Speculative Dataflow Circuits

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

ABSTRACT

With FPGAs facing broader application domains, the conversion of imperative languages into dataflow circuits has been recently revamped as a way to overcome the conservatism of statically scheduled high-level synthesis. Apart from the ability to extract parallelism in irregular and control-dominated applications, dynamic scheduling opens a door to speculative execution, one of the most powerful ideas in computer architecture. Speculation allows executing certain operations before it is known whether they are correct or required: it can significantly increase fine-grain parallelism in loops where the condition takes many cycles to compute; it can also increase the performance of circuits limited by potential dependencies by assuming independence early on and by reverting to the correct execution if the prediction was wrong. In this work, we detail our methodology to enable tentative and reversible execution in dynamically scheduled dataflow circuits. We create a generic framework for handling speculation in dataflow circuits and show that our approach can achieve significant performance improvements over traditional circuit generation techniques.

ACM Reference Format:
Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative Dataflow Circuits. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293914

1 INTRODUCTION

In the realm of processors, statically scheduled processors (usually referred to as *very long instruction word* or VLIW processors) have most suffered from the inability to accommodate arbitrary forms of speculative execution: Predicated execution (committing an instruction only if a specific condition is true) can be seen as a form of speculation when used to implement *if*-conversion (two branches of an *if-then-else* statement are both executed until the value of the condition is known). Yet, even aggressive predication is not applicable to every performance-critical control decision and Intel, as part of a failed attempt to develop a competitive general-purpose VLIW architecture, had to introduce a few dedicated speculative instructions (e.g., *advanced* and *speculative* loads [13]), de facto squeezing back into a statically scheduled processor some essential dynamic behaviour. We believe that, analogously, statically scheduled circuits, such as those generated by common *high-level synthesis* (HLS) tools, cannot be competitive in some applications because of their inability to exploit broad classes of speculative execution.

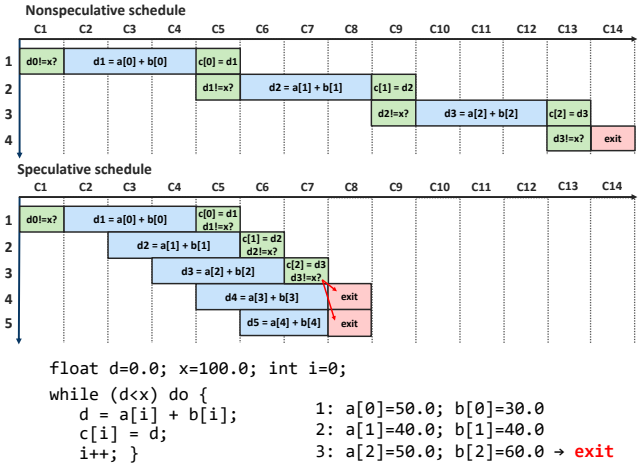


Figure 1: A nonspeculative schedule, compared to a schedule produced by a system supporting speculative behavior. The code below the schedules takes multiple clock cycles to compute the condition for executing another loop iteration. A nonspeculative circuit needs to wait for the condition, whereas the speculative circuit tentatively starts another iteration and then discards the newly computed values if they are later on determined unneeded.

On the other hand, part of the success of speculative execution in dynamically scheduled processors is the fact that a fairly limited set of universal techniques (i.e., register renaming, reordering buffers, and commit mechanisms) is sufficient to support speculation of virtually any critical decision worth predicting. Dataflow circuits are the spatial-computing equivalent of dynamically scheduled processors and can be generated by particular HLS tools; in this paper, we explore whether similarly broad classes of speculation can be easily supported in such circuits. We demonstrate that this indeed is possible, that it also needs a fairly small number of generic components and techniques, and that the advantage can be significant when waiting for a key execution decision is particularly time-consuming.

2 WHY HLS NEEDS SPECULATIVE BEHAVIOR

To illustrate the need to accommodate speculative behavior in circuits produced from imperative languages such as C, consider the code in Figure 1. A standard, nonspeculative HLS tool would not allow a new loop iteration to start until the condition to exit the loop has been checked—this condition is available only after performing almost the entire loop body, which largely prevents pipelining

FPGA '19, February 24–26, 2019, Seaside, CA, USA

ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293914

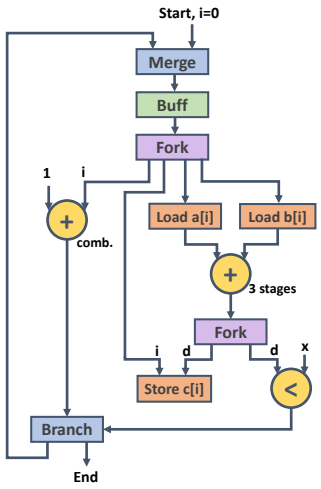


Figure 2: A dynamically scheduled circuit executing the code of Figure 1. All connections between components carry data with the corresponding bidirectional handshake signals. The Branch unit requires both the a value and a binary condition before it can issue the value it has received to the Merge unit and thus start a new iteration. Therefore, a condition which takes a long time to compute may significantly hinder performance.

of the loop. In contrast, speculation would make possible a high-throughput pipeline which tentatively starts another loop iteration on every clock cycle and, later on, discards the speculatively computed values if the loop was supposed to terminate prior to their execution (in the case of Figure 1, the addition results from the fourth and the fifth iteration are unneeded and will be discarded once this is decided by the termination check in cycle *C*7).

3 BACKGROUND ON DATAFLOW CIRCUITS

Dynamic scheduling is crucial to enable generic speculative behavior because the schedule needs to be adapted on the fly to any combination of prediction outcomes. Many dataflow or latency-insensitive protocols implement dynamically scheduled circuits, either using asynchronous or synchronous protocols; some have been used to produce circuits from imperative languages (e.g., Budiu et al. [2] and Josipović et al. [14]). In this work, without loss of generality, we choose the same handshake protocol used by Josipović et al. [14] because it leads to designs which can readily be implemented as synchronous circuits and thus directly compared to commercial tools for FPGAs. Our dataflow components communicate with their predecessors and successors using a pair of handshake signals and transfer data whenever the successor is available (passing a piece of data with the correct handshaking is colloquially referred to as exchanging a *token*). In this section, we provide an overview of the structure and properties of our circuits and describe the dataflow components that are most relevant for this work.

