure to maximize throughput.

Algorithm 2: Performance Optimization

```
// Input: CFG (control-flow graph)
// Input: DFG (dataflow graph)
// Output: buffers (list of dataflow channels
// characterized with buffer capacity and
// transparency)

// 1. Identify choice-free subgraphs of the
dataflow graph
$profile = ProfileApplication\ (CFG)$

// ILP for iterative cycle extraction
$cycles = ExtractCycles\ (CFG, profile)$

// Find dataflow subgraph of each cycle
foreach $c \in cycles$ do
    $subgraph\ (c) = FindDataflowSubgraph\ (c, DFG)$

// 2. Optimize performance
foreach $c \in cycles$ do
    // Choice-free subgraph throughput
    $th.add(ThroughputConstraints(subgraph(c)))$
foreach $u \in DFG$ do
    // CP of entire dataflow graph
    $cp.add(PeriodConstraints(u, e))$

// MILP to max. throughput under CP constraint
$buffers = MILP\ (th, cp, CP_{target})$
```

Fig. 10(b) shows a possible circuit configuration with optimal throughput and the critical path constrained to 4 ns. The additional nontransparent buffer lowers the critical path by breaking the combinational delay of 6 ns between the multipliers. Inserting transparent buffers of larger capacity increases effective parallelism, as accumulating data in these buffers allows to trigger the faster paths at a higher rate and, in this case, achieves the ideal loop II of 1.

We developed an optimization approach [36] which allows for resource-optimal buffer placement and sizing, with the purpose of maximizing throughput of the performance-critical loops at the desired clock frequency. Our optimization strategy consists out of two main steps, as illustrated in Algorithm 2: 1) we profile the application and employ an integer linear programming model to identify performance-relevant choice-free subgraphs of the DFG and 2) we employ a mixed-integer linear programming model based on Petri net theory [41] which strategically places and sizes buffers to optimize the throughput of each choice-free subgraph, while ensuring that the entire DFG meets the target clock period (CP) constraint. The resulting circuits correspond to the one in Fig. 10(b): larger transparent buffers regulate throughput and smaller nontransparent buffers control the critical path.

Analogous to static HLS, where the decision on how many units to employ is made together with operation scheduling, our performance optimization model can also decide which operations can share a functional unit: the obtained throughput directly determines the rate of token propagation through each unit and identifies underutilized units that may be shared without a performance degradation. Appropriately multiplexing the incoming tokens at the shared unit inputs avoids unit starvation and ensures the absence of deadlock [30].

V. Connecting to Memory

Thanks to their latency insensitivity and in contrast to statically scheduled designs, dataflow circuits can easily connect to any memory interface and hierarchy without requiring memory-specific modifications; the handshake logic will

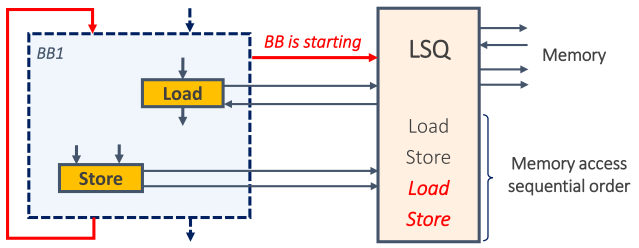


Fig. 11. LSQ required for correct out-of-order memory accesses. In addition to load and store ports, the LSQ requires a specialized signal indicating the start of each BB in the program-determined order.

ensure that the execution naturally adapts to any memory latency and variability (e.g., variable latency of a DRAM memory controller or a cache). Connecting every load or store operation to a read and write port, respectively, seems a natural decision, but the result may be incorrect. Access requests will arrive to the memory interface in an arbitrary order. In general, this is the dynamic out-of-order feature that we desire—in contrast, statically scheduled circuits must conservatively serialize possibly dependent accesses, resulting in suboptimal performance. Yet, this out-of-order execution may lead to the violation of memory dependences: for instance, if a write happens at the same address as some successive read, and if the read token arrives in the dataflow circuit before the write token, the result of the read will be incorrect.

The solution is to use an LSQ similar to those present in dynamically scheduled processors. Yet, we have shown that building an LSQ for dataflow circuits has one fundamental difference [32]: the LSQ 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. The details are beyond the scope of this article; it suffices to say that the key condition for the LSQ to execute correctly is to receive tokens that follow the actual order of execution of the BBs of the circuit. This ordering enables the LSQ to determine and resolve dependences as memory access arguments from different BBs arrive out-of-order.

