From C to Elastic Circuits

Lana Josipovic*, Philip Brisk†, Paolo Ienne*
*École Polytechnique Fédérale de Lausanne, Switzerland, †University of California, Riverside, USA
lana.josipovic@epfl.ch, philip@cs.ucr.edu, paolo.ienne@epfl.ch

***Abstract*—Today's *high-level synthesis* (HLS) tools rely on static scheduling. When control and/or data dependencies that cannot be resolved at compile time are present in program code, HLS tools produce pessimistic schedules based on worst-case assumptions, which may not be the common case or may never even occur in practice. At present, the only alternative is for circuit designers to eschew HLS, and to instead implement dynamic scheduling, which stalls or slows execution when control or data hazards manifest themselves at runtime, in hardware. This paper examines these issues in detail using a histogram kernel as a case-study. Starting with an inefficiently scheduled implementation produced by Vivado HLS, we introduce a latency-insensitive control mechanism that enables reaching a point in the design space that Vivado HLS could not discover on its own. We describe a step-by-step process by which a compiler could infer a dynamically scheduled circuit, starting from a high-level C language specification.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) have recently been integrated into datacenters [23], [6], [2], packaged with processors [7], and introduced to new application domains; however, their commercial success critically depends on the issue of programmability, i.e., the ability of software application developers with little to no hardware experience to nonetheless be able to accelerate their applications [14], [13], [22]. To ease the burden on the programmer, *High-Level Synthesis* (HLS) tools can generate hardware designs from high level programming languages, providing a higher level of abstraction for accelerator design compared to traditional design methodologies using VHDL or Verilog [26], [4].

Despite commercial adoption in the last decade, HLS tools are limited in scope. For certain types of applications, namely loop nests whose memory access patterns and control dependencies can be resolved statically, HLS tools produce accelerator architectures that are competitive with manual hardware designs; however, achieving optimal design points requires peculiar code restructuring, expert user interaction, and extensive trial and error with configuration parameter settings. Moreover, modern HLS tools cannot produce good quality designs for irregular applications, whose control and data dependencies cannot be resolved statically; they produce static schedules and control mechanisms based on worst-case assumptions regarding memory and control dependencies, which significantly limits the throughput achievable by pipelining.

Using a histogram loop kernel as a case study, this paper points towards a fundamentally different form of HLS for irregular applications; a subsequent paper describes the tool itself [18]. Specifically, we demonstrate how to transform arbitrarily-written high-level code into a dynamically scheduled circuit. In doing so, we elucidate design points that traditional HLS tools, which employ static scheduling, cannot discover on their own. The ability of our approach to find these design points without requiring significant code restructuring by the programmer is likely to be extremely important in the future.

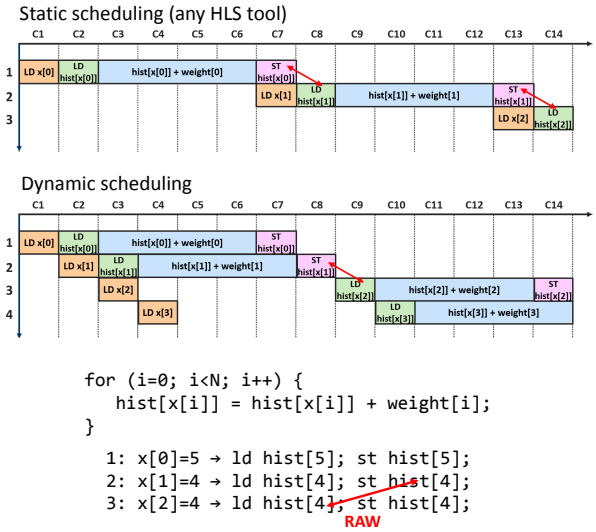
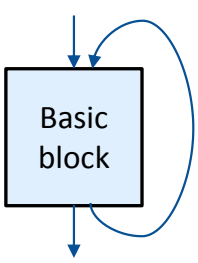


Fig. 1: A static schedule produced by a standard high-level synthesis tool, compared to a dynamic schedule generated by our approach. The HLS tool conservatively assumes that a dependency exists between each pair of loop iterations, whereas the dynamic design only stalls when dependencies occur (in this case, between iterations 2 and 3).