To produce our circuits, we follow pretty literally the methodology described by Josipović et al. [14]—and, in many respects, the exact topology is not critical for the modifications we will introduce. What matters is that our circuits are composed of subcircuits corresponding to *basic blocks* (BBs)—i.e., straight pieces of code with a single entry and single exit point. The body of each BB is a direct translation of the dataflow graph into an interconnect of dataflow components. The control flow (i.e., the interconnections between BBs) is implemented using the following components: (1) A Merge unit propagates a token received on any of its inputs to its single output and is equivalent to *phi* nodes in the static single assignment form [19]. We allocate a Merge for every variable entering a BB. (2) A Branch unit propagates a token received at its single input to one of its multiple outputs based on a condition; it is used for implementing control flow statements. We place a Branch for every value exiting a BB and used by any successor BB.

The execution of our dataflow circuits is triggered by a start token that enters the initial BB. The token propagates through the BBs, following the control flow of the program, until it eventually arrives at the final BB, representing program completion. Although individual operations inside BBs are executed out of order, BBs are triggered in exactly the same order as the software execution of the unmodified original program. This way of building dataflow circuits implies three properties essential for the correctness of our speculative circuits: (1) **One token per loop.** On a cyclic path, there can be only a single token at a time (a token enters a BB on a cycle through a Merge; as this BB determines the next control flow decision, it is the only block that can send the token backwards into the Merge—i.e., there is no other active predecessor BB that could inject another token into this path). (2) **Strict token ordering on a path.** If there are multiple tokens on an acyclic path, they could only be injected into it by repeatedly forking at every passage the single token of a cycle and the propagation of this token in the cycle is determined by the in-order control flow decision. This means that tokens are inserted into the acyclic path in a deterministic order and nothing on this path can affect their ordering. (3) **Determinism.** Although our circuits contain nondeterministic components (e.g., Merge), the strict ordering of BBs guarantees that the execution is race-free: as tokens can enter the BB only from the single active predecessor, and no other source can inject tokens into the BB, there is nothing that can interfere with the token ordering at the BB input.

Figure 2 shows the dataflow circuit corresponding to the example in Figure 1; this simple program needs a single Branch and a single Merge node for the loop iterator i. The token carrying the iterator value is propagated to the next loop iteration whenever this is decided by the Branch condition (i.e., the comparison of the value d with x). Despite the flexibility of dynamic scheduling, if the condition takes a long time to compute (as it is the case here), the Branch unit will hold the token representing $i + 1$ until the token with the condition arrives, and the start of a new loop iteration will be delayed—speculative execution (in this case branch prediction and speculation) is needed to achieve an efficient pipeline.

4 SPECULATION IN DATAFLOW CIRCUITS

Our goal is to create a generic framework for handling speculation in dataflow circuits. The idea is that some components might be allowed to issue *speculative tokens*—that is, pieces of data which might or might not prove correct and which will combine with other (nonspeculative) tokens, resulting in more speculative tokens travelling through the circuit. In other words, speculative tokens trigger some computations which might have to be squashed and possibly repeated with the correct nonspeculative tokens. Figure 3

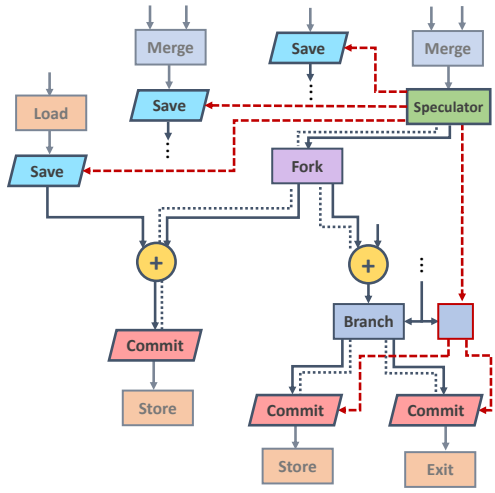


Figure 3: A region of a dataflow circuit implementing our speculative execution paradigm. The Speculator initiates the speculative execution by injecting tokens tentatively, Save units capture required inputs to the region to enable a correct replay in case of misspeculation, and Commit units prevent speculative tokens from affecting irreversibly the architectural state, such as memory. Speculative tokens are marked explicitly using an additional bit (represented by the dotted line). A dataflow control circuit (in red, dashed line) between the Speculator and the Save and Commit units carries information about speculative events (start, commit, squash, etc.).

gives a sense of our strategy: speculative tokens will be contained in a region of the circuit delimited by special components.

The first component is a *Speculator*. A Speculator is a special version of a regular dataflow component which, besides its standard functionality also has the liberty of injecting tokens before receiving any at its input(s). The most natural example is that of a Branch node which receives the value to dispatch but not the condition; a Branch Speculator could predict the missing condition and send tentatively the value through one of its outputs. If, after issuing a speculative token, the Speculator eventually receives the same data which it assumed speculatively (e.g., the condition it predicted), all is fine and execution was probably sped up; if, on the other hand, the data it eventually receives does not match the prediction, we have a case of misspeculation: the Speculator should now perform its function correctly (e.g., resend the value on the other output), but must first make sure that the speculative work done is discarded.

The reason for the output boundary of the speculative region of Figure 3 is fairly evident: clearly, speculative tokens cannot be allowed to propagate indefinitely and must not affect the architectural state of the circuit, that is the part of the state which is known and visible to the user. Therefore, the speculative region must be limited at least before components which store values in memory or before the end of the circuit. The components at the output end of the speculative region are called *Commit* units. These units simply let propagate further speculative results which turn out to be correct; much as it happens in speculative software processors, results due to misspeculation are simply squashed. Because Commit units must differentiate speculative from nonspeculative tokens (the former ones need explicit commit information before propagating, while the latter ones can always go ahead), as Figure 3 suggests, all channels between the Speculator and the Commit units must be enriched with a control signal which indicates the speculativeness of the token being passed.

Finally, we need to bound the speculative region on the input side in order to save a copy of all regular tokens which may combine with a speculative token so as to be able to reissue them if the previous computation is squashed. We call these components *Save* units.

