Dynamically Scheduled High-level Synthesis

Lana Josipović, Radhika Ghosal, and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland

ABSTRACT

High-level synthesis (HLS) tools almost universally generate statically scheduled datapaths. Static scheduling implies that circuits out of HLS tools have a hard time exploiting parallelism in code with potential memory dependencies, with control-dependent dependencies in inner loops, or where performance is limited by long latency control decisions. The situation is essentially the same as in computer architecture between *Very-Long Instruction Word* (VLIW) processors and dynamically scheduled superscalar processors; the former display the best performance per cost in highly regular embedded applications, but general purpose, irregular, and control-dominated computing tasks require the runtime flexibility of dynamic scheduling. In this work, we show that high-level synthesis of dynamically scheduled circuits is perfectly feasible by describing the implementation of a prototype synthesizer which generates a particular form of latency-insensitive synchronous circuits. 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: in demanding applications, the performance is very significantly improved at an affordable cost. We here demonstrate only the first steps towards more performant high-level synthesis tools adapted to emerging FPGA applications and the demands of computing in broader application domains.

ACM Reference Format:
Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174264

1 INTRODUCTION

The use of FPGAs in datacenters by Microsoft [7, 35] and Amazon [2] as well as the acquisition of Altera by Intel [10] signal 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 through modern programming paradigms. While there is conspicuous research activity on this front, practically all attempts ultimately rely on classic forms of *High-Level Synthesis* (HLS) to generate the actual circuits. In turn, 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 [20]. The similarity to code generation for *Very-Long Instruction Word* (VLIW) processors is all but accidental: much of the key transformations to exploit fine-grain parallelism between operators derives from VLIW compilation techniques [28, 37]. This analogy with computer architecture is enlightening: Around two decades ago, Intel started working on the now defunct Itanium architecture [17], the first and only VLIW processor aimed to general-purpose markets. Unfortunately, it turned out significantly more difficult than expected for a compiler to extract the parallelism that dynamically scheduled processors routinely exploit. Today, VLIW processors are successful only in markets with extremely regular and predictable applications, and where it is acceptable to tune code manually.

Perhaps HLS and FPGAs are following the same trajectory: Statically scheduled HLS serves well applications that are fairly regular and when development time is measured against coding in RTL languages. But, with FPGAs moving to datacenters and facing broader classes of applications, the ability of dynamic scheduling to automatically extract parallelism may prove essential. With dynamic scheduling, not only complex loop transformations (and related hints from the programmers) are often unnecessary, but more parallelism can be extracted in the presence of control and memory dependencies undecidable at compile time. Although beyond the scope of this paper, dynamically scheduled circuits open the door to speculative execution, one of the most powerful ideas ever in computer architecture. If FPGAs should compete with CPUs running on one order of magnitude faster clocks, they will need every ounce of exploitable operation parallelism with minimal programmer effort.

This paper presents a methodology to automatically generate dynamically scheduled circuits from 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 paper is organized as follows: Section 2 explores an example of one of the situations where dynamic extraction of operation-level parallelism proves essential to performance. Section 3 details our circuit generation methodology as implemented in our prototype tool. Section 4 gives the results of the comparison of our technique with static HLS and contrasts our methodology with previous efforts to create dynamically scheduled circuits. In Section 5, we discuss some of the future perspectives opened by our circuit generation strategy. In Section 6, we outline what others have done to circumvent some of the problems of statically scheduled HLS, before concluding the paper in Section 7.

2 WHY DYNAMIC SCHEDULING?

To illustrate the limitations of standard HLS approaches, consider the code in Figure 1a. 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 which might take place in a specific iteration (`s = s + d`) introduces a dependency between iterations and delays

FPGA 2018, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174264

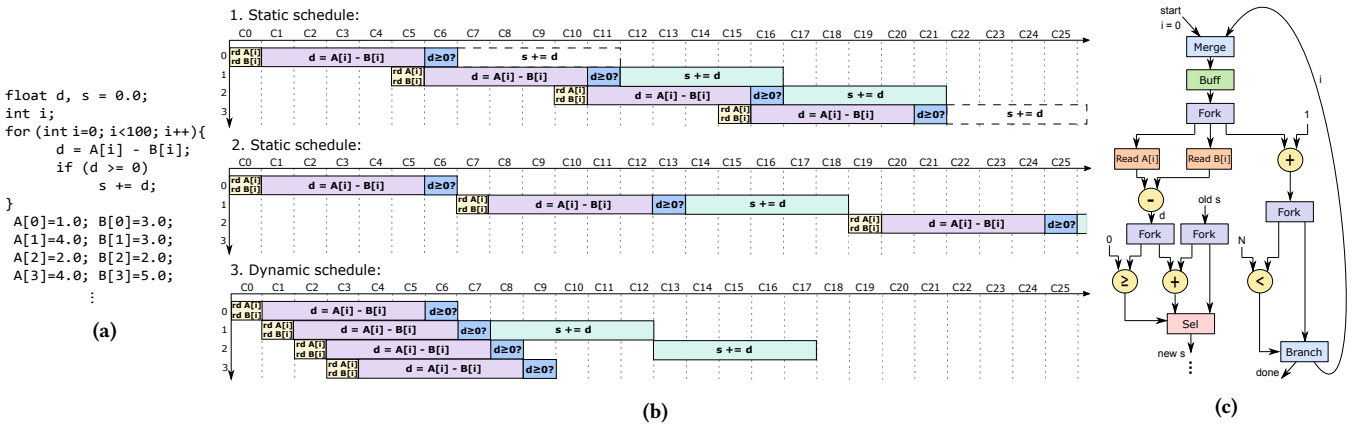

Figure 1: Limitations of static scheduling. Figure 1a gives a code segment where dependencies cannot be determined at compile time. Figure 1b contrasts two possible schedules (top and middle) created by an HLS tool with a dynamic schedule (bottom). Figure 1c shows a portion of a dynamically scheduled circuit achieving the optimal execution schedule of Figure 1b.