II. Towards Dynamic Scheduling

Modern HLS tools build datapaths that are controlled using a *pre-planned, central controller*. The controller relies on a static schedule, fixed at compile time, to determine the cycle at which each operation executes [12]. Figure 1 depicts a simple histogram kernel that illustrates the limitations of static scheduling. Within the loop, a data dependency may exist between the memory read of `hist[x[i+1]]` and the memory write to `hist[x[i]]` of the previous iteration, depending on the contents of array `x`. A static schedule must assume that the dependency occurs between each pair of consecutive iterations, irrespective of the data values fetched from memory. The resulting schedule requires the write from the previous iteration to complete before the read from the current iteration begins. This schedule is conservative, and is




```
.loop:    ; preds = .loop, .start
  i.0 = phi [ i.1, .loop ], [ 0, .start ]
  %1 = load *x, i.0
  %2 = load *weight, i.0
  %3 = load *hist, %1
  %4 = fadd  %2, %3
  store %4, *hist, %1
  i.1 = add i.0, 1
  %5 = icmp slt i.1, N
  br %5, label .loop, label .exit
```

Fig. 2: The histogram kernel from Figure 1, shown as a basic block (left), with a simplified version of the LLVM intermediate representation representing the code (right). This is the starting point for translating the kernel into a dynamically-scheduled circuit.

guaranteed to be valid for all possible input values, but is far from performance-optimal.

Figure 1 contrasts the static schedule with an example of a *dynamic schedule* in which the dependency only occurs between the second and third iterations. When the dependency does not occur, the dynamic schedule initiates a new iteration each cycle, yielding an improvement of $6\times$ compared to the static schedule; when the dependency does occur, the dynamic schedule stalls the pipeline, allowing the write to complete before the read begins. This paper describes a step-by-step process to convert the histogram kernel from Figure 1 into a dynamically scheduled circuit; a subsequent paper encapsulates this process into a novel dynamically scheduled HLS tool [18].
.

III. Elastic Control

Latency-insensitive circuit design techniques can implement dynamic schedules by replacing the pre-planned controller with a distributed control system that makes local dynamic scheduling decisions [5], [16], [8], [25]: each dynamic dependence is converted to a Boolean condition, and each operation can execute once all of its triggering conditions are satisfied.

We use *elastic circuits* [8] to implement *synchronous* dynamic scheduling in hardware. Each elastic component [15], [8] communicates with its predecessor(s) and successor(s) through a pair of handshake control signals: a *valid* signals informs the successor that the component is ready to transmit a valid piece of data, whereas the *ready* signal indicates to the predecessor that the component can now accept a new piece of data. Transferring a piece of data from one component to another is referred to as an abstract *token transfer* [21].

Figure 2 shows the LLVM compiler representation of the histogram kernel, which, essentially, consists of a single basic block. Prior to the first iteration, the iterator i is set to zero; otherwise, the loop uses the value of i calculated by the previous iteration. Within the loop body, the values of `x`, `weight`, and `hist` are loaded from memory and used to compute the new `hist` value. The iterator i is incremented by 1 and compared to the loop bound to decide the loop termination condition.