Section 5 details these new dataflow components and Section 6 describes how to correctly place them in the circuit. An important aspect of a speculative region, i.e., the communication between the speculative components, is only sketched in Figure 3: the Speculator should communicate with the Commit and Save units whenever it starts and stops a speculative event. We have elected to implement this communication through an additional dataflow circuit connecting all the new speculative units; while this communication is relatively straightforward (essentially, binary tokens indicating whether a speculation was successful or not), there are a few peculiarities to take into account when speculative tokens traverse Merge and Branch units. We will detail the construction of this control circuitry in Section 7. A critical situation, not represented in the qualitative example of Figure 3, occurs when the Speculator is placed on a loop: we will use an intuitive but too conservative approach in Section 6, likely to result in speculative circuits with little performance advantage, and then fully tackle this fundamental problem in Section 8.

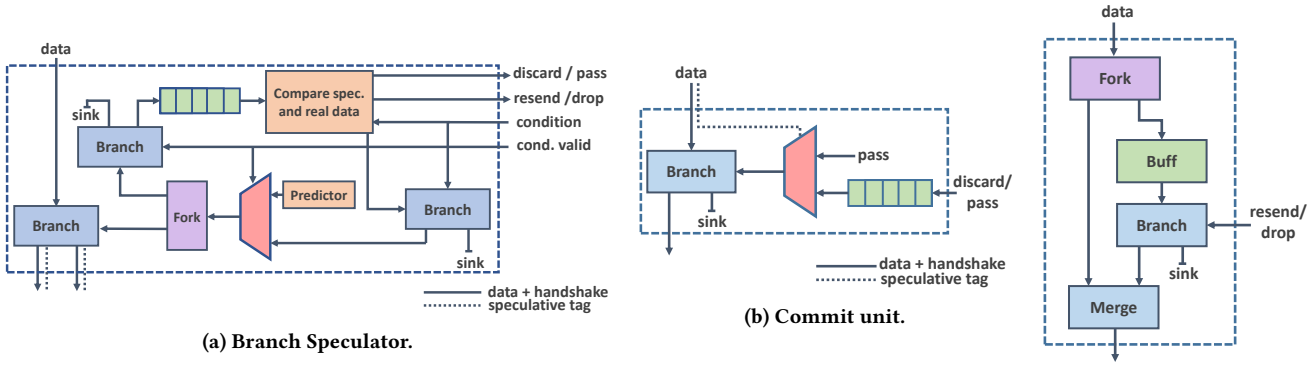
5 COMPONENTS FOR SPECULATION

This section details the components needed to delimit a speculative region in a dataflow circuit: a Speculator to initiate the process, Save units on the inputs of the region, and Commit units on its outputs. The components are, in general, built out of standard dataflow components and they communicate with the rest of the design using the handshake protocol mentioned in Section 3.

5.1 Speculators

Speculative execution starts when a *Speculator* triggers the execution of a part of the circuit before it is certain that it needs to execute or that the execution is correct. Any dataflow component can operate as a Speculator by issuing *speculative tokens* before all of the component's input information is available (i.e., when only a subset of the input tokens is available at the inputs of the component). For instance, a Speculator Branch can speculate on the condition, causing the Branch to output the data token to one of its successors before the condition token arrives; a Speculator within a load-store queue can perform a speculative load and eagerly output a speculative data token as soon as the load address is available and before all memory dependencies are resolved.

Apart from issuing speculative tokens, the Speculator's role is to determine the correctness of a speculation and trigger actions accordingly. It therefore saves the prediction and assesses the situation once the missing input arrives. As the tokens propagate through the circuit strictly in order, the first token arriving at the particular input whose value was speculated will hold the value which resolves the first speculation. After deciding if the prediction was successful, the Speculator informs the appropriate units in the



(a) Branch Speculator.

(b) Commit unit.

(c) Save unit.

Figure 4: Components for speculation. Figure 4a outlines the structure of a Branch Speculator, which can speculate on the branch condition and output a speculative data token. It later on determines the correctness of a speculation and communicates this information to the Save and Commit units. The Commit unit (Figure 4b) stalls speculative tokens until the correctness of the speculation has been determined. The Save unit of Figure 4c saves tokens that might interact with speculative ones to be able to replay the computations in case of a misspeculation.

circuit of the comparison result, allowing them to commit the speculative results or to discard the misspeculated tokens and recompute with the correct values. In the second case, the Speculator needs to insert the token holding the correct value into the circuit in order for the computations to execute anew.

The structure of one of the most natural Speculators, i.e. the Branch Speculator, is given in Figure 4a. Unlike a standard Branch, which waits for both the data and the condition to arrive before producing an output, the Branch Speculator can output a data token even when the condition is not yet present, together with a bit indicating whether the token is speculative. Eventually, it will receive a regular condition token, which it will compare with the previously speculated value (all speculatively issued values are stored in a queue within the Speculator) and send a confirmation or cancellation token to the other units. It will then either discard the real token (using the Branch subcomponent on the right of Figure 4a) or resend it into the circuit. The Speculator can issue a token only when its basic block is active, otherwise, there is no guarantee that the real token will eventually arrive to confirm or cancel the speculation (this is easy for a Branch Speculator because the arrival of a data token is a guarantee that the condition will also arrive).

Initially, we will discuss the case where only one speculative token at a time is issued into the circuit—i.e., a new speculation cannot start before the previous one has been resolved. While this is not a problem for pieces of code that do not need to repeat (i.e., speculating on a branch of an *if-else* statement), it could easily result in suboptimal performance if the Speculator is placed on a cyclic path (i.e., speculating on a loop termination condition). We will make necessary modifications to support multiple speculations from a single Speculator in Section 8.

5.2 Commit Unit

All dataflow components, apart from the Speculator, use a conservative firing rule—i.e., they produce tokens only once all of the required input operands become available. However, if one of the inputs to a dataflow component is a speculative token, the produced output token will become speculative as well—there is no guarantee that the computed value or the decision made by the component is correct until the speculation is resolved by the Speculator. In case the speculation is incorrect, the component will output incorrect data or send a token in the wrong control flow direction: at some point, this misspeculated data will need to be discarded.

To this end, we use *Commit* units that stall speculative tokens until they receive the corresponding decision from the Speculator: in case the speculation is determined correct, the speculative tokens are converted into regular tokens and passed on to the rest of the circuit; otherwise, they are discarded by this unit. Any regular token that reaches the unit is unaffected and simply propagated through.