Consider a program containing a single BB with a potentially dependent read and write access, as shown in Fig. 11. Apart from the read and write port communicating with the LSQ, an additional signal indicates to the LSQ the start of the particular execution of the BB. Each BB with accesses targeting the same memory will employ such a signal—the ordering of these tokens enables the LSQ to appropriately handle out-of-order memory accesses. In the example in the figure, the second read request may arrive before the previous store has completed; the LSQ will appropriately stall its execution if it is dependent on the store or allow the accesses to execute out-of-order otherwise.

Apart from the allocation strategy, which is unique for our dataflow approach, the remainder of our LSQ implementation qualitatively corresponds to a standard processor LSQ, as Fig. 12 suggests. Our challenge here is to guarantee that the signals coming from the BBs to the LSQ are produced *in order* by a circuit which we have, otherwise, designed to be as aggressively out-of-order as we could. To this end, we exploit the in-order control path that we introduced in Section III-C. The tokens in this path trigger the allocation of BBs as soon

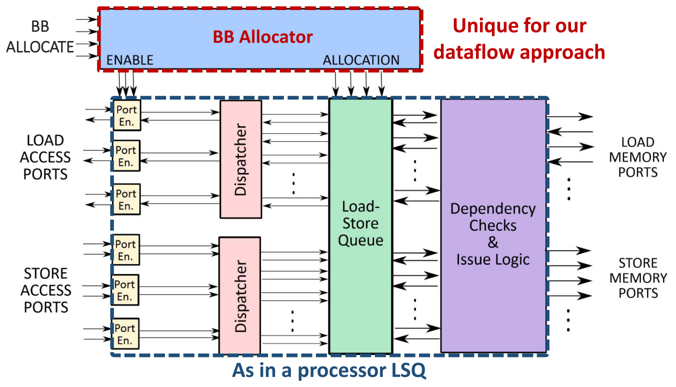


Fig. 12. LSQ structure. Apart from the allocation strategy, our LSQ is essentially identical to that of a common processor.

as the control flows there (i.e., as soon as a decision is made to enter a particular BB). However, applying the standard dataflow circuit design strategy described in the previous sections might result in the incorrect order of token arrival to the LSQ. Fig. 13 shows two example situations leading to a potentially wrong execution: 1) if the token is forked to the LSQ using the typical eager fork, one of the fork outputs might send a token to the next BB before the LSQ has accepted a token from its predecessor, as shown in Fig. 13(a) and 2) although placing buffers in dataflow circuits has no impact on correctness (as discussed in Section IV), a buffer on the fork output connected to the LSQ might compromise the order of token arrival to the queue—if the token remains stored in the buffer, the successor BB could send a new token before the prior allocation has been completed, as shown in Fig. 13(b).

The correct way to connect the LSQ to the dataflow circuit is shown in Fig. 13(c): 1) the forks used to send the tokens to the LSQ are lazy forks (lforks)—if one of the fork outputs is stalled, the other one will stall as well and 2) no sequential elements (i.e., buffers) are allowed on the fork outputs connected to the LSQ. This ensures that a token can be passed to the successor BB only when the allocation of its predecessor BB has been completed—if an allocation is deferred (e.g., due to limited space in the LSQ), the token stalls and no further allocation requests reach the LSQ.

To connect our datapaths to memory, we leverage compiler analysis to simplify our memory interface. Whenever the compiler can disambiguate memory accesses, groups of accesses that cannot mutually conflict use separate LSQs, while accesses that cannot have dependences with any other accesses are connected to simple memory interfaces [31].

VI. Complete Flow

In the previous sections, we have shown how an arbitrary program described in a high-level language can be transformed into a dynamically scheduled, dataflow circuit, which executes operations out-of-order, naturally implements pipelining, and efficiently handles potential memory dependences.