We describe a systematic process to convert a loop kernel to an elastic circuit: (1) Each loop body is a dataflow graph: we

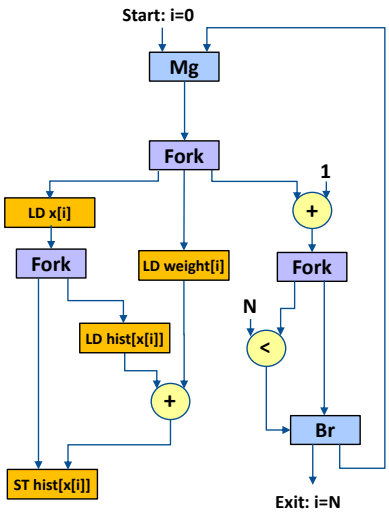


Fig. 3: The histogram kernel translated to an elastic circuit. The basic block body is converted to a dataflow circuit consisting of elastic components. The basic block inputs and outputs employ Merge and Branch nodes to propagate data across basic block boundaries, in accordance with the control flow of the kernel. Fork nodes distribute tokens from one node to multiple successors.

translate each operator into an equivalent elastic component and each dataflow edge into bidirectional handshake signals to connect dependent components. (2) For each variable that enters the loop, we allocate a Merge node to propagate the variable into the loop body. (3) For each variable that exits a the loop, we allocate a Branch node to transmit the variable to a designated successor basic block outside the loop body, triggered by the loop control condition. (4) Components that have multiple successors within the loop body require a Fork to distribute copies of its data to all of its successor components.

This strategy can convert the histogram kernel shown in Figure 2 into an elastic circuit shown in Figure 3. The iteration variable i requires Merge and Branch nodes: the Branch at the output either transmits the updated value of i back to the Merge or to the loop exit, depending on the triggering condition (i.e., the comparison of i with the loop bound N); if i is not used outside the loop, then the value can be discarded when the loop terminates.

The elastic circuit shown in Figure 3 contains two combinational cycles. We break these cycles by inserting an elastic buffer (Buff) in a portion of the circuit that is common to both cycles, as shown in Figure 4. Buffs can be inserted on *any wire* without affecting circuit functionality, since elastic control signals implicitly synchronize data communication [9].

IV. FIFO Buffers: Increasing Throughput

The cycle-free elastic circuit shown in Figure 4 is correct and fully functional, but does not offer particularly high performance. Specifically, long paths take a long time to process tokens; when a Fork distributes a data value to several independent pipelines, a slow path that cannot rapidly consume

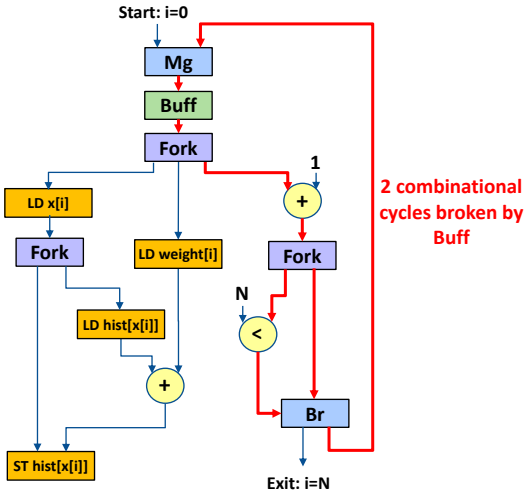


Fig. 4: Elastic buffer insertion to break combinational loops.

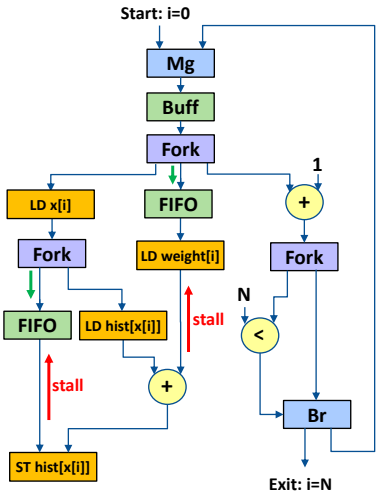


Fig. 5: Inserting FIFOs on Fork outputs can reduce backpressure caused by slow paths, resulting in increased throughput.

tokens may cause backpressure on one of the Fork outputs. This prevents the Fork from generating new tokens and starves its other successor pipelines, each of which may be ready to consume additional tokens.