Figure 4b outlines the structure of the Commit unit. Data enters the unit through an internal Branch; depending on the value of the speculative bit, it is either directly passed on to the successor components (in case the data is nonspeculative), or stalled until the unit receives a decision from the Speculator. Assuming that the data path from the Speculator to this unit is long, the Speculator might issue and resolve multiple speculations before the data tokens arrive at the Commit unit. Hence, the unit contains a queue to save the decisions from the Speculator if they arrive before the data. As the tokens arrive in order on both paths, the timing relations of the two paths cannot influence correctness: the Commit unit will keep the first speculative piece of data on one path until the first confirmation or cancellation on the other path becomes available, and all tokens will be correctly matched. The output of the Commit unit is always a regular nonspeculative token.

5.3 Save Unit

In case a speculation is determined incorrect, speculative tokens are discarded and speculated computations need to reexecute with the correct values. This means that each nonspeculative token which at some point interacts with a speculative token needs to be appropriately saved until the speculation is confirmed or canceled. To this end, we use *Save* units which store the last token that passed through it until the Speculator determines the correctness of the speculation. In case the Speculator indicates that the speculation was correct or that it did not speculate on the saved values, the saved tokens are not needed and can be discarded—these values

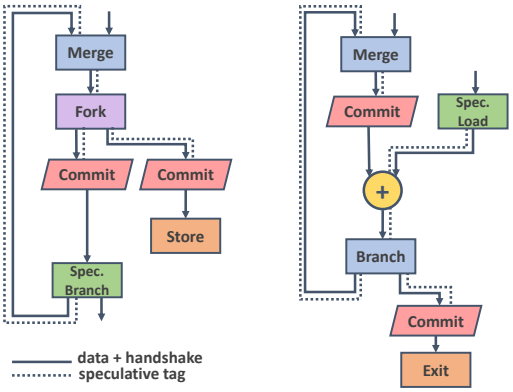


Figure 5: Placing Commit units. Our placement strategy ensures that memory is never modified by a speculative token, the program never terminates before speculation is resolved, and only nonspeculative values interact with components that might carry a speculative value.

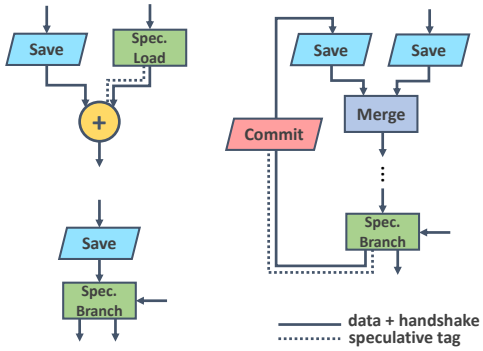


Figure 6: Placing Save units. Each token that interacts with a speculative token must be saved until the speculation is confirmed or canceled.

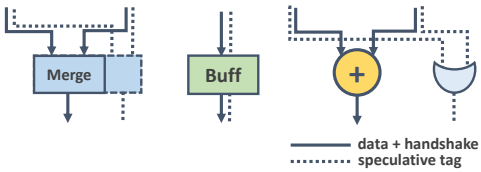


Figure 7: Extending dataflow components with a speculative tag. In most cases (e.g., Merge, Buffer, Fork), the tag is simply an additional bit propagated with the data. Components that combine multiple inputs into an output (e.g., arithmetic operations) require an OR to make the output speculative when any of the inputs is speculative.

have already been correctly propagated through the circuit and their interactions with any token issued by the Speculator produced correct results. On the other hand, if the speculation was incorrect, all Save units need to reinsert their saved token into the circuit to repeat the previously miscalculated computations.

The Save unit in Figure 4c takes a nonspeculative token as input and outputs a nonspeculative token. It requires only a single register for storing a token: for another token to arrive at the input (possible only if the unit is on a loop), the previous speculation must have been resolved and the old value inside the register has either already been reinserted into the circuit or determined unneeded and discarded through the Branch.

Note that the *discard* and *resend* outputs of the Speculator, connected to the Commit and Save units, respectively, are not equivalent: If a speculation does not occur, the Save unit still kept a token which needs to be thrown away—the Speculator must inform the unit when issuing a nonspeculative token. Commit units do not require any confirmation from the Speculator to let the nonspeculative tokens pass.

6 PLACING THE COMPONENTS

Every speculative region needs to be delimited with its own set of Commit and Save units: they ensure that misspeculated computations are appropriately squashed and replayed. This section shows where to place Commit and Save units into dataflow designs.

Every speculation needs to be resolved before terminating the program—that is, before a token reaches the Exit node. Furthermore, only regular tokens can be used for modifying memory (assuming that writes cannot be reverted) or as inputs to the Speculator (we will relax this constraint in Section 8). Therefore, we place a Commit unit on each path of the graph of dataflow components which starts at the Speculator and ends with the first of any of the following components encountered on the path: (1) an exit point of the graph; (2) the Speculator or a component carrying a speculative value; (3) a store unit. Figure 5 gives examples of correct placements of the Commit unit. Placing more than one Commit unit on a single path does not bring any benefit, as the first unit will always resolve the speculation. The Commit units should be placed as far as possible from the Speculator, as this allows speculating on more computations and therefore increases performance in case the speculation was correct.

A Save unit is required whenever a regular token can interact with a speculative one, so the operations can reexecute in the case of a misspeculation. The following paths must contain a Save unit: (1) Each path from the start of the graph of dataflow components to any component that could combine the token with a speculative value. (2) Each cyclic path containing a Speculator or any component that could combine the token with speculative values. Since these cycles contain a Commit unit (as described in the previous section), the Save unit must be placed after it—this ensures that only regular tokens enter the Save unit, as any speculation will be previously resolved. Figure 6 shows examples of placing the Save units. To maximize performance (i.e., smaller number of correct computations to reexecute in case of a misspeculation) and minimize resource requirements (i.e., smaller number of Save units required), we place the Save units as close as possible to the end of these paths (i.e., as close as possible to the paths carrying speculative tokens).

As already suggested, the dataflow circuit between a Speculator and its Commit units needs to carry data with a speculative tag. This modification requires only a minor change to standard dataflow components: it is simply one more bit of payload which is propagated or OR'ed from all inputs to make the output is speculative when any of the inputs is speculative, as depicted in Figure 7.

