Toward Speculative Loop Pipelining for High-Level Synthesis

Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki

Toward Speculative Loop Pipelining for High-Level Synthesis

Steven Derrien, Thibaut Marty, Simon Rokicki, Tomofumi Yuki
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

***Abstract*—Loop pipelining is a key optimization in modern HLS tools for synthesizing efficient hardware datapaths. Existing techniques for automatic loop pipelining are limited by static analysis that cannot precisely analyze loops with data-dependent control-flow and/or memory accesses. We propose a technique for speculative loop pipelining that handles both control-flow and memory speculations in a unified manner. Our approach is entirely expressed at the source-level, allowing a seamless integration to development flows using HLS. Our evaluation shows significant improvement in throughput over standard loop pipelining.**

I. Introduction

Although FPGA accelerators benefit from excellent energy/performance characteristics, their usage is hindered by the lack of high-level programming tools. High-Level Synthesis (HLS) tools aim at making FPGAs more accessible by enabling the automatic derivation of highly optimized hardware designs, directly from high-level specifications (C, C++, OpenCL). Although HLS is now a mature technology, existing tools still suffer from many limitations, and are usually less efficient compared to manual designs by experts.

HLS tools take algorithmic specifications in the form of C/C++ as inputs, just like standard compilers. They benefit from decades of progress in program optimization and parallelization, since most of these optimizations are also relevant for deriving efficient hardware.

In practice, the usage of HLS tools resembles that of CAD tools, such as Logic/RTL synthesizers, where the design process involves a lot of interactions with the user. These interactions allow the exploration of performance and area trade-offs in the resulting hardware. They usually consist in evaluating the impact of certain key optimizations to derive the best solution given performance and/or area constraints.

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue.

Loop pipelining is one of these key transformations, as it enables the synthesis of complex yet area-efficient hardware datapaths. Current HLS tools are effective at applying loop pipelining to loops with regular control and simple memory access patterns, but struggle with data-dependent control-flow and/or memory accesses. The main reason is that existing loop pipelining techniques rely on static schedules, which cannot precisely capture data-dependent behaviors.

In this work, we address this limitation through a speculative loop pipelining framework supporting both control-flow and memory dependence speculation. More specifically, our contributions are the following:

- Speculative Loop Pipelining (SLP) for HLS, expressed entirely as source-level transformations.
- Automation of SLP in a source-to-source compiler.
- An experimental evaluation of the approach that shows significant performance improvement over standard loop pipelining.

An important strength of our work is that the speculative design is expressed entirely at the source-level. This allows our approach to be seamlessly integrated into HLS design flows providing two key benefits: (i) the pipelined datapath is synthesized by the HLS tools that are capable of deriving efficient designs, and (ii) we do not compromise on the ease-of-use aspects: programmers keep all the productivity benefits (e.g., easier/faster testing) of having high-level specifications.

The rest of this paper is organized as follows. Section II introduces loop pipelining in HLS and its current limitations. Section III details our source-level transformations to realize SLP, and Section IV describes its automation. We evaluate our work in Section V, discuss related work in Section VI, and conclude in Section VII.

II. Background and Motivation

Designing efficient hardware accelerators using HLS relies on the ability to uncover and exploit parallelism

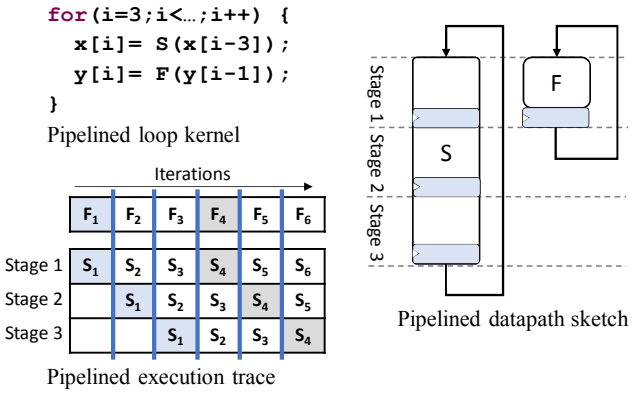


Fig. 1. A pipelined loop kernel. The function `S` is mapped to a three-stage pipelined operator and has a dependence distance of three. The function `F` is mapped to a single-stage operator, with a dependence distance of one. This gives a pipelined schedule with II=1 and Δ=3.

from the algorithm. Unlike compilers targeting CPUs or GPUs, where the target hardware is known at compile time, HLS tools enjoy more degrees of freedom. As a consequence, quantifying the effective performance of a design is more complex than for a programmable device.