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 is shown on the top of Figure 1b: in the example the condition is true only for the second and third iteration but "space" is reserved in the schedule as if the condition were true everywhere. An alternative could be to avoid pipelining the loop and creating a sequential finite-state machine. The result could be the middle schedule in Figure 1b, 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 bottom one in Figure 1b: 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 dependencies 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 dependencies through memory—we will discuss them in Section 6—but dynamically scheduled circuits represent the most general solution to the problem.

2.1 Elastic Circuits

The key to avoid the limitations of static scheduling is to refrain from triggering the operators through a centralized pre-planned controller but to take 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. In line with the computer architecture analogy of the introduction, this is exactly what dynamically scheduled processors do through their reservation stations [23]. 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.

Figure 1c shows a simplified version of an *elastic* circuit [11] implementing the loop of Figure 1a. Besides normal datapath components, this circuit uses a few control components labelled Buff, Merge, Sel, Fork, and Branch. All data signals are accompanied by handshake control signals. The handshake signals are two, in opposite direction, indicating respectively the availability of a new piece of data from the source component and the readiness of the target component to accept it. The loop to the right of the figure shows the part of the circuit which updates the iterator `i`: At the beginning, the constant 0 is sent from the start node. The Merge node takes this value and passes it further. The elastic buffer node Buff is the register which holds `i` and distributes it on the next clock cycle to three consumers through the Fork node; all successors must consume the value before Fork accepts a new input value. The right 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 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 right of Figure 1c is completely combinational excluding the register Buff and thus a new value for `i` can be computed on every cycle. It is the left part of the circuit which can delay this: when the `if` is not taken, the result of the addition is dumped by the Sel node as soon as it arrives and the old value of `s` becomes immediately the new value that is sent back to the adder on the following cycle (loop and Buff omitted for simplicity); if, on the other hand, the result is needed, Sel will wait for the sum to complete, the adder will be stalled next cycle waiting

for its right operand, a new subtraction will not be computed and the memory accesses will not be performed due to backpressure from the adder. Ultimately, 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 Figure 1b shows.

2.2 Dynamic vs. Static Scheduling

As the example above shows, loop pipelining happens naturally in an elastic circuit such as the one in Figure 1c. Again, this is in line with the experience in computer architecture: whereas complex compilation techniques have been developed for VLIWs with the purpose of transforming loops to exploit instruction-level parallelism (often requiring either complex heuristics to drive the optimization or pragmas from the programmers), dynamically scheduled out-of-order processors are capable of achieving good levels of parallelism on-the-fly and without extensive code preparation. Apart from the advantage of exploiting parallelism when static scheduling cannot, this ability to find the "good" solution without 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 [22]) and ultimately by software programmers.

It is clear that, as in the case of processors, taking scheduling decisions dynamically costs resources and time (such as the area and delay of the control elements in Figure 1c). Our purpose in this paper is (1) to show how one can generate automatically dynamically scheduled circuits from C-programs and (2) to compare their circuit complexity and critical path with those of circuits obtained through state-of-the-art HLS tools. Although the potentials of gain in terms of cycles saved are at least qualitatively clear, to the best of our knowledge, the problems at bullet (1) have never been thoroughly studied in the domain of modern, synchronous HLS and the comparison at bullet (2) has never been attempted.

3 SYNTHESIZING ELASTIC CIRCUITS

In this section, we describe the process we use to convert an arbitrary piece of code into a dynamically scheduled circuit. As evoked in Section 2.1, we have chosen the elastic circuits [11] as the paradigm for the circuits we produce. This section is organized as follows: In Section 3.1, we describe the elastic components and Section 3.2 shows how to use them to build datapaths corresponding to basic blocks of code. Section 3.3 discusses a few problems which appear when datapaths from different basic blocks are interconnected following the control flow of the program. Section 3.4 illustrates how we add registers to our circuits to produce correct functionality. The next problem is connecting the design to memory, which we describe in Section 3.5. There is an essential optimization to extract reasonable performance; we outline it in Section 3.6.

3.1 Elastic Components

This section provides an overview of the elastic primitives we use. Inspired by asynchronous circuits, elastic circuits are strictly synchronous and perfectly adapted to traditional VLSI and FPGA flows.

Most elastic components are immediately equivalent to ordinary datapath components but they implement latency-insensitivity by communicating with their predecessors and successors through point-to-point pairs of handshake control signals: a *valid* signal

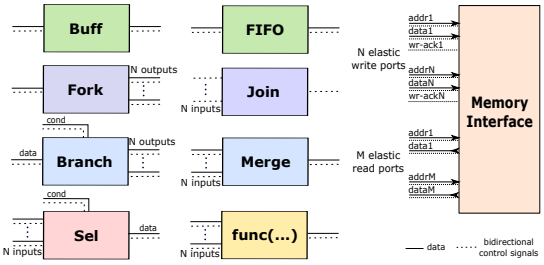

Figure 2: Elastic components.

indicates that a component is sending a valid piece of data to its successor(s), whereas the *ready* signal informs the predecessor(s) that a component can accept a new piece of data. The availability of a piece of data is colloquially indicated as the presence of *token*, for analogy with Petri nets; tokens indicate valid data and a transition (event) occurs as a component absorbs the token [32].

Figure 2 outlines the elastic components we use. Their gate-level descriptions can be found in literature [11, 25] and none is original of this piece of work. All have the above-mentioned *ready* and *valid* control signals and most are associated to a data component.