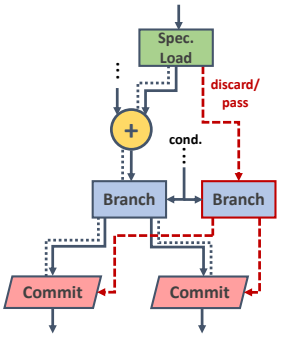


Figure 8: Connecting the Speculator to the Commit units. The cancellation or confirmation from the Speculator must be directed only to the Commit unit which is on the path of the misspeculated token. Otherwise, another token could be discarded incorrectly: if both Commit units in the figure were to receive a cancellation signal and the misspeculated token took the left branch, a correct token coming down the right branch would be eventually discarded.

7 CONNECTING THE COMPONENTS

When the Speculator determines the correctness of a speculation, it needs to inform the appropriate Save and Commit units. We add a specialized handshake network for this purpose.

7.1 Connecting the Speculator to the Commit Unit

The Speculator connects to the Commit units through a specialized network and informs them whether to discard or propagate speculative tokens. However, sending the decision to all Commit units would result in incorrect behavior. Consider the example in Figure 8: If a decision to discard the token due to a misspeculation is sent to both Commit units, and the misspeculated token takes the left output of the Branch, another token taking the right Branch output later on would be incorrectly discarded. Therefore, the information from the Speculator needs to be sent only to the units that were on the actual path taken by the speculative tokens. In such cases, we place Branches on the path connecting the Speculator and the Commit unit which receive the same conditions as the regular Branches of the dataflow circuit. Whenever a speculative token passes, the Branch in the specialized network will mimic the control flow decision took by the data token and thus correctly direct the information from the Speculator to one of the Commit units.

7.2 Connecting the Speculator to the Save Unit

The complementary problem arises when connecting the Save units—only some of them hold tokens that need to be resent to the circuit. Consider the example in Figure 9, where the Save units are placed before a Merge node. If the speculation is determined incorrect, only one of the Save units should reissue a token—however, there is nothing that can determine which of the two Save units holds the direct predecessor (i.e., which token needs to be reissued). Therefore, Merges that are on the path from the Save units to the Speculator need to remember which side a token came from. The Speculator uses this information to correctly direct the confirmation/cancellation to the proper Save unit. The dispatching is

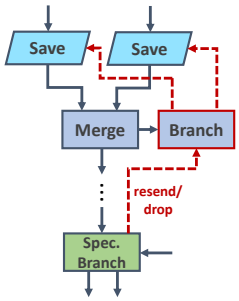


Figure 9: Connecting the Speculator to the Save units. Any Merge that is on the path from the Save units to the Speculator should memorize where tokens came from so that the Speculator can send the correct resend or discard message to the appropriate Save unit.

implemented in the specialized network as a Branch which takes the Speculator decision as data and the information from the original Merge as the condition and forwards the decision accordingly.

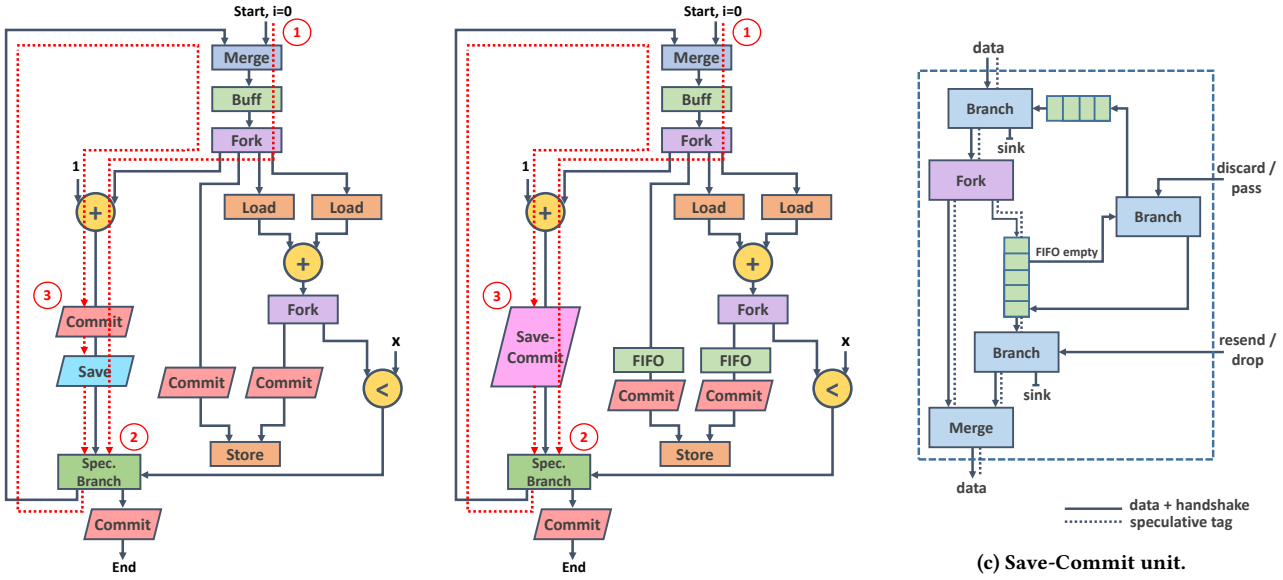
8 MULTIPLE SPECULATIONS FROM A SINGLE SPECULATOR

The approach described so far does not bring significant performance benefit when speculation occurs in a loop, as it requires us to conservatively wait for one speculation to end to be able to trigger a new one. This section discusses the modifications needed to increase loop parallelism.

8.1 Merging the Save and Commit Unit

In points where Save and Commit units meet, the approach taken so far allowed a new token to enter the Save unit only after the Commit unit sent out a confirmed token. Thus, all speculations through cyclic paths are sequentialized, which prevents us from achieving a high-throughput pipeline. Figure 10a shows the circuit from Figure 2 modified with the speculative components and the strategies described in the previous sections (note that the speculative tags are omitted for graphical simplicity). A nonspeculative token enters the Merge through the starting point (labeled as point 1 in the figure), passes through the Commit and Save unit (as it is nonspeculative) and reaches the Speculator (point 2). The Speculator issues a speculative value back through the Merge and into the Commit unit (point 3), which stalls the token until the condition reaches the Speculator and it informs the Commit unit of the correctness of the speculation—only then does the token pass through to the Speculator again, triggering the start of a new speculation.