In Figure 4, the store component cannot accept a token from the load operation until it receives a token from the multi-cycle floating-point adder, which will eventually create backpressure on the upper Fork. Similarly, the floating-point adder must synchronize its two input values, which may not arrive simultaneously. Consequently, the floating-point adder must stall the earlier-arriving input (i.e., the `weight` value), while waiting for the other. The backpressure imposed on the left and the middle outputs of the upper Fork prevent it from generating a token allowing the update of value i on the right-hand side of the circuit. The Fork must hold the value of i for the current iteration until all three outgoing branches have consumed their respective tokens, which significantly lowers the loop initiation interval. Absent backpressure, the updated value of i (i.e., `i++`) could be computed every clock cycle.

To increase throughput, we insert elastic FIFOs into the paths with long latencies. The tokens can accumulate in the FIFOs, which mitigates the backpressure issue, allowing the faster paths to consume Fork-generated tokens at a higher rate. Figure 5 shows the histogram circuit from Figure 4, with FIFOs selectively inserted after Forks. After FIFO insertion, a new loop iteration can start as soon as the next value of iterator i is computed, and the loop condition has been evaluated. New tokens can then be injected to all operations downstream from the Fork, effectively pipelining the design.

V. Out-of-Order Memory Interface

Thus far, we have ignored the issue of memory dependencies and the possibility of memory access reordering. In Figure 1, it is desirable to read `hist` before the write from the preceding iteration completes, if both the read and write target different memory addresses. These memory accesses should only be serialized if a dependency exists. Such behavior requires a memory interface that can correctly handle memory accesses that arrive in arbitrary order, while respecting data dependencies and ensuring appropriate ordering for semantic correctness.

We have previously demonstrated a *load-store queue* (LSQ) tailored for spatial computing architectures with dynamically scheduled circuits [17]. The novelty lies in an allocation policy that allows the LSQ to correctly handle out-of-order memory requests in a dataflow-like context. The allocation is organized into *groups* corresponding to the basic blocks of a high-level language program: whenever a basic block starts to execute, all of the memory accesses belonging to that basic block are allocated in a predetermined sequential order; within the dynamic execution, the memory accesses may be issued to the LSQ in any order. The allocation policy allows the basic block control circuitry to know precisely which allocated LSQ entry corresponds to each memory access. Accesses to provably disjoint regions of memory can be disambiguated at compile-time, and distinct LSQs can be allocated for each region.

Each iteration of the histogram application features a load and a store operation which target a data structure `hist`, stored in a unique memory; separate memories are allocated to provide accesses to `weight` and `x`, as these accesses provably do not conflict with one another or accesses to `hist`; as their access patterns are deterministic, they do not require an LSQ. Thus, a single LSQ is allocated for `hist` which features one load and one store port, corresponding to the basic block. Once the execution of the next loop iteration has been determined, a dedicated signal triggers the allocation of the read and write accesses into the LSQ (Figure 6). The LSQ exploits this information to handle out-of-order memory requests arriving from the dynamically scheduled circuit.

Figure 7 depicts our dynamically scheduled elastic histogram (left-hand-side) circuit with LSQ interface (right-hand side). The memory interface ensures that LSQ allocation is performed in the correct sequential order. The LSQ, not shown in Figure 7,

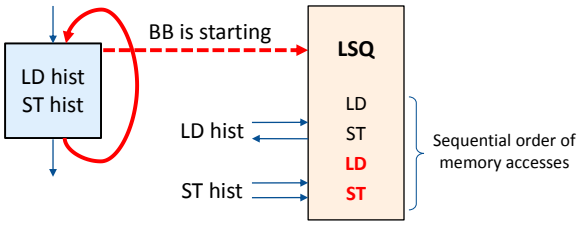


Fig. 6: An LSQ with an allocation policy designed for dynamically scheduled circuits [17]. When the program transfers control from one group (i.e., basic block) to another, the LSQ allocates all memory accesses that the group will generate. In the example from Figure 1, group allocation includes one read and one write per iteration, enabling the LSQ to handle out-of-order memory requests.

honors memory dependencies, and postpones reads of `hist` that are dependent on any not-yet-completed writes. This design realizes the dynamic schedule shown in Figure 1.