- *Elastic Buffers* (Buffs) are the elementary storage structure of elastic circuits and the immediate equivalent of D flip-flops or registers in regular circuits.
- *Elastic FIFOs* (FIFOs) are ordinary first-in first-out queues with the appropriate handshaking signals.
- An *Eager Fork* (Fork) replicates every token received at the input to multiple outputs; it outputs tokens to each successor as soon as possible (i.e., as soon as each individual successor is ready to accept the data) but does not accept any new token until all successors have accepted the previous one.
- A *Lazy Fork* (LFork) performs essentially the same function as an eager fork, but it outputs tokens only when all successors are ready. It is, in general, a less optimized version of an eager fork, as the more conservative triggering rule reduces the opportunities for out-of-order execution. Yet, it will be useful in Section 3.5 in a very specific part of the circuit where we will need to emit tokens in a particular order.
- A *Join* (Join) is the reciprocal of an Fork—it acts like a synchronizer by waiting to receive a token on each and every one of its inputs before emitting a token at its output. We seldom employ Joins explicitly but they are used in components requiring multiple operands to trigger advancement.
- A *Branch* (Branch) implements program control-flow statements (i.e., *if* or *switch*) by dispatching a token (and, sometimes, the corresponding piece of data) received at its single input to one of its multiple outputs based on a condition.
- A *Merge* (Merge) is a reciprocal of a Branch—it propagates a token and data received on any input to its output. Merges are analogous to Φ functions in the static single assignment form, inserted in points where control-flow paths meet [40].
- A *Select* (Sel) acts as a multiplexer—it waits for the required input to produce the output and discards the tokens at the nonselected inputs as soon as they arrive.

In addition, we use any functional unit the code requires, such as integer and floating point units. Components that require multiple operands contain a Join to trigger the operation only when all inputs

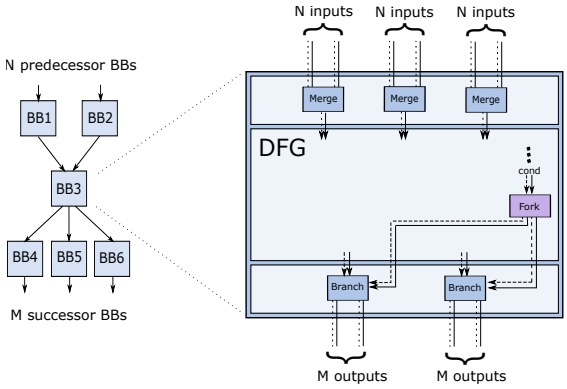

Figure 3: The basic block template.

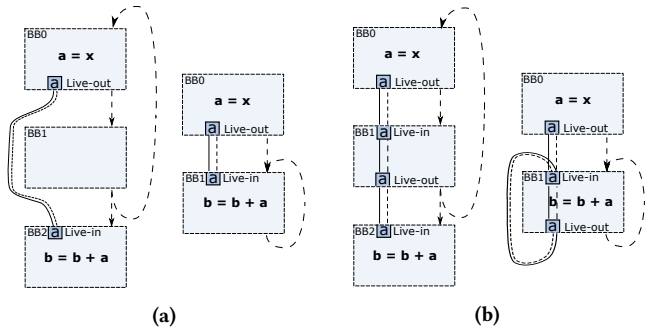

Figure 4: Implementing control flow. Figure 4a shows two cases where a direct conversion of a data and control flow graph into an elastic circuit would fail. Coupling data and control to ensure correct token transfers between BBs is given in Figure 4b.

are available. Finally, we interface with memory through *elastic memory ports*. The write port has two inputs (*data* and *address*) and a control-only signal from the memory interface indicates successful completion. The read port sends an address to memory and receives data with its corresponding elastic control. Yet, to achieve correct execution in an intrinsically out-of-order system, interfacing to memory is more challenging than just connecting to the appropriate memory ports; this will be addressed in Section 3.5.

3.2 From Basic Blocks to Datapaths

We use the standard data and control flow graphs obtained by a compiler as the starting point of our circuit generation. First, we create a datapath for each *basic block* (BB)—i.e., for each straight piece of code not containing conditionals. The basic conversion is a literal translation of the data flow graph into an elastic circuit: every operator corresponds to a functional unit, edges are connections between the components, and a Fork is added when a node has more than one direct successor and at least one is in the same basic block. At this point, our circuit does not contain any register (Buff).

BBs are connected by directed edges representing data and control transfers. Once the datapath of each BB has been built, we need to connect them to other datapaths (or BBs). We have chosen the conversion template of Figure 3. We allocate a Merge node for every variable entering the BB (*live-in*). Every Merge receives a piece of data with the corresponding token from one of the predecessor BBs and forwards it to the main datapath. In the example in Figure 3, BB3 accepts three live-in variables from one of its two predecessor BBs—note that, as the circuit follows the control flow, only one of the predecessor blocks is active at any point in time (i.e., *BB*1 and *BB*2 will never send tokens to *BB*3 simultaneously) and this is key to deadlock avoidance (see Section 3.4).

To implement control flow decisions, for every value used by any successor BB (*live-out*), we place a Branch at the BB output. If the successor block does not require a particular data, the output of the corresponding Branch is discarded into a sink.

3.3 Implementing Control Flow

Connecting the datapath elements corresponding to BBs (Figure 3) is relatively straightforward, except for a couple of problems which arise from the fundamental difference of software programs implemented on a processor compared to elastic circuits.

Figure 4a shows two examples: (1) In the example on the left, the variable a is defined in BB0 and used in BB2. A typical representation in a compiler propagates the desired information directly from the source to the destination block (i.e., a live-in of a basic block comes from a basic block 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 on the right, 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, there would be no problem in a processor—the value would be stored in a register or memory and read as many times as needed.