The presented flow is implemented in Dynamatic, our open-source HLS compiler [35]. The basic flow of Dynamatic is shown in Fig. 14. It 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,

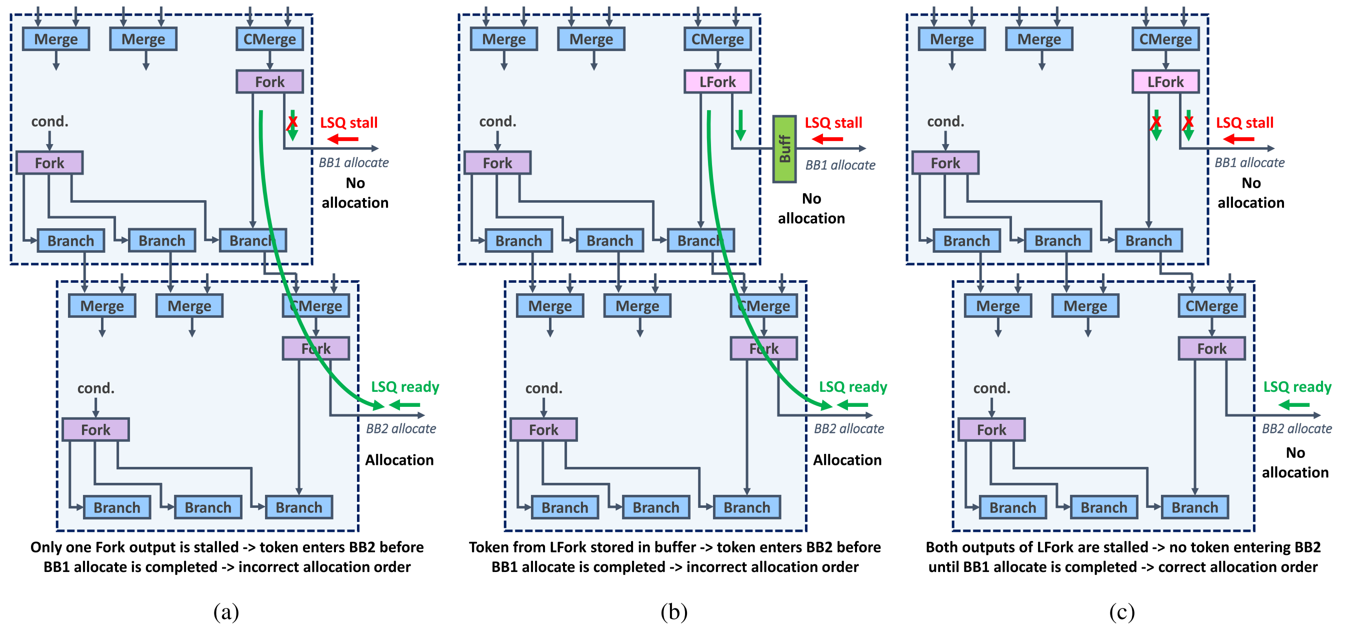

Merge
CMerge
Fork
LFork
cond.
Branch
LSQ stall
BB1 allocate
No allocation
LSQ ready
BB2 allocate
Allocation
Buff
Only one Fork output is stalled -> token enters BB2 before BB1 allocate is completed -> incorrect allocation order
Token from LFork stored in buffer -> token enters BB2 before BB1 allocate is completed -> incorrect allocation order
Both outputs of LFork are stalled -> no token entering BB2 until BB1 allocate is completed -> correct allocation order
(a)
(b)
(c)

Fig. 13. Connecting the dataflow circuit to the memory interface. (a) and (b) give examples of incorrect connections. In (a), the eager fork may send an allocation to BB2 before the allocation of BB1 completes. In (b), the allocation order may be reversed due to the storage element on the control line between the circuit and the load-store queue (LSQ). (c) shows the correct way to connect the LSQ—an allocation cannot occur unless all predecessor allocations have been completed.

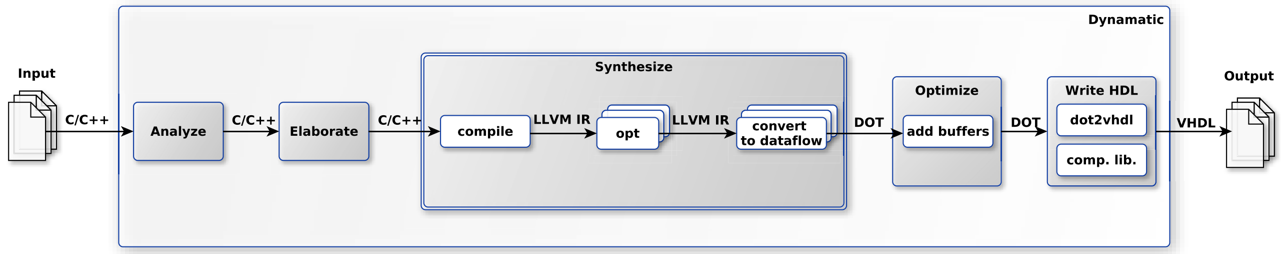


Fig. 14. Dynamic HLS compiler: software-to-hardware flow.