A. Loop Pipelining in HLS

The performance of a hardware accelerator is determined by its achievable clock speed ($f_{\max}$), and its operation-level parallelism (i.e., the number of useful operations per cycle). HLS optimizations may affect either of these two characteristics.

Loop pipelining is one of such optimizations that consists in overlapping the execution of consecutive iterations of a loop. A pipelined loop is characterized by two metrics.

- The **pipeline latency** (Δ) corresponds to the number of pipeline stages in the synthesized datapath.
- The **Initiation Interval** (II) is the delay separating the execution of two consecutive iterations. When targeting FPGAs, designers generally aim for fully pipelined loops with II=1.

Figure 1 shows a simple kernel along with its fully pipelined execution (that is, with II=1). The value produced by `S` is said to have a *dependence distance* of three, since it is used three iterations later. If the distance is shorter than three, then the II must be increased to wait for the next iteration's input to be computed. HLS tools use compiler analyses to determine if consecutive iterations of a loop can be executed in parallel, which may not be possible due to dependences and/or resource constraints (e.g., memory ports).

Loop pipelining slightly differs from software pipelining, which targets instruction-set based architectures [1], because the latency of operations in the resulting hardware datapath can be flexibly tuned to the target clock speed specified by the designer. Therefore, the depth of the resulting pipelined schedule can be made larger to enable higher clock speeds. Because of the abundant number of registers in FPGAs, the area overhead due to pipelining is low, making it an effective transformation (datapaths with 100+ pipeline stages are not uncommon).

B. Limits of Loop Pipelining

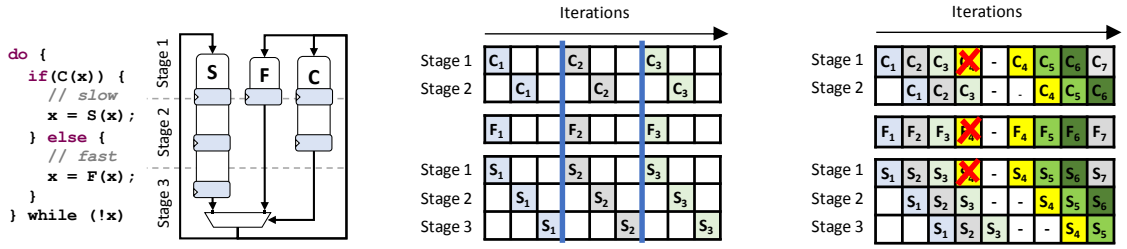
HLS tools rely on simple dependence analysis techniques, and often fail at precisely identifying how much a loop can actually be pipelined. Although there are more advanced analyses and optimizations [2]–[5], the compiler still fails at identifying pipeline opportunities in many situations. This is the case when the control flow and/or access patterns become too irregular and/or data-dependent to be amenable to any compiler analysis.

For example, consider the loop kernel shown in Figure 2a, which exposes two execution paths: a *fast* path (single cycle), and a *slow* path (three cycles). In this kernel, the actual execution path depends on the value of `x`, which is only known at runtime. Because of this, any static dependence analysis fails at precisely computing the dependences. Therefore, the compiler has no choice but to follow a conservative (i.e., worst-case) schedule, as illustrated in Figure 2b. In this schedule, the compiler conservatively assumes that the slow path is always taken, and does not expose iteration-level parallelism.

Such examples are quite common in practice, especially when dealing with emerging application domains such as graph analytics, network packet inspection, or algorithms operating on sparse data structures.

However, the execution of multiple iterations may be overlapped by speculating that the fast path is always taken, as illustrated in Figure 2c. In case of a misspeculation, operations started with incorrect values (`C(x)` and `F(x)` at the fourth iteration) must be canceled, and wait for the correct values to become available (`S(x)` at the third iteration). This approach is similar to that of a pipelined in-order CPU, where the execution of instructions is speculatively overlapped, which potentially causes pipeline hazards.