Directly implementing such connections in an elastic circuit would result in incorrect behavior because every value is associated with a control token: the number of generated tokens must exactly match the number of distinct uses. Both cases in Figure 4a violate this principle: (1) In the first case, if the control flow were {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 wants only the most recent value. 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 corresponds to strictly coupling data propagation with control flow, as Figure 4b shows. We modify the data and control flow graphs to ensure that (1) every BB provides a live-out for every live-in of all of its immediate successor BBs and exclusively to them, and that (2) every BB receives all of its live-ins from its immediate predecessor BBs and exclusively from them. We implement this by identifying the origin block for every live-in variable of every BB. We then find all the paths of the control flow graph connecting the origin BB and the BB that the live-in belongs to, and we reconnect the variable through these paths. This approach guarantees that every piece of data for a BB receives a fresh token each and every time the BB actually executes.

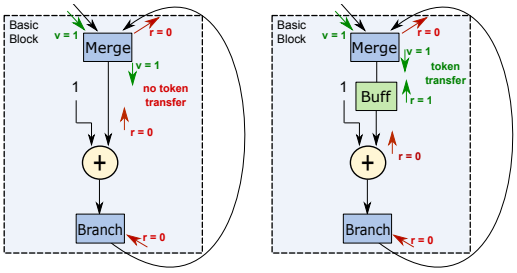

Figure 5: Adding registers. A combinational cycle causes deadlock, as the token with the updated data cannot propagate back into the Merge node. Breaking the combinational loop with an Buff enables the token to loop back.

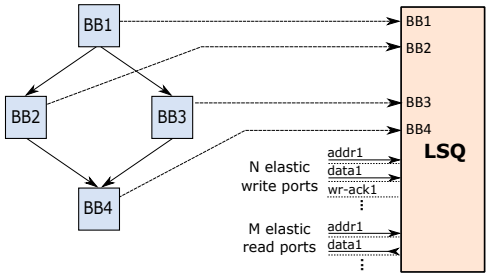

Figure 6: The load-store queue required for correct out-of-order memory accesses.

3.4 Adding Registers

So far, our circuits do not contain any registers. Before illustrating our strategy for Buff placement, we discuss their impact on the circuit functionality and their role in avoiding deadlock.

Elastic buffers and circuit functionality. Elastic 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 late items [21]. Although Buffs shift the values in time with respect to the pure synchronous behavior, their presence or absence does not affect the functional correctness of the system, as any consumer of multiple values synchronizes the corresponding valid tokens. Contrary to registers in traditional synchronous designs, this characteristic allows the insertion of Buffs on any wire without any effect on functionality but only on performance. In other words, insertion or removal of Buffs is correct by construction, as it preserves flow equivalence and guarantees an unchanged order of valid data [11, 21].

Elastic buffers and avoiding deadlock. The necessary and sufficient condition for deadlock-free execution requires any cycle in the circuit to contain at least one Buff [12, 19]. Figure 5 contrasts a design of a simple cycle with and without a Buff on the cyclic path—in the first case, a token inserted into the Merge node cannot propagate through the loop due to the combinational relationship of the valid and ready tokens on the cycle (labeled as v and r in the figure). As in traditional synchronous circuits, the combinational loop needs to be broken through a register. Adding a Buff ensures that a token can propagate through the cycle [11]. Once the cyclic combinational relations have been resolved, all tokens will flow through the circuit in the absolute execution order specified by the original program; if the program terminates, the token inserted in the start BB will eventually reach the end BB, following the control flow of the program; this guarantees the absence of deadlock.

Once the requirement above is satisfied, adding more Buffs has no influence on functionality but only on performance by (1) delaying the corresponding piece of data by a clock cycle and (2) by breaking a combinational path in the circuit into two paths, possibly reducing the critical path of the circuit. A Buff can be placed on any edge of the graph of elastic nodes (i.e., between any two elastic components). Each edge is associated with its weight, equivalent to the number of bits of the corresponding data. We define an optimal Buff placement as one which ensures that (1) every graph cycle is cut by at least one Buff or sequential element within a functional unit and (2) the sum of the weights of all cut edges is minimized. In our experiments we implement a simple heuristic to approximate this optimal placement (and, in practice, we think that in all our experiments the placement is optimal in the above sense).

3.5 Connecting to Memory

Figure 2 shows a memory component with elastic read and write ports (i.e., an elastic interface to a traditional memory hierarchy). 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 arbitrary order (this is the dynamic out-of-order feature that, in general, we desire) and this may lead to the violation of memory dependencies. For instance, if a write happens at the same address as some successive read, and if the read token arrives in the elastic circuit before the write token, the result of the read will be incorrect.

The solution is to use a *load-store queue* (LSQ) similar to those present in dynamically scheduled processors. Yet, we have shown that building a LSQ for elastic circuits has one fundamental difference [26]: 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 paper; it suffices to say that the key condition for the LSQ to execute correctly is to receive tokens which follow the actual order of execution of the basic blocks of the circuit. This ordering enables the LSQ to determine and resolve dependencies as memory access arguments from different basic blocks arrive out-of-order.

Consider a program containing four basic blocks as given in Figure 6. The difference from the simple memory interface of Figure 2 is only in the additional elastic control signals (e.g., BB1, BB2). These signals indicate to the LSQ the start of the particular BB. When the program starts, BB1 sends a token to the LSQ. Assuming that the control flow determines the execution of BB2 afterwards, BB2 will send a token to the LSQ next. The order of these tokens enables the LSQ to appropriately handle out-of-order memory accesses; accesses from BB1 need to be completed before those from BB2. If, for instance, a read request arrives from BB2 before all writes from BB1 have been completed (or determined independent of the read), the LSQ will appropriately stall the execution of the read.