VI. Experimental Results

We compare FPGA implementations of the histogram kernel (Figure 1) using state-of-the-art static scheduling (Vivado HLS [26]) and three designs based on dynamic scheduling. The first dynamic scheduler (Huang et al. [15]) is similar in principle to elastic control, but features memory serialization (no LSQ), and employs a single Branch node [15]; the second dynamic scheduler [Elastic (LSQ)] employs elastic scheduling as described in this paper, and features an LSQ, but does not include FIFO buffers. The third dynamic scheduler [Elastic (FIFO+LSQ)] features FIFO buffers in addition to the LSQ. All four designs use identical arithmetic units and memory interfaces. Table I reports the timing (i.e., loop initiation interval, clock period, and latency), and resource requirements (i.e., FPGA slices) of these experiments.

A. Static Scheduling (Vivado HLS)

Memory dependencies between the reads and writes of `hist` cannot be disambiguated at compile time. Consequently, the static scheduler constructs a conservative and inefficient schedule, assuming that each read is dependent on the previous write. In its favor, the static schedule features the lowest critical path delay and consumes the smallest number of FPGA slices.

B. Dynamic Scheduling (Huang et al. [15])

The dynamic scheduling approach introduced by Huang et al. [15] originally targeted an elastic coarse-grained reconfigurable array—we adapt their approach for an FPGA. Huang et al. use the same elastic components as described here, but take a slightly different approach to synthesizing elastic circuits. Most importantly, they employ a single Branch node for all signals that exit a basic block, which synchronizes all output values, thereby preventing pipelining across basic block boundaries (and thus, notably, of loops); in contrast, we allocate a separate Branch node for each basic block output, which allows us to decouple the fast outputs from the slower ones. Secondly, Huang et al. do not employ an LSQ, and therefore must serialize memory accesses that cannot be disambiguated at

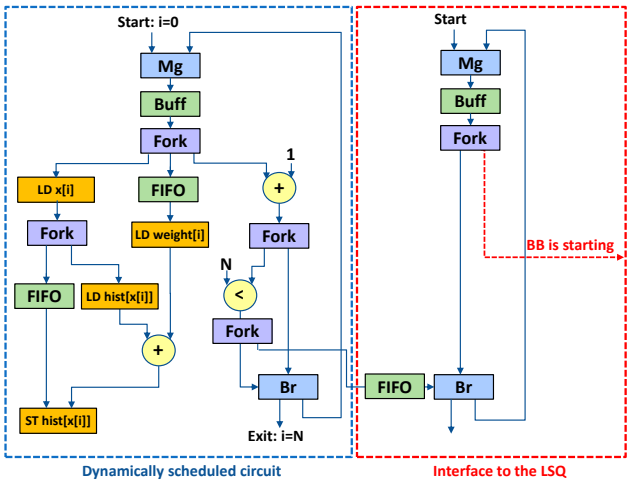


Fig. 7: A dynamically-scheduled elastic implementation of the histogram kernel (left). The LSQ interface (right) notifies the LSQ to allocate entries when control is transferred to a new basic block. The LSQ uses this information to correctly handle memory dependencies.

compile-time, which is similar to how a static scheduler would generate a modulo schedule. As shown in Table I, Huang et al. achieve a conservative initiation interval, despite the flexibility provided by dynamic scheduling, and their results are uniformly worse than Vivado HLS. Dynamic scheduling in isolation is insufficient to achieve high throughput.