Whenever a Save and Commit unit meet on a cyclic path, we can merge them into a single unit which allows issuing a speculative token even before the previous speculation has been resolved. The *Save-Commit unit* (Figure 10c) performs the combined functionality of both units: as a Save unit, it issues regular tokens to restart computations or discards them when they are no longer needed; as a Commit unit, it turns speculative tokens into regular ones or discards speculative tokens. However, unlike a regular Commit unit, this unit will also let speculative tokens pass to the successors; it will save all the tokens, corresponding to regular or speculated data from multiple loop iterations, until they are no longer needed. We



(a) Speculating on a single value at a time.

(b) Speculating on multiple values.

(c) Save-Commit unit.

Figure 10: Enabling multiple speculations from a single Speculator in the example from Figure 2. Our strategy from Section 5 results in suboptimal behavior when the speculation occurs in a loop: as Figure 10a shows, a token that is speculatively inserted into the loop will be stalled in the Commit unit (point 3 in the figure) until the speculation is resolved, preventing the triggering of a new speculation. Merging the Save and Commit units on the loop into a single unit (Figure 10b) allows issuing a new speculative token before the previous speculation has been resolved. The structure of the Save-Commit unit is illustrated in Figure 10c.

exploit the fact that the tokens are stored in the unit in order, as well as that the decisions arrive in order from the Speculator—this allows us to easily match every decision to a token queued in this unit. The action of reissuing or discarding token (usually performed by a Save unit) will be applied on the oldest stored token, which will then be discarded or reissued and, in both cases, removed from the unit as it is no longer required. If the Speculator informs the unit that a speculation was correct, the oldest token will be removed from the unit and its speculative successor will be transformed into a regular token. If the Speculator sends a decision to discard a misspeculated token, the oldest speculative token will be discarded. The Speculator will issue cancellations for each speculative token produced after the first misspeculation and each will discard one of the queued tokens. If the data tokens to cancel are not yet available, the cancellations are queued in a dedicated FIFO and the data is discarded as soon as it enters the unit.

Figure 10b shows the circuit of Figure 10a where the Save and Commit unit on the loop has been replaced with a combined Save-Commit unit. As before, the token enters through the Merge (point 1 in the figure) and is sent to the Speculator. The Speculator issues a speculative token (point 2), which is stored in the Save-Commit unit (point 3), but also immediately propagated to the Speculator to trigger another speculation, hence finally resulting in the high-throughput pipeline achieving the lower schedule of Figure 1.

8.2 Connecting the Speculator to the Save-Commit Unit

There are two paths connecting the Save-Commit unit and the Speculator, and both could contain control flow decisions: the one from the Save-Commit output to the Speculator could contain Merges (exactly like the path from the Save unit to the Speculator in Figure 9), and the one from the Speculator to the Save-Commit input could contain Branches (same as the paths to the Commit units depicted in Figure 8). Therefore, as in the previous cases, our network dedicated for sending decisions to this unit will have to collect the control flow information from the original circuit to ensure that decision tokens are distributed the correct way. The principle is exactly the same as for connecting the Speculator to the Save and Commit units; however, each control flow point will now have to hold multiple control flow decisions (as many as the Save-Commit unit can accommodate tokens). Whenever the Speculator sends a decision, the oldest queued condition will be used and discarded. This ensures that every unit is correctly informed of the speculation.

9 SPECULATIONS FROM MULTIPLE SPECULATORS

The methodology discussed in the previous section describes a circuit with only one Speculator issuing speculative tokens. Our approach can easily be extended to support multiple Speculators in the design. The Save and Commit units and their placement strategy would be exactly the same; the only difference is that each speculative token needs to be tagged to keep track of the speculation origin—this enables each Commit unit to properly handle speculative tokens (i.e., each Commit unit should consider as speculative only the tokens from the Speculator it is connected to; all speculative tokens of a different origin should be treated as nonspeculative).

Benchmark	Design	II	CP (ns)	Time (μs)	Speedup	Slices	LUTs	FFs	DSPs
While loop	Static	11	3.7	37.4		130	270	436	2
	Dynamic	12	4.4	48.8	0.8×	129 (-1%)	353	511	2
	Speculative	~ 1	4.8	4.5	8.3×	186 (+43%)	486	582	2
Backtrack	Static	21	3.7	76.2		175	353	625	5
	Dynamic	22	3.5	75.6	1.0×	251 (+43%)	555	859	7
	Speculative	~ 1	5.1	5.1	14.9×	320 (+82%)	774	956	7
Subdiagonal	Static	17	3.6	60.0		164	342	591	5
	Dynamic	18	3.6	64.0	0.9×	179 (+9%)	424	611	5
	Speculative	~ 1	4.6	5.1	11.8×	233 (+42%)	559	650	5
Fixed point	Static	15	3.3	3.3		187	354	573	5
	Dynamic	17	3.3	3.8	0.9×	177 (-5%)	371	581	5
	Speculative	~ 6	3.8	1.6	2.1×	198 (+6%)	477	601	5
Newton-Raphson	Static	8	5.4	4.3		201	585	636	9
	Dynamic	10	5.0	5.1	0.8×	234 (+16%)	775	498	9
	Speculative	~ 1	5.5	0.6	7.2×	348 (+73%)	1181	603	9

Table 1: Timing and resource requirements for the benchmarks from Section 10.1: static scheduling (Vivado HLS), dynamic scheduling (Josipović et al. [14]), and dynamic scheduling featuring speculation.

10 EVALUATION

We compare static and dynamic implementations of various realistic kernels. The statically-scheduled baselines (indicated as *static*) are obtained using a commercial tool (Vivado HLS [20]). We compare them with dynamic designs automatically produced from C code using the methodology described by Josipović et al. [14], which results in nonspeculative circuits like that of Figure 2; we indicate the dynamically-scheduled references as *dynamic*. Finally, we manually modify these circuits with the speculative components presented in this work to obtain circuits as in Figure 10b, which are the main result of this work and are indicated as *speculative*. Although our speculative methodology is perfectly general, in our examples we speculate on a single control flow decision using Branch Speculators from Figure 4a. The Speculators contain a static predictor that assumes the branch always taken whenever the input data becomes available. Each design contains as many Speculators as there are variables which need to be speculatively issued to the successor basic block. All designs use identical floating point and integer arithmetic units and connect to the exact same RAM interface as the baseline designs from Vivado HLS. We use simulations in ModelSim [16] for functional verification and for measuring the loop initiation intervals (*II*). We synthesize the designs with Vivado to obtain the clock period from the post-routing timing analysis and the resource usage from placing and routing the designs.