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 framework [39]: the *clang* frontend parses the C/C++ program and produces an SSA 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 III-A, following the transformations described in Sections III-B–III-D to produce a functionally correct dataflow circuit; additional passes perform analysis and optimizations (e.g., memory access analysis to create the memory interfaces from Section V). The output is a DFG in the form of a DOT netlist. This netlist is then provided to the optimizer—it uses a MILP solver [22] to find the optimal buffer placement and sizes for a user-defined CP constraint, as indicated in Section IV. This step produces an optimized DOT netlist. Finally, the DOT describing the dataflow circuit is converted into a VHDL netlist of dataflow units. This netlist, in conjunction with a predefined library of dataflow units, can be synthesized and implemented on an FPGA. The generated circuit is packaged as an IP and integrated as a hardware accelerator into a heterogeneous FPGA design, which includes a soft or hard CPU core and communicates through an AXI interface. Dynamatic currently targets Xilinx and Intel FPGAs; our component library is easily extensible to target other platforms. Thanks to their flexible handshake logic, dataflow circuits can easily adapt their execution to units with different timing properties; yet, their timing may impact circuit performance and, therefore, must be considered by our performance optimization model. Hence, we characterize each unit in our unit library with its latency, II, and critical path (determined based on static timing analysis of the unit); based on the target FPGA, we choose the units to employ in the circuit and include the corresponding timing values in our performance optimizer.

VII. Evaluation

In this section, we compare the dynamically scheduled circuits produced by Dynamatic with a commercial HLS tool. We give an overview of our methodology and benchmarks before presenting our results. Our complete HLS tool and our benchmarks are publicly available at dynamatic.epfl.ch.

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

Benchmark	**II$_{avg}$**		**CP (ns)**		**Exec. time (us)**		**Slices**		**LUTs**		**FFs**		**DSPs**	
	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN
Histogram	13.0	2.3	3.5	4.9	45.5	11.1	129	220 + 1073	254	4294	510	2033	2	2
Matrix power	13.0	2.7	3.4	4.9	16.8	4.9	200	295 + 1020	340	4463	735	2055	5	5
Matching	16.0	9.0	3.3	5.1	21.1	18.4	141	296 + 104	336	1087	446	1084	0	0
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	3.5	2.9	3.5	47	221	83	575	176	628	3	3
Matvec	1.0	1.0	3.2	4.0	2.9	3.6	63	309	119	903	221	699	3	3

A. Methodology

To demonstrate the benefits of using dynamic scheduling in HLS, we compare our circuits with designs generated by Vivado HLS [54], a state-of-the-art commercial HLS tool. In all Vivado designs, we apply the pipelining optimization directive. Although supported by our approach, we do not employ unrolling as this code restructuring optimization is orthogonal to the scheduling paradigm and would affect similarly Vivado's results and ours. To provide a fair comparison, we employ the same arithmetic units (with custom wrappers employing handshake signals) and RAM blocks used by Vivado in our designs. When our compiler cannot disambiguate memory accesses, we employ the LSQ in our designs and connect it to the RAM interface; otherwise, we connect the dataflow read/write ports to the RAM through a simple memory arbiter.

We functionally verify the designs in ModelSim [40]. We obtain the average loop II from the simulation and the CP from the postrouting 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).

B. Benchmarks

The designs that we discuss in this section 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 kernels where static scheduling is fully successful, to show that dynamically scheduling achieves virtually the same result with acceptable overheads.

1) *Histogram* reads an array of features and increases the value of the corresponding histogram bins. The memory access pattern cannot be determined at compile time—the loop may contain read-after-write dependences if the same bin is updated in neighboring iterations.
2) *Matrix power* performs a series of matrix–vector multiplications. Each iteration of a nested loop reads a row and a column coordinate and updates the corresponding matrix element. At compile time, it is not possible to determine if successive iterations perform conflicting writes and reads.
3) *Matching* performs the maximal matching algorithm, which iterates through the edges of a graph and checks whether their endpoint vertices are marked; if this is not the case, the vertices are updated using conditional stores. There are potential read-after-write dependences between the stores and the loads from the following iterations.
4) *If loop add* is the kernel discussed in Section II, with a potential dependency across loop iterations.
5) *If loop mul* is a variation of the previous kernel where we replace the conditional addition with a multiplication of the same variables and which we will contrast with the previous kernel in terms of resource utilization.
6) *FIR* is an ordinary FIR filter calculating the output based on the inputs and the coefficients. The memory reads and writes are independent and disambiguated at compilation.
7) *Matvec* is a standard matrix–vector multiplication; as in the previous case, all memory accesses can be disambiguated during compilation.