Existing work on speculative execution for HLS either addresses the problem of runtime data-dependence management [6], [7], or improve tool support for loops with complex control-flow [8]–[13]. The only exception is the work by Josipović et al. [14] that adds speculation to dataflow circuits and is capable of addressing both issues. We discuss related work further in Section VI.



(a) Loop with data-dependent control flow. (b) Static loop pipelining (II=3). (c) Speculative pipelining (average II=1.4).

Fig. 2. Comparison of standard loop pipelining and speculative loop pipelining. In Figure 2c, `C(x)` at the third iteration evaluates to `true`, causing misspeculation. Average II assumes 20% misspeculation rate.

III. Proposed Approach

Our goal is to automatically derive circuits implementing SLP from C programs. This section describes the program transformations we use to achieve this.

A. Speculative Loop Pipelining at the Source-Level

We propose to express SLP directly at the source-level, in contrast to prior work [7], [14] that operates at the HLS back-end or at the RTL-level. Our insight consists in decoupling the pipeline control logic from the pipelined datapath in the source code.

We take advantage of this decoupling as follows:

- We increase the dependence distance exposed in the datapath by speculating over the outcome of a conditional. This extra slack allows the HLS tool to derive more aggressively pipelined designs.
- We enhance the control logic to monitor the loop execution and handle misspeculations.

A key characteristic of this source-level transformation is that it does not directly perform operation-level pipelining. Instead, this task is delegated to the HLS loop pipelining optimization. This approach has many benefits:

- It simplifies code generation and testing, preserves the benefits of static pipelining (low control overhead, ability to share resources when targeting II>1, etc.) while benefiting from the performance improvements offered by a speculative schedule.
- Since the misspeculation detection logic is directly weaved into the code, cycle-accurate performance numbers can be obtained simply by executing the program and tracking misspeculation overhead.

Figure 3 shows an example of our source-level implementation of SLP. The transformation takes estimated pipeline latency of each path as inputs: $\Delta_C = 2$, $\Delta_S = 5$, and $\Delta_F = \Delta_H = 1$. The resulting code exposes a shorter critical path length (Δ_F) to the HLS tools, enabling fully pipelined schedules with II=1, by speculatively using outputs of `F`.

In this transformed source code, loop iterations correspond to cycles in the eventual HW, and no longer correspond to the original loop iterations. The functions operate over circular buffers (see ❶), and the dependence distances over these buffers bound the pipeline depth. For example, `S` uses `s_x[t-5]`, and hence `S` may take up to five cycles to produce `mis_x[t]`. Since the dependence analyses in HLS tools are sometimes too conservative, we use annotations[1] to override their analysis when necessary.

The control is delegated to a Finite State Machine (see ❸) whose state is updated every iteration. The role of each state is as follows:

- The `FILL` state is the pipeline startup phase.
- The `RUN` state is the speculative pipeline steady-state. During this stage, the correctly speculated values are committed to their corresponding variables. The FSM remains in the `RUN` state until a misspeculation is detected, in which case the state transits to the `STALL` state.
- The `STALL` state is used during misspeculation recovery. It is used as a wait state until the outcome of the recovery path is ready. When this is the case, the FSM switches to the `ROLLB` state.
- The `ROLLB` state is the last stage of the misspeculation recovery process. In this state, the recovered state is committed and then the pipeline is restarted by looping back to the `FILL` state.

The *rollback* mechanism (see ❹) is implemented in the datapath as a conditional statement. The speculated value is replaced by the value computed in the recovery path. The *commit* step (see ❺) is implemented as a collection of guarded assignments to the original variables. The committed value for a speculated variable is selected

[1]Our example uses Vivado HLS annotation style to override distances for each buffer (see ❷).