10.1 Benchmarks

The designs that we consider in this section represent typical cases which can profit by branch prediction and where speculative execution should bring significant performance benefits over conservative, static scheduling. The benchmark loops are derived from real applications which can be found in literature [18].

- *While loop* is the kernel from Figure 1. The dynamic design results in the circuit of Figure 2 which we extend with speculative components to obtain the circuit of Figure 10b.
- *Backtrack* is the inner loop of the backtracking pass of the Bellman-Dijkstra-Viterbi algorithm. After labeling each state with the minimum cost to reach it, the backtracking pass looks for a unique set of edges that produce the global minimum. The states are traversed in a *for* loop which breaks when the predecessor state with the minimum cost is found. The *break* statement prevents loop pipelining, as the static tool starts a new loop iteration only after the break condition from the previous iteration has been determined false.
- *Subdiagonal* is an inner loop of a QL algorithm for determining the eigenvalues of a tridiagonal matrix. The loop looks for a single small subdiagonal element to split the matrix and contains a conditional *break* inside the loop body to return the correct subdiagonal index. As the condition for the return takes a long time to compute, it prevents static scheduling from efficiently pipelining the loop.
- *Fixed point* is an iteration method for finding the real roots of a function. It consists of a *while* loop which iterates through a sequence of improving approximate solutions until the desired degree of accuracy is achieved. Static scheduling postpones the start of a new iteration until the error computation from the previous iteration has been completed.
- *Newton-Raphson* is a hybrid algorithm of bisection and the Newton-Raphson method for finding the roots of a function. The hybrid algorithm takes a bisection step whenever Newton-Raphson would take the solution out of bounds and therefore improves the convergence properties of the algorithm over the standard Newton-Raphson method. The algorithm contains a *for* loop with an *if-else* statement to determine which of the two methods to use for a particular data point. Static predication is limited by the complex *if* condition and, as the next loop iteration requires the data computed in the current one, it must be scheduled for after the condition has been determined.

```
int i = 0;
int s = 1;

for (i = 0; i < 12; i++){
    if (x[i]*s >= 1000)
        s+=1;
}
```

Figure 11: Code used for the analysis of Section 10.3, qualitatively similar to the Newton-Raphson benchmark.

Design	II	CP (ns)	Time (μs)	Slice	LUT	FF	DSP
Static	1	5.7	0.1	1281	2088	5311	24
Dyn.	6	4.3	0.3	65	163	156	3
Spec.	2.3	5.3	0.2	154	481	301	3

Table 2: Timing and resource requirements for the static (Vivado HLS), dynamic (Josipović et al. [14]), and speculative implementation of the loop from Figure 11. The code given to the static tool was restructured to produce an aggressively-predicated schedule.

10.2 Results

Table 1 reports the timing and resource requirements of our experiments. The static scheduler constructs a conservative schedule which prevents almost any pipelining of these loops because it waits for the condition to be determined before starting a new loop iteration. In spite of the flexibility of dynamic circuits, dynamic scheduling alone does not suffice to achieve high parallelism for the exact same reason as the static schedule does not: a new loop iteration is delayed until the previous decision has been determined—i.e., the Branch waits for the condition token to arrive before propagating a data token backwards into the loop body. In contrast, the Speculator in the final design issues speculative tokens into the loop as soon as the input data becomes available and enables achieving the ideal loop initiation interval. Note that the speculative initiation interval II_{spec} is, in fact, a weighted average of the value in case of good prediction and of that for a misprediction. For all circuits but *Newton-Raphson*, there is a single misprediction when the loop is exited and therefore the average II is for all practical purposes exactly the one in Table 1. Note that $II_{spec} \approx 1$ for all benchmarks but *Fixed point*: in this case, the input data to the Branch takes 6 cycles to compute, therefore limiting the maximum issue rate of speculative tokens. The resource increase and the longer critical path (*CP*) are due to the additional components for speculation and the FIFOs that we added to achieve maximum parallelism.

Although the table indicates $II_{spec} \approx 1$ for *Newton-Raphson*, the situation is slightly different than in the other benchmarks: in this case, the misprediction is not an event happening only once per loop execution, but every time a bisection step is taken. The actual II is therefore data dependent but still close to 1, as the bisection step is meant to be a relatively rare event. It is worth noting that our circuits do not have any additional penalty for misprediction other than incurring the longer latency of the corresponding dynamic nonspeculative circuit. Therefore, in general and to a first-order approximation (because we ignore the difference in critical path), our circuits would perform better than a static circuit whenever the prediction accuracy $p_{correct}$ is such that $p_{correct} \cdot II_{spec\_opt} + (1 - p_{correct}) \cdot II_{nonspec} < II_{static}$. To put this in perspective using this example and again ignoring the *CP* difference, our circuit needs here only $p_{correct} > 22\%$ to perform better, and this branch prediction accuracy is massively below typical achievable rates.

10.3 Analysis

It is clear from Table 1 that the use of a dynamically scheduled paradigm has a nonnegligible cost in resources (already pointed out by Josipović et al. [14]); the situation is only aggravated by the support for speculation. Although all our designs are Pareto optimal (and significantly faster than the baseline designs), it is worth looking a bit closer at such results. As suggested in Section 1, predication is the way purely static scheduling methods can implement speculation (that is, by executing in parallel every possibility and selecting the right outcome later). It is usually viable when the number of predicated branches is small; thus, it is customarily used in the textbook case of *if*-conversion where only two short branches need to be followed for a very short period and are soon resolved. If resources are not strongly limited (that is in the world of spatial computing as opposed to traditional VLIW compilation), one could explore an aggressive use of *if*-conversion where many branches are predicated at once—for example, with predication spanning multiple iterations of a loop body, as it would be required in some of our benchmarks. In this section, we want to explore how competitive our technique is against highly-speculative statically-scheduled circuits beyond what our commercial tool produces.