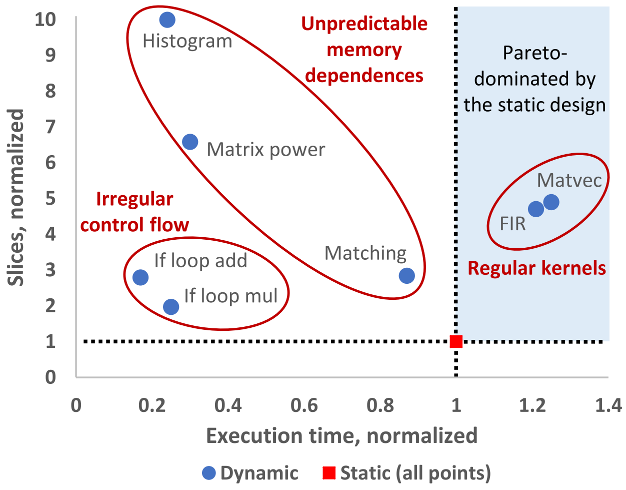


Fig. 15. Resource utilization and execution time of the dynamically scheduled designs, normalized to the corresponding static designs produced by Vivado HLS.

C. Results: Comparison With Static HLS

Table I summarizes the timing and resource results for all kernels and Fig. 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).

Timing: 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 (note that the dynamic results are data dependent: the best possible II is achieved when there are no dependences and the worst possible II when all neighboring iterations are dependent; this II corresponds to the statically computed one). The additional dataflow control logic (i.e., the merge, branch, fork, and join units which we insert into the design) typically have an acceptable impact on the CP. The critical path of the LSQ is extremely sensitive to the number of queue entries [32]; hence, it also impacts the achieved CP. Although this timing overhead is quite tangible, it is still conspicuously small when compared to the potential improvement in II and, consequently, the net performance. 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 [47] 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.

Resource Utilization: The right of Table I contrasts the resource utilization of statically and dynamically scheduled circuits. 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 that we desire. The overhead of the FIFOs that we introduced to increase throughput, as discussed in Section IV-D, is probably overblown by the simplicity of the examples with only a few functional units. Additionally, Vivado employs allocation and binding algorithms to share (i.e., time multiplex) functional units among operators; sharing is possible without a performance penalty due to the low throughput, which the static designs achieve. Since all the dynamic designs achieve high-throughput pipelines, sharing units is not possible without compromising throughput; we, therefore, allocate a new unit per operator, which contributes to the resource difference between the static and dynamic design. For instance, our *If loop add* design requires two functional units to perform the addition and the subtraction whereas Vivado HLS time-multiplexes the same one (as evident from the DSP usage). By replacing one of the operations with a multiplication (i.e., *If loop mul*), we verified that the DSP count is now equal and the overall resource difference is smaller.

It is immediately visible from Fig. 15 that the circuits requiring an out-of-order memory interface demand significant additional resources. Although others have accelerated similar kernels to a qualitatively comparable extent and with only insignificant overhead [18], their solution is highly specific and solves only a subset of problems discussed in this work. 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).

VIII. Related Work

Standard commercial (e.g., Vivado HLS [54]) and academic (e.g., LegUp [7], PandA [44]) HLS tools rely on a static schedule, determined at compile time; this schedule dictates the clock cycle in which each operation will execute. Pipelining is typically achieved through modulo scheduling [6], [47], [55]: the aim is to minimize the loop II under the given clock and resource constraints. In regular applications, this approach results in high-throughput pipelines; however, when memory accesses or control decisions are not determinable during code compilation, the HLS tool must make pessimistic scheduling assumptions, often yielding inferior schedules and lower performance.