Our challenge here is to guarantee that the BBi signals needed for 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 generate a control path that follows the control flow of

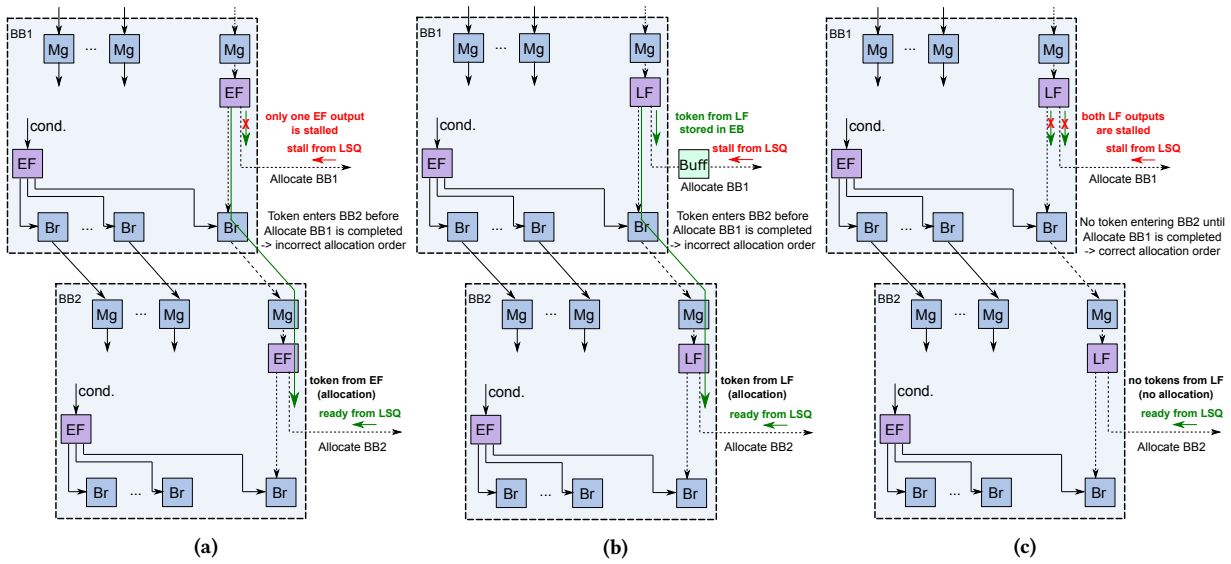

Figure 7: Connecting the elastic circuit to the memory interface. Figures 7a and 7b give examples of incorrect connections. In Figure 7a, the Eager Fork may send an allocation to BB2 before the allocation of BB1 completes. In Figure 7b, the allocation order may be reversed due to the storage element on the control line between the circuit and the LSQ. Figure 7c shows the correct way to connect the LSQ—an allocation cannot occur unless all predecessor allocations have been completed.

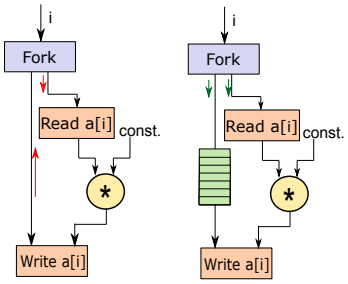

Figure 8: Increasing parallelism by adding FIFOs.

the program through the BBs—essentially, as a data-less variable which is a live-in and live-out of each and every BB. The tokens in this path trigger the allocation of BBs as soon as the control flows there (i.e., as soon as a decision has been made to enter a particular BB). However, applying the standard elastic circuit design strategy described in the previous sections might result in the incorrect order of token arrival to the LSQ. Here are two example situations leading to a potentially wrong execution: (1) If the token is forked to the LSQ using the typical Eager Fork (Fork), one of the fork outputs might send a token to the next BB before the LSQ has accepted a token from its predecessor (Figure 7a). (2) Although placing Buffs in elastic circuits has no impact on correctness (as discussed in Section 3.4), an Buff 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 Buff, the successor BB could send a new token before the prior allocation has been completed (Figure 7b).

The correct way to connect the LSQ to the elastic circuit is given in Figure 7c: (1) The forks used to send the tokens to the LSQ are Lazy Forks (LFork)—if one of the fork outputs is stalled, the other one will stall as well. (2) No sequential elements (Buffs) 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 which cannot have dependencies with any other accesses are connected to simple memory interfaces.

3.6 Decoupling Paths for More Parallelism

The methodology described so far results in semantically correct circuits; however, they may not yet be competitive with statically scheduled circuits: A Fork, used to distribute some value to potentially independent pipelines, does pass the token to any successor as soon as it is ready to take it, but, as mentioned in Section 3.1, does not accept a new token until all successors have consumed the previous one. Since some paths through a basic block take longer to process a token, a Fork may prevent a shorter path to execute faster. A critical example is the Fork distributing the condition to all Branches, shown in Figure 3: Even if the control decision is resolved quickly, the Fork accepts a new condition only when all Branch nodes receive their values. This prevents hardware pipelining; even if the need for another iteration can be decided very fast, the new iteration will not start until the current iteration finishes.

To increase the effective parallelism, we decouple the fast and slow paths of the basic block by inserting FIFOs into the paths with longer latency. This allows token accumulation without blocking the Fork and thus allows to trigger the faster paths at a higher rate. Figure 8 contrasts the naive slow design with the FIFO-optimized version. This modification is sufficient to overlap iterations of a

loop at a rate which corresponds to the speed at which the control decision can be made. Algorithmically determining the optimal size of the FIFOs seems akin to buffer sizing in networking and has been discussed in the context of dataflow machines [13]. We have not yet properly studied the problem, given the fact that it affects only performance and not correctness. In this work, we place a FIFO on every Fork output and experimentally determine its optimal size.

3.7 A Complete Flow

This section has shown how an arbitrary program described in a high-level language can be transformed into a dynamically scheduled circuit. The resulting circuit executes operations out-of-order, naturally implements hardware pipelining, and handles efficiently potential memory dependencies. Although our transformation flow is susceptible to improvements, we think it is interesting to compare it with a mature HLS tool producing statically-scheduled circuits, as well as with approaches similar to ours.

4 EVALUATION

We describe here our prototype synthesizer, then give an overview of our methodology to compare with a commercial HLS tool, and finally discuss our benchmarks before presenting our results.