C. Dynamic Scheduling [Elastic (LSQ)]

Replacing the single Branch node with multiple Branch nodes and providing an LSQ overcomes many of the bottlenecks of Huang et al.'s dynamic scheduler. Although the critical path delay increases by a factor of $1.2\times$, the II and latency are reduced by respective factors of $3.2\times$ and $2.9\times$. This timing overhead is non-negligible and is due to the critical path of the LSQ [17]; however, the increase in cycle time is conspicuously small compared to the potential improvement in II, leading to higher overall throughput. The area of the datapath is only slightly increased, but the overall area is $7.8\times$ larger due to the cost of the LSQ and its interface; this is simply the price that one must pay to benefit from memory access reordering.

D. Dynamic Scheduling [Elastic (FIFO+LSQ)]

Adding FIFOs to the design reduces both the II and latency by $1.6\times$ without increasing the critical path delay; the FIFO buffers increase the datapath area by an additional 40%.

In both of our elastic designs, the pipeline throughput dynamically varies based on the input data; when there are no dependencies, the dynamic schedule achieves the optimal II; otherwise, the pipeline stalls. In contrast, Vivado HLS and Huang et al.'s dynamic scheduler stall the pipeline for all memory accesses that the compiler cannot disambiguate.

VII. Related Work

Prior work has explored construction of dynamically-scheduled circuits using latency-insensitive control [5], [16],

Design	II_{avg}	CP (ns)	Time (μs)	Slices	LSQ Slices
Vivado HLS	11	3.3	36.3	130	0
Huang et al. [15]	12	4.9	59.3	134	0
Elastic (LSQ)	3.7	5.7	20.8	145	901
Elastic (FIFO+LSQ)	2.3	5.7	13.3	200	901

TABLE I: Experimental results for the histogram kernel: static scheduling (Vivado HLS); dynamic scheduling, limited by memory serialization and a single Branch node (Huang et al. [15]); dynamic scheduling featuring elastic control and an LSQ to provide memory access reordering [Elastic (LSQ)]; and dynamic scheduling featuring elastic control with FIFO buffers and LSQ [Elastic (FIFO+LSQ)].

[8], [25], [19]; however, they lack generic transformations to derive such circuits from high-level language specifications. Compilers from high-level language specifications to asynchronous circuits have been investigated [11], [3]; the techniques summarized here share principle similarities, but target a *perfectly synchronous implementation* which eschews many of the fundamental difficulties that have historically plagued asynchronous design styles. Recent HLS advances extended static scheduling [1], [20], [24], [10] to handle specific types of dynamic events, which limits the achievable performance improvements to specific cases; in contrast, the case study presented here is more general and resulted in a 100% dynamically-scheduled circuit.

VIII. Conclusion

Dynamic scheduling is key to accelerate irregular applications using FPGAs but is insufficient on its own. Using a simple histogram kernel, we demonstrate that dynamic memory access disambiguation and mitigating the effects of backpressure among concurrent arithmetic pipelines are key to throughput maximization. Although we generated our histogram circuits manually, opportunities to automate these transformations are clear. Future work will develop a compiler that can convert a high-level language program specification into an elastic circuit that features the additional optimizations evaluated here.

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 177–186, Austin, Tex., Mar. 2005.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24:1–24:27, Sept. 2013.

[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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] N. George, H. Lee, D. Novo, M. Owaida, D. Andrews, K. Olukotun, and P. Ienne. Automatic support for multi-module parallelism from computational patterns. In *Proceedings of the 24th International Conference on Field-Programmable Logic and Applications*, pages 1–8, London, Sept. 2015.

[14] 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.

[15] 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.

[16] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *Proceedings of the 8th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, Manchester, Apr. 2002.

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

[18] L. Josipovic, R. Ghosal, and P. Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2018. To appear.

[19] 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.

[20] 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.

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

[22] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1591–1604, Dec. 2015.

[23] 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.

[24] 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.

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

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