Recent advances in HLS have explored methods to overcome the conservatism in static scheduling. Several techniques [1], [38] 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 the code copy to execute. Tan *et al.* [49] described an approach called ElasticFlow to apply loop pipelining on a particular class of irregular loops. Dai *et al.* [17] proposed methods for pipeline flushing by performing scheduling for multiple IIs of the pipeline; they later developed application-specific dynamic hazard detection circuitry [18] and have shown the ability of speculation but with stringent constraints (e.g., stateless inner-loop datapath). Nurvitadhi *et al.* [43] performed automatic pipelining, assuming that the datapath is already partitioned into pipeline stages. To effectively tolerate variable memory latencies, several authors propose prefetching and access/execute decoupling; they rely on complex compilation techniques to automatically separate data access and address calculations from value computations [12], [27]. 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.

Different authors exploited latency-insensitive protocols [8], [15], [21] to construct synchronous and asynchronous dataflow circuits. Elastic circuits [15] are probably the best-studied form of latency insensitivity, but the original paradigm is too restrictive for HLS. Several approaches [11], [28] extended the SELF protocol [15] with constructs similar to the branch and merge which we use in this work. Kam *et al.* [37] showed the ability of elastic circuits to create dynamic pipelines, but do not provide generic transformations to create them out of high-level descriptions. Efforts in the asynchronous domain, such as Balsa [20] and Haste/TiDE [42], applied syntax-driven approaches for mapping a program into a structure of handshake components [48]; a synchronous backend for Haste/TiDE has later been developed. Putnam *et al.* [46] also explored synthesizing dataflow-like circuits from high-level specifications. Townsend *et al.* [51] synthesized dataflow networks from functional programming representations. Dataflow circuits, with their handshake signals, bring to mind Bluespec and its firing rules [52]. However, all these approaches provide little information on some critical conversion aspects that are at the heart of this article; to the best of our knowledge, these approaches have never been contrasted to modern HLS tools.

The efforts closest to ours are the work by Huang *et al.* [29] and Budiu *et al.* [3], [4]. Huang *et al.* [29] mapped dataflow circuits generated from C code to a coarse-grain reconfigurable array. Their circuit generation differs from ours in two aspects: 1) they use a single branch node per BB, thus synchronizing all the BB outputs and preventing pipelining and 2) they do not employ an LSQ; all memory accesses which may conflict need to be conservatively sequentialized. Budiu *et al.* described a compiler for generating *asynchronous circuits* from C code [3], [4]. Although their final circuits are fundamentally different from ours (our circuits are *synchronous* and avoid the traditional difficulties of asynchronous design), the generation strategy is similar to ours. Unfortunately, the exact methodology is never described in full detail; although they also employ an LSQ to handle memory dependences, their allocation policy is more conservative than ours.

Several HLS approaches explore coarse-grained dataflow design. Commercial HLS tools support task-level pipelining (i.e., "dataflow optimization" [53]), which overlaps functions and loops, connected via FIFOs, to increase throughput and concurrency. The FIFOs are typically sized conservatively by the HLS tool so that they can hold all data exchanged between tasks; it is up to the user to identify and manually specify the appropriate FIFO sizes such that the resource utilization is minimal and that deadlock never occurs. Furthermore, the optimization is applicable only to tasks without bypass, feedback, or conditionals between each other. In contrast, our approach successfully supports cyclic behavior and conditionals and is able to compute the required FIFO sizes even in those cases, while at the same time ensuring the absence of deadlock. Cheng and Wawrzynek [13] described sequential programs as networks of processes in which hardware accelerators exchange data via FIFOs. Geilen *et al.* [24] used model checking to minimize buffer requirements in coarser synchronous dataflow graphs (SDFs). Govindarajan *et al.* [25] targeted large-grain actor graphs and presented an approach to minimize buffer storage while executing at the optimal computation rate. Castellana and Ferrandi [9] presented an HLS flow for generating dynamically scheduled accelerators that use an adaptive distributed controller to implement coarse-grained parallelism, concurrent function calls, and variable-latency operations. In contrast to all these works, we explore fine-grained dataflow design (i.e., scheduling individual loop and function datapaths) and focus on methods to exploit instruction-level parallelism in the presence of irregular memory accesses and control flow. Our circuit generation strategy supports constructs that typically appear in imperative high-level languages, our buffer placement method guarantees optimality and the absence of deadlock, and our memory interface dynamically resolves dependences that are undeterminable at compile time.