4.1 Prototype Synthesizer

Our hardware generation flow uses the LLVM compiler framework [30]: (1) The clang frontend parses the C/C++ program and produces a *static single assignment* (SSA) intermediate representation (LLVM IR) [40]. (2) The LLVM optimizer applies standard transformation and analysis passes on the IR. (3) Our custom-made pass transforms the optimized LLVM IR into an elastic circuit. The main steps of this transformation are described in Section 3. (4) The IR of the elastic circuit is converted into a VHDL netlist of the elastic primitives described in Section 3.1. Our flow still includes a few semimanual steps (including some for the comparisons of the next section), but nothing more than what is described in Section 3.

4.2 Methodology

To demonstrate the benefits of using an elastic hardware generation strategy in HLS, we compare our circuits with designs generated by Vivado HLS [43], a state-of-the-art commercial HLS tool. In all Vivado designs, we apply the pipelining optimization directive.

To provide a fair comparison of our designs against those generated by Vivado, we employ the same arithmetic units used by Vivado into our designs. We extract the components manually from Vivado's results and create custom wrappers with handshake signals. We use the same RAMs for our design as Vivado employs. We rely on the same memory analysis as Vivado: when a compiler cannot disambiguate memory accesses, we manually employ the LSQ in our designs and connect it to the RAM interface; otherwise, we connect the elastic read/write ports directly to the RAM.

We simulated the designs in ModelSim [31] and used a set of test vectors for functional verification. We obtain the 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).

4.3 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 a simple kernel where static scheduling is fully successful, to show that dynamically scheduling achieves virtually the same result with small overheads.

- *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 dependencies if the same bin is updated in neighbouring iterations.
- *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.
- *Loop with condition 1* is the kernel discussed in Section 2, with a potential dependency across iterations dependent on the data from arrays A and B.
- *Loop with condition 2* 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.
- *FIR filter* 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 compile time.

4.4 Results: Comparison with Static HLS

Table 1 summarizes the timing and resource results for all kernels and Figure 9 shows our results relative to those from Vivado HLS (results to the left or below the red circle are better).

Timing. Avoiding conservative assumptions on memory and control dependencies results in a significant improvement of the execution time in all of the corresponding benchmarks (note that the dynamic results are data dependent: the average II can, in all these examples, be as low as 1 and never larger than the statically computed II). This increases the throughput with usually an acceptable impact on the CP due to the additional handshake signals between elastic components. The strongest impact on the CP is when we use the LSQ, whose critical path is extremely sensitive to the number of queue entries [26]. 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* benchmark, static HLS techniques produce a highly optimized pipeline because memory accesses can be disambiguated at compile time. The static HLS tool depends on techniques akin to modulo scheduling [37] to restructure and pipeline the loop, whereas we effortlessly compile the LLVM IR into an elastic circuit as-is: this is the only example of design where our result is Pareto dominated by the static one, but the impact of the elastic circuitry on the CP appears not to be a cause of major concern—especially since nothing was yet attempted to optimize the elastic circuits.

Resource utilization. The right of Table 1 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 elastic

Benchmark	II_{avg}		CP (ns)		Execution time (us)		Slices		LUTs		FFs		DSPs	
	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN
Histogram	11	2.3	3.3	5.7	36.3	13.3	130	200 + 901	296	3,632	447	1,734	2	2
Matrix power	16	4.2	3.4	6.0	20.7	9.6	219	352 + 1113	500	4,237	790	2,050	5	5
Loop with condition 1	9	1.3	2.8	4.8	25.3	6.2	161	289	391	767	525	984	2	4
Loop with condition 2	5	1.2	3.4	4.8	17.1	5.7	187	240	409	659	623	811	5	5
FIR	1	1	3.3	4.4	3.3	4.4	62	127	89	341	224	382	3	3

Table 1: Dynamically scheduled results (our elastic circuits) contrasted to statically scheduled results (Vivado HLS). The slice count for *Histogram* and *Matrix Power* is given as slices of kernel + slices of LSQ.

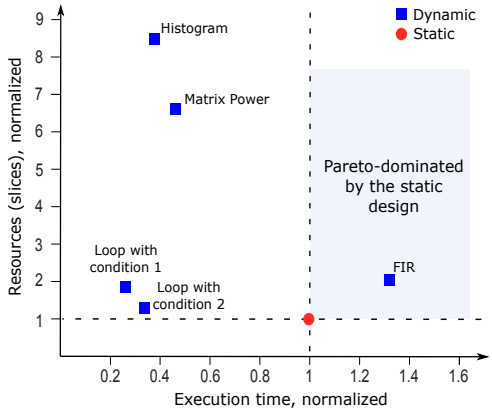

Figure 9: Resource utilization and execution time of the dynamically scheduled designs, normalized to the corresponding static designs produced by Vivado HLS.

circuits contain and which allows them to achieve the latency-insensitivity which we desire. The overhead of the FIFOs that we introduced to increase throughput, as discussed in Section 3.6, is probably overblown by the simplicity of the examples with only a few functional units. Additionally, we have not yet looked into time-multiplexing of functional units—we trivially allocate a new unit per operator, whereas the allocation and binding algorithms that Vivado employs allow a single unit to be shared: see for instance *Loop with condition 1* where our design requires two functional units to perform the addition and the subtraction whereas Vivado HLS time-multiplexes the same one. To show this, we replaced one of the operations with a multiplication (*Loop with condition 2*) and verified that the resource difference is now significantly smaller.

It is immediately visible from Figure 9 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 [15], 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).

4.5 Results: Comparison of Dynamically Scheduled Techniques

In this section, we compare our work to two approaches that are perhaps the closest ones to ours, and discuss some issues preventing them to attain the performance we strive for.