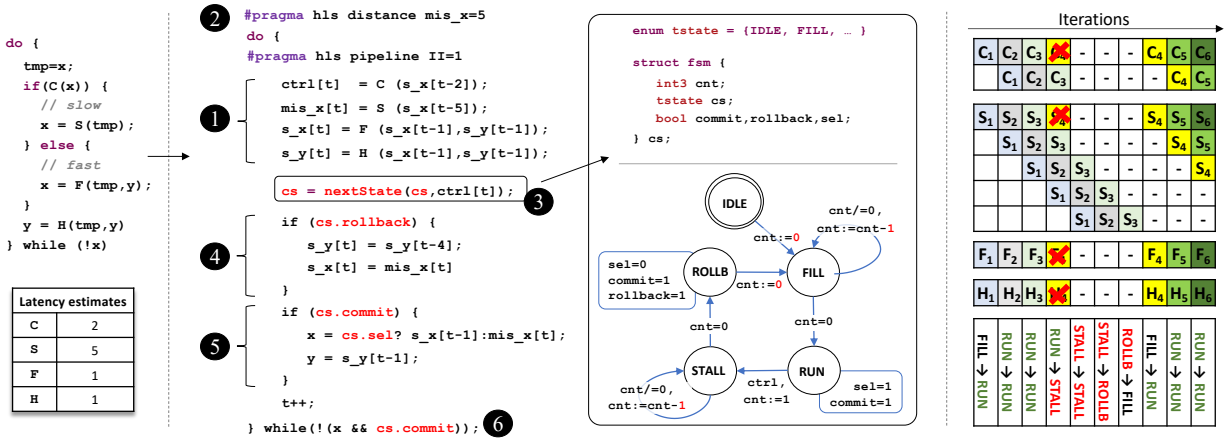


Fig. 3. An example of SLP implemented at the source-level. The figure shows, from left to right, (a) the original code, (b) the code after SLP, (c) the FSM invoked in the code, and (d) the pipelined schedule with SLP. The prefix `s_` denotes speculative values. There are a few simplifications: The FSM is represented as a state diagram, but is expressed as a `switch/case` statement in the C code. Arrays of infinite length are used for `mis_x`, `s_x`, `s_y`, and `ctrl`, where the actual implementation uses circular buffers (using modulo addressing) whose depths correspond to the reuse distances. Initializations of the circular buffers before entering the loop are omitted.

from either the speculated path (when speculation is successful), or the recovery path (when speculation is unsuccessful). In the former case, the commit uses a previously computed result (`s_x[t-1]` in this case) instead of `s_x[t]` to compensate for the pipeline latency for computing `ctrl[t]`, which is scheduled at stage 2 in our example.

B. The Parameters of the Recovery Mechanism

The recovery mechanism is implemented as extra control logic in the FSM, and circular buffers to keep track of history of values produced by the datapath.

The legality of the transformation may also be split into two: data and control. The datapath (❶) does not affect correctness, since it is the control logic that dictates which values are committed. The estimated latency of each function is used as the dependence distance to give the HLS tools sufficient slack to pipeline the datapath.

The control-path (❸ through ❻) is responsible for providing the right data to the datapath, and committing correct values in order. The duration of recovery states in the FSM, as well as the rollback distances, are computed based on a schedule of all operations in the loop body as illustrated in Figure 3 (rightmost). The `FILL` state waits until both `C` and `F` are ready: one cycle in this example. The pipeline is stalled until the slow path is done computing, after the speculation is detected, and hence it is the difference between latency of `S` and `C`: three cycle, which is split between `STALL` and `ROLLB` as described above. During rollback, all values that were computed with incorrectly speculated values must be

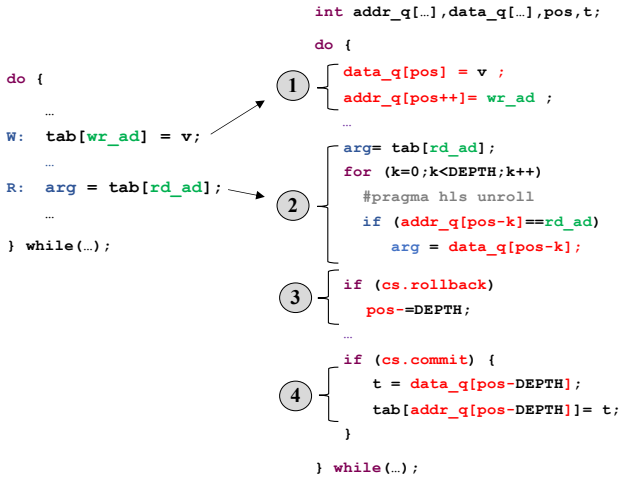


Fig. 4. Rollback on arrays using a store buffer mechanism implemented at the C-level. We use arrays of infinite length to simplify the presentation. The actual implementation uses shift registers.

reverted. In this example, `x` is replaced by the output of `S`, and outputs of `H` are rolled back by four cycles, or $\Delta_S - \Delta_H$: the difference between when `H` and `S` have finished computing.