IX. Conclusion

With FPGAs finding their way into datacenters, HLS tools are set to play a key role in the future of reconfigurable computing. Yet, generating good static circuits from high-level languages requires peculiar code restructuring algorithms (e.g., modulo scheduling), demands expert user interaction (e.g., pragmas and code restructuring), forces worst case assumptions on important issues (e.g., memory and control dependences), and precludes key performance optimizations (e.g., general forms of speculative execution). In this article, we have described a dynamically scheduled form of HLS, which produces dataflow circuits, able to resolve dependences as the circuit runs. When static HLS exploits the maximum parallelism available, our technique achieves similar results with minimal degradation in cycle time; when static HLS misses some key performance optimization opportunities, our circuits seize them, achieving large performance improvements with the investment of more resources. We believe that this avenue of HLS has potential to open new doors for reconfigurable computing and its applications.

References

[1] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proc. 50th Design Autom. Conf.*, Austin, TX, USA, Jun. 2013, pp. 1–10.

[2] *Amazon EC2 F1 Instances*, Amazon, Seattle, WA, USA, 2017.

[3] M. Budiu, P. V. Artigas, and S. C. Goldstein, "DataFlow: A complement to superscalar," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Mar. 2005, pp. 177–186.

[4] M. Budiu and S. C. Goldstein, "Pegasus: An efficient intermediate representation," School Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, Rep. CMU-CS-02-107, May 2002.

[5] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar, "A general model for performance optimization of sequential systems," in *Proc. Int. Conf. Comput.-Aided Design*, San Jose, CA, USA, Nov. 2007, pp. 362–369.

[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 Program. Logic Appl.*, Munich, Germany, Sep. 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.*, vol. 13, no. 2, pp. 1–24, Sep. 2013.

[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. CAD-20, no. 9, pp. 1059–1076, Sep. 2001.

[9] V. G. Castellana and F. Ferrandi, "An automated flow for the high level synthesis of coarse grained parallel applications," in *Proc. IEEE Int. Conf. Field Program. Technol.*, Dec. 2013, pp. 294–301.

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

[11] 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, Jun. 2012.

[12] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *Proc. 49th Int. Symp. Microarchit.*, Oct. 2016, pp. 1–12.

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

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

[15] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proc. 43rd Design Autom. Conf.*, San Francisco, CA, USA, Jul. 2006, pp. 657–662.

[16] 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.

[17] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," in *Proc. 51st Design Autom. Conf.*, San Francisco, CA, USA, Jun. 2014, pp. 1–6.

[18] S. Dai *et al.*, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proc. 25th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Monterey, CA, USA, Feb. 2017, pp. 189–194.

[19] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York, NY, USA: McGraw-Hill, 1994.

[20] D. Edwards and A. Bardsley, "BALSA: An asynchronous hardware synthesis language," *Comput. J.*, vol. 45, no. 1, pp. 12–18, Jan. 2002.

[21] S. A. Edwards, R. Townsend, and M. A. Kim, "Compositional dataflow circuits," in *Proc. 15th ACM/IEEE Int. Conf. Formal Methods Models Syst. Design*, Sep. 2017, pp. 175–184.

[22] J. Forrest *et al.*, "coin-or/Cbc: Version 2.9.9," Zenodo. Jul. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1317566

[23] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky, "Speculation in elastic systems," in *Proc. 46th Design Autom. Conf.*, San Francisco, CA, USA, Jul. 2009, pp. 292–295.

[24] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proc. 42nd Design Autom. Conf.*, Anaheim, CA, USA, Jun. 2005, pp. 819–824.

[25] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks," *J. VLSI Signal Process. Syst. Signal Image Video Technol.*, vol. 31, no. 3, pp. 207–229, Jul. 2002.

[26] M. R. Greenstreet and K. Steiglitz, "Bubbles can make self-timed pipelines fast," *J. VLSI Signal Process.*, vol. 2, no. 3, pp. 139–148, Nov. 1990.

[27] T. J. Ham, J. L. Aragón, and M. Martonosi, "Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures," *ACM Trans. Architect. Code Optim.*, vol. 14, no. 2, pp. 1–27, Jun. 2017.

[28] G. Hoover and F. Brewer, "Synthesizing synchronous elastic flow networks," in *Proc. Design Autom. Test Europe Conf. Exhibit.*, Munich, Germany, Mar. 2008, pp. 306–311.

[29] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in *Proc. 21st ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Monterey, CA, USA, Feb. 2013, pp. 171–180.

[30] L. Josipović, "High-level synthesis of dynamically scheduled circuits," Ph.D. dissertation, School Comput. Commun. Sci., EPFL, Lausanne, Switzerland, Nov. 2020.

[31] 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 Program. Technol.*, Dec. 2019, pp. 197–205.