Huang et al. generated elastic circuits from C code, to be mapped to a coarse-grain reconfigurable array [25]. 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 a LSQ at the memory interface and, thus, all memory accesses which cannot be disambiguated at compile time 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 *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. Nevertheless, their best results appear to match qualitatively ours, except when memory accesses are involved: they present two strategies for handling memory dependencies and both are more conservative than ours.

We implemented our two benchmarks with memory dependencies following the design strategies above and compared their timing and resource requirements to our designs. In the case of Budiu et al., we have replaced their asynchronous components with the corresponding synchronous elastic components. Table 2 shows the results. The designs of Huang et al. cannot achieve any pipelining, which results in performances lower than even those of the static HLS designs. For the designs by Budiu et al., we provide two sets of results, corresponding to the two approaches for handling memory dependencies that the authors present: The first version (labeled *CASH 1* in the table) contains no LSQ at the memory interface; as in the work by Huang et al., the authors conservatively sequentialize memory accesses which are potentially dependent; however, they manage to create a pipeline across iterations and achieve some performance improvement compared to Huang et al. In the second version, the authors add an LSQ but use a conservative allocation policy which inserts an entry into the LSQ only when an address or a data item for the corresponding access is known; despite increasing the pipeline throughput, this strategy still cannot match the performance that we achieve using our group allocation policy.

5 PERSPECTIVES

Although many results of Section 4 appear attractive to us, it is also clear that our synthesizer is still primitive in many respects. We think it is fair to emphasize that statically scheduled HLS benefits from decades of research that the automatic design of latency insensitive circuits cannot sport. We spend this section to evoke

Benchmark	II_{avg}				CP (ns)				Execution time (us)				Slices			
	Huang	CASH 1	CASH 2	DYN	Huang	CASH 1	CASH 2	DYN	Huang	CASH 1	CASH 2	DYN	Huang	CASH 1	CASH 2	DYN
Histogram	12	11	3.7	2.3	4.9	4.8	5.9	5.7	58.9	52.9	21.2	13.3	134	149	182+901	200+901
Matrix power	17	16	5.0	4.2	4.1	3.9	6.3	6.0	26.6	23.8	11.9	9.6	204	233	332+1113	352+1113

Table 2: Dynamically scheduled results (our elastic circuits, DYN) contrasted to other dynamic approaches.

some of the most important areas where dynamically scheduled HLS could improve in the future.

5.1 Pipelining and Area Optimizations

Pipelining ordinary synchronous circuits is a thoroughly studied problem [16]. Our heuristic in Section 3.4 is nothing more than first working shot, certainly susceptible of significant improvements—for instance, we did not even try to break critical paths with sequential elements. Other typical concerns of HLS which we did not address here are allocation and binding: deciding how many units of a specific type to implement and how to time-multiplex them to perform needed operations. Elastic circuits can time-multiplex functional units [6] and we will try to exploit this.

5.2 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 component and on any path. Although in some cases this is exactly the strength of our methodology, 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 control flexibility. There may be 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. This is a completely unexplored avenue which might reduce significantly the area and timing overhead of elasticity.

5.3 Speculative Execution

Finally, 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 iteration interval is very large due to a condition on the loop continuation that takes very long to compute: control speculation can predict very early (possibly with an iteration interval of one) whether it makes sense to execute *tentatively* another iteration. Similarly, speculation can further improve the problem of memory dependencies, not only by reordering accesses once the lack of dependency is known but even by assuming independence early on and reverting back if the prediction was wrong. The ability to implement speculation depends on reliable mechanisms to revert state changes due to wrongly executed operations—what in processors is entrusted to reorder buffers and store queues. In the scope of elastic circuits, first steps of speculative execution (much more limited compared to the above goals) have been shown already [21] and suggest that latency insensitive protocols can be modified to accommodate tentative and reversible execution.

6 RELATED WORK

Recent advances in HLS have explored methods to overcome the conservatism in static scheduling. Several techniques [1, 29] generate multiple schedules which are dynamically selected during runtime, once the values of all parameters are known. Tan et al. [39] describe an approach called ElasticFlow to apply loop pipelining on a particular class of irregular loops. Dai et al. [14] propose methods for pipeline flushing by performing scheduling for multiple initiation intervals of the pipeline; they later developed application-specific dynamic hazard detection circuitry [15] and have shown the ability of speculation but with stringent constraints (e.g., stateless inner-loop datapath). Nurvitadhi et al. [34] 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 opportunistically 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 [5, 11, 19] to construct dynamic circuits. Elastic circuits [11] 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 [8, 24] extended the SELF protocol [11] with constructs similar to the Branch and Merge which we use in this work. Kam et al. [27] testified of the ability of elastic circuits to create dynamic pipelines, but do not provide generic transformations to create elastic circuits out of high-level descriptions. Efforts in the asynchronous circuit domain, such as Balsa [18] and Haste/TiDE [33], applied syntax-driven approaches for mapping a program into a structure of handshake components [38], and a synchronous backend for Haste/TiDE has later been developed. Putnam et al. [36] have also explored synthesizing dataflow-like circuits from high-level specifications. However, all these approaches provide little information on some critical aspects of the conversion which are at the heart of this paper; to our best knowledge, these approaches have never been contrasted to modern HLS tools. The efforts closest to ours (i.e., the work by Huang et al. [25] and Budiu et al. [3, 4]) have been discussed in Section 4.5.

Cheng et al. [9] describe circuits as networks of processes in which hardware accelerators exchange data via dynamic communication channels. We are here interested in exploring dynamicity on a finer grain and thus we do not face some of the deadlock issues that are critical in their work. Standard HLS tools [43] also often interconnect with handshakes various datapaths from nested loops and functions but, again, we care here for the fine-grain schedule of individual datapaths. Townsend et al. [41] used a functional programming intermediate representation as a starting point for synthesizing dataflow networks. Elastic circuits, with their handshake signals, immediately bring to mind Bluespec and its firing rules [42]. However, nothing in these two approaches is directly related to our goal: transforming a program written in an imperative, high-level language into a dynamically scheduled circuit.