We study here the code of Figure 11, which is qualitatively similar to our Newton-Raphson benchmark but stripped for clarity of everything except key operations. The naive version by Vivado HLS has $II = 4$ because of the loop-carried dependence on `s` and the multiplication (latency 4) in the condition which determines the new `s` (the conditional addition is predicated and executed in parallel). It is perfectly possible to restructure the code to perform aggressive *if*-conversion across basic blocks: every iteration spawns two branches corresponding to the new *if* condition, and this for each of the existing predicated branches; on the other hand, four cycles later, the computed condition resolves pairwise all open branches and halves them, leading to a steady state of in-flight branches. Assuming that the critical latencies are 1 for the addition and 4 for the multiplication, as it is the case for the components used by Vivado HLS, achieving $II = 1$ requires 16 parallel branches which compute `s` for every combination of the *if* conditions in the last four iterations and 8 branches computing the new conditions, also in turn depending on the conditions of the last three iterations. Essentially, the needed computational resources to achieve $II = 1$ with a purely static schedule are 8 multipliers, 8 adders and 8 comparators to execute all predicated branches in parallel. Table 2 shows the comparison of the static, manually restructured code (to achieve a static schedule with $II = 1$), dynamic, and speculative design using a dataset which predicts correctly the condition in 75% of the cases. These results suggest that, although more speculation than what common HLS tools implement is possible, the cost can be very high (notice that the cost is exponential in II_{static}, which is *only* 4 in this case). Clearly, our speculative circuit is Pareto-optimal compared to the aggressively-predicated static design. The area cost is due to the fundamental inability of statically scheduled circuits to revert some arbitrary computation and recompute it from scratch; a statically scheduled solution can only evaluate all possibilities at once

and this only when the number of possible outcomes is tractable (which is not the case in a situation we have not demonstrated here but is perfectly covered by our technique—the prediction of independence trough memory of a load from all previous pending stores). On the contrary, a dynamically scheduled, speculative circuit can simply execute the single most likely path and squash and recompute mistakenly predicted outcomes. In all fairness, this also implies a worsening of the execution time when almost-perfect predictions cannot be made, like in the present example, whereas the static solution has *exactly* $II = 1$ irrespective of predictability.

11 RELATED WORK

Much as compilers for VLIW processors do, in order to extract parallelism, many HLS approaches exploit aggressive code motion techniques to anticipate the execution of some operations before it is certain [12, 15, 17]. However, the conservatism of static scheduling hinders such optimizations in the presence of complex control flow or memory accesses. The importance of speculation has not escaped HLS researchers and some have shown partial forms of speculation; albeit remarkable, their approach lacks generality in the ability to revert arbitrarily the state after failed predictions and suffers from being applied to otherwise static schedules [7].

Latency-insensitive protocols [3, 6, 9] have been explored as a way to overcome the limitations of static scheduling and offer the flexibility needed for true speculation [11]. Several latency-insensitive approaches [4, 5, 10] describe early evaluation—predicated execution based on special tokens which discard mispredicted data. However, these techniques are applicable only for standard *if*-conversion, which static HLS can handle as well, and do not cover the more general cases of speculation that we discuss in this work. Budiu et al. went further in implementing features similar to those existing in superscalar processors in their asynchronous dataflow circuits, yet they also failed to implement a generic framework for speculation due to "the difficulty of building a mechanism for squashing the computation on the wrong paths" [1]. Our scheme for discarding and replaying computations does exactly this.

Desikan et al. [8] describe a mechanism for load-store dependence speculation in the context of dataflow processors, but have also faced challenges in building a suitable speculation resolution network: their approach is based on sending the commit/discard decisions through the dataflow graph, so the traversal of these decisions delays the commits and therefore impedes performance. In contrast, we use a dedicated, fast network which enables speculative components to communicate directly and efficiently. Moreover, their speculation scheme requires version numbering and token tagging to handle out-of-order speculative bits—our circuit design strategy ensures that tokens traverse the graph in order, which simplifies our speculation mechanism.

12 CONCLUSIONS

In this work, we present a generic methodology to enable speculative execution in dataflow circuits and show that it can reap significant benefits in appropriate situations. Our simple and methodical approach to bring arbitrary forms of speculation to dataflow circuits mirrors out-of-order processors, where the same commit-or-squash-and-replay approach is at the heart of their very successful speculative mechanisms. Others have shown that dependencies through memory are an important case where dynamic schedules are highly profitable: the next logical step will be to build a speculative load-store queue which executes speculatively loads before pending and unresolved stores, as in common processors; the generality of our speculation scheme will simply work unmodified for this important situation. We believe all this to be key for FPGAs and HLS to be successful in new contexts such as datacenters, where applications will be more irregular, control-dominated, and software oriented than most FPGA applications are today.

ACKNOWLEDGMENTS

Lana Josipović is supported by a Google PhD Fellowship in Systems and Networking.

REFERENCES

[1] 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.
[2] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.
[3] 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.
[4] M. R. Casu and L. Macchiarulo. Adaptive latency insensitive protocols and elastic circuits with early evaluation: A comparative analysis. *Electronic Notes in Theoretical Computer Science*, 245:35–50, Aug. 2009.
[5] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *Proceedings of the 44th Design Automation Conference*, pages 416–19, San Diego, Calif., June 2007.
[6] 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.
[7] 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.
[8] R. Desikan, S. Sethumadhavan, D. Burger, and S. W. Keckler. Scalable selective re-execution for EDGE architectures. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 120–132, Boston, MA, Oct. 2004.
[9] 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, Sept. 2017.
[10] H. Gädke and A. Koch. Accelerating speculative execution in high-level synthesis with cancel tokens. In *International Workshop on Applied Reconfigurable Computing*, pages 185–195, Berlin, Mar. 2008. Springer.
[11] 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.
[12] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 171–176, Oct. 2001.
[13] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, Sept.–Oct. 2000.
[14] L. Josipović, R. Ghosal, and P. Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 127–36, Monterey, Calif., Feb. 2018.
[15] V. Lapotre, P. Coussy, C. Chavet, H. Wouafo, and R. Danilo. Dynamic branch prediction for high-level synthesis. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*, pages 1–6, Porto, Sept. 2013.
[16] Mentor Graphics. ModelSim, 2016.
[17] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct. 2016.
[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, third edition, 2007.
[19] L. Torczon and K. Cooper. *Engineering a Compiler*. Morgan Kaufmann, second edition, 2011.
[20] Xilinx Inc. *Vivado High-Level Synthesis*.