[32] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–125, Sep. 2017.

[33] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proc. 26th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Monterey, CA, USA, Feb. 2018, pp. 127–136.

[34] L. Josipović, A. Guerrieri, and P. Ienne, "Speculative dataflow circuits," in *Proc. 27th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Seaside, CA, USA, Feb. 2019, pp. 162–171.

[35] L. Josipović, A. Guerrieri, and P. Ienne, "Dynamatic: From C/C++ to dynamically scheduled circuits," in *Proc. 28th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Seaside, CA, USA, Feb. 2020, pp. 1–10.

[36] 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 Program. Gate Arrays*, Seaside, CA, USA, Feb. 2020, pp. 186–196.

[37] 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.

[38] 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 Program. Custom Comput. Mach.*, May 2015, pp. 159–162.

[39] (2018). *The LLVM Compiler Infrastructure*. [Online]. Available: http://www.llvm.org

[40] *Mentor Graphics*, ModelSim, Wilsonville, OR, USA, 2016.

[41] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[42] 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, NC, USA, May 2009, pp. 185–194.

[43] 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.

[44] (2020). *PandA*. [Online]. Available: https://panda.dei.polimi.it/

[45] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. 41st Int. Symp. Comput. Architect.*, Minneapolis, MI, USA, Jun. 2014, pp. 13–24.

[46] 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 Program. Gate Arrays*, Monterey, CA, USA, Feb. 2017, pp. 173–178.

[47] B. R. Rau, "Iterative modulo scheduling," *Int. J. Parallel Program.*, vol. 24, no. 1, pp. 3–64, Feb. 1996.

[48] J. Sparsø, "Current trends in high-level synthesis of asynchronous circuits," in *Proc. 16th IEEE Int. Conf. Electron. Circuits Syst.*, Dec. 2009, pp. 347–350.

[49] 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, USA, Nov. 2015, pp. 78–85.

[50] L. Torczon and K. Cooper. *Engineering a Compiler*, 2nd ed. London, U.K.: Morgan Kaufmann, 2011.

[51] 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, USA, Feb. 2017, pp. 76–86.

[52] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *Proc. 9th Int. Conf. Formal Methods Models Codesign*, Cambridge, MA, USA, Jul. 2009, pp. 171–180.

[53] *Vivado Design Suite User Guide: High-Level Synthesis*, Xilinx Inc., San Jose, CA, USA, 2018.

[54] *Vivado High-Level Synthesis*, Xilinx Inc., San Jose, CA, USA, 2018.

[55] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *Proc. 32nd Int. Conf. Comput.-Aided Design*, San Jose, CA, USA, Nov. 2013, pp. 211–218.

Lana Josipović (Member, IEEE) received the B.Sc. and M.Sc. degrees in electrical engineering and information technology from the University of Zagreb, Zagreb, Croatia, in 2013 and 2015, respectively, and the Ph.D. degree in computer and communication sciences from the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, in 2021.

Her research interests include high-level synthesis, compilers, and reconfigurable computing.

Dr. Josipović is a recipient of the Best Paper Award at FPGA'20, Google Ph.D. Fellowship in Systems and Networking, and Google Women Techmakers Scholarship.

Andrea Guerrieri received the M.Sc. degree in electronic engineering from the Politecnico di Torino, Turin, Italy, in 2015.

He started working on embedded systems in 2006. In 2017, he joined the Processor Architecture Laboratory, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, where he leads and participates in research projects in collaboration with industry. Recent projects involve high-level synthesis, reconfigurable SoCs, and exploiting dynamic partial reconfiguration of FPGAs for future space missions and exoplanet observation.

Mr. Guerrieri is a recipient of the Best Paper Award at FPGA'20, and he is also a co-developer of Dynamatic.

Paolo Ienne (Senior Member, IEEE) received the Laurea degree in electrical engineering from the Politecnico di Milano, Milan, Italy, in 1991, and the Ph.D. degree in computer science from the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 1996.

Since 2000, he has been a Professor with the School of Computer and Communication Sciences, EPFL. He has published over 200 articles in peer-reviewed journals and international conferences, some of which have received the Best Paper Awards at prestigious venues (including the FPGA, FPL, CASES, and DAC conferences).

Prof. Ienne is an Associate Editor of *ACM Computing Surveys* and *ACM Transactions on Architecture and Code Optimization*. He serves on the steering committee of the ARITH, FPL, and FPGA conferences. He is a member of ACM.