7 CONCLUSIONS

With FPGAs finding their way into datacenters, HLS tools are set to play a key role in the future of reconfigurable computing. Yet, they are relying on a paradigm which is conceptually identical to the problem of compilation for VLIW processors: generating good static circuits from high-level languages requires peculiar code restructuring algorithms (e.g., modulo scheduling), demands expert user interaction (e.g., pragmas), forces worst-case assumptions on important issues (e.g., memory and control dependencies), and precludes key performance optimizations (e.g., general forms of speculative execution). In this paper, we have described a dynamically scheduled form of HLS and run a simple synthesizer on a few relevant kernels to compare results to a commercial, statically scheduled HLS tool. 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. Although much remains to be done to refine the optimizations and to add key features we have only evoked so far, we believe our work points to a very promising avenue to make HLS truly valuable on irregular and control-dominated applications.

REFERENCES

[1] M. Alle, A. Morvan, and S. Derrien. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Proceedings of the 50th Design Automation Conference*, pages 51:1–51:10, Austin, Tex., June 2013.

[2] Amazon.com, Inc. *Amazon EC2 F1 Instances*.

[3] M. Budiu, P. V. Artigas, and S. C. Goldstein. Dataflow: A complement to superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–86, Austin, Tex., Mar. 2005.

[4] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[5] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-20(9):1059–76, Sept. 2001.

[6] J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky. A scheduling strategy for synchronous elastic designs. *Journal Fundamenta Informaticae*, 108(1–2):1–21, Jan. 2011.

[7] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th International Symposium on Microarchitecture*, pages 1–13, Taipei, Taiwan, Oct. 2016.

[8] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras. xMAS: Quick formal modeling of communication fabrics to enable verification. *IEEE Design & Test of Computers*, 29(3):80–88, June 2012.

[9] S. Cheng and J. Wawrzynek. Synthesis of statically analyzable accelerator networks from sequential programs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 126–33, Austin, Tex., Nov. 2016.

[10] D. Chiou. Intel acquires Altera: How will the world of FPGAs be affected? In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 148, Monterey, Calif., Feb. 2016.

[11] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 657–62, San Francisco, Calif., July 2006.

[12] J. Cortadella, M. G. Oms, M. Kishinevsky, and S. S. Sapatnekar. RTL synthesis: From logic synthesis to automatic pipelining. *Proceedings of the IEEE*, 103(11):2061–75, Nov. 2015.

[13] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, Honolulu, Hawaii, May 1988.

[14] S. Dai, M. Tan, K. Hao, and Z. Zhang. Flushing-enabled loop pipelining for high-level synthesis. In *Proceedings of the 51st Design Automation Conference*, pages 1–6, San Francisco, Calif., June 2014.

[15] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 189–194, Monterey, Calif., Feb. 2017.

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

[17] J. C. Dvorak. *How the Itanium Killed the Computer Industry*, Jan. 2009.

[18] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, Jan. 2002.

[19] S. A. Edwards, R. Townsend, and M. A. Kim. Compositional dataflow circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 175–184, Vienna, Austria, Sept. 2017.

[20] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, first edition, 2010.

[21] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky. Speculation in elastic systems. In *Proceedings of the 46th Design Automation Conference*, pages 292–95, San Francisco, Calif., July 2009.

[22] N. George, H. Lee, D. Novo, T. Rompf, K. Brown, A. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*, pages 1–8, Munich, Sept. 2014.

[23] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011.

[24] G. Hoover and F. Brewer. Synthesizing synchronous elastic flow networks. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 306–11, Munich, Mar. 2008.

[25] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu. Elastic CGRAs. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 171–80, Monterey, Calif., Feb. 2013.

[26] L. Josipović, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):125:1–125:19, Sept. 2017.

[27] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms. Correct-by-construction microarchitectural pipelining. *Proceedings of the 27th International Conference on Computer-Aided Design*, pages 434–41, Nov. 2008.

[28] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation*, pages 318–28, Atlanta, Ga., June 1988.

[29] J. Liu, S. Bayliss, and G. A. Constantinides. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. In *Proceedings of the 23rd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 159–62, Vancouver, May 2015.

[30] The LLVM Compiler Infrastructure. *http://www.llvm.org*.

[31] Mentor Graphics. ModelSim, 2016.

[32] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–80, Apr. 1989.

[33] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen. A behavioral synthesis frontend to the Haste/TiDE design flow. In *Proceedings of the 15th International Symposium on Asynchronous Circuits and Systems*, pages 185–94, Chapel Hill, N.C., May 2009.

[34] E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. L. Lu. Automatic pipelining from transactional datapath specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–54, Mar. 2011.

[35] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st International Symposium on Computer Architecture*, pages 13–24, Minneapolis, Minn., June 2014.

[36] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 173–178, Monterey, Calif., Feb. 2017.

[37] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64, Feb. 1996.

[38] J. Sparsø. Current trends in high-level synthesis of asynchronous circuits. In *Proceedings of the 16th IEEE International Conference on Electronics, Circuits, and Systems*, pages 347–50, Yasmine Hammamet, Tunisia, Dec. 2009.

[39] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang. ElasticFlow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the 34th International Conference on Computer-Aided Design*, pages 78–85, Austin, Tex., Nov. 2015.

[40] L. Torczon and K. Cooper. *Engineering a Compiler*. Morgan Kaufmann, second edition, 2011.

[41] R. Townsend, M. A. Kim, and S. A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 76–86, Austin, TX, USA, Feb. 2017.

[42] M. Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign*, pages 171–80, Cambridge, MA, July 2009.

[43] Xilinx Inc. *Vivado High-Level Synthesis*.