C. Managing Arrays

In this section, we discuss how our approach can be extended to support array accesses. The main difficulty is associated with the need to rollback from incorrectly speculated operations. This can easily be achieved for scalar variables by recording their previous values using

shift registers. However, recording the entire state of arrays is not practical.

We alleviate this issue through a mechanism similar to *store buffers* used in speculative processors, which we implement at the source-level as depicted in Figure 4. This mechanism delays write operations to the memory bank (array) through a queue of pending store operations (`data_q` and `addr_q`; see ①).

Whenever a read operation is issued to the array, it is first matched against pending writes (see ②). We make sure matching is performed in parallel by fully unrolling the comparison loop. In case of a match, the most recent write value is returned. Otherwise, the read is forwarded to the memory bank. This ensures that read always access the most recent array state. It is also possible to read data from any of the k last states of the array, by selecting the set of pending writes matched during the read operation.

A rollback to the d^{th} previous array state is implemented by dropping the d most recent pending writes in the queue (see ③). If a state is not invalidated by the end of the queue, the value is written to the memory. However, this mechanism may incur significant area overhead, and the depth of the buffer (i.e., the number of pending writes) must be kept to a minimum.

D. Supporting Memory Speculation

Loop bodies with irregular control flow tend to have data-dependent memory access patterns as well. Because they are not amenable to static analysis, these patterns limit the ability to statically pipeline the loop. Prior work has addressed this limitation through memory speculation: speculate that the dependence distances are sufficiently large to overlap consecutive iterations [6], [7], [12]–[14]. These approaches use runtime checks to determine the actual dependence distance to detect and to recover from memory dependence hazards. We use the approach by Alle et al. [6] to implement memory speculation at the source-level. The main difference from their work is in the automation—we support both control-flow and memory speculation in a single framework—discussed in Section IV.

The rest of this section illustrates their approach through an example in Figure 5. The original loop (left) has two statements (`R` and `W`) that access the array `tab`, creating RAW dependences. The dependence distance cannot be analyzed at compile-time, because of data-dependent indexing functions (`H` and `G`). Thus, a static scheduler must conservatively assume that the dependence distance is one. Consider the case when the main operation (`E+W`) is mapped to a two-stage pipelined

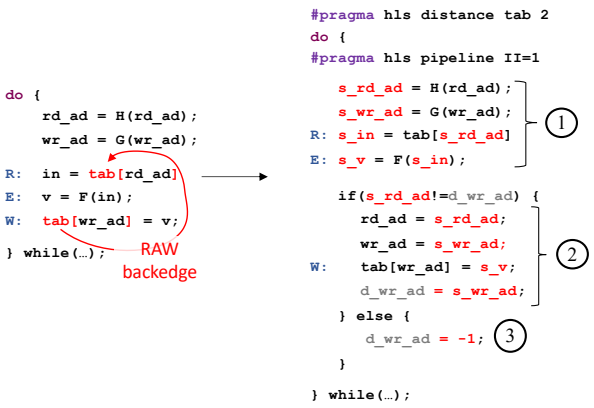


Fig. 5. A loop with data-dependent memory dependence (left), and its speculative equivalent (right), assuming one-iteration write latency.

operator. Then the II must be at least two because of the conservatively assumed dependence distance. This II may be reduced by speculating that the dependence distance is at least two.

The approach by Alle et al. [6] transforms the loop into the one depicted in Figure 5 (right). In this example, the memory write latency is assumed to be one iteration: writes to the `tab[]` array are correctly reflected to reads two or more iterations later. (One iteration eventually translates to one cycle when II=1.) This transformation consists in several steps:

- The loop body is executed speculatively by reading possibly incorrect data from array `tab[]` (see ①). The prefix `s_` denotes speculative values.
- If no address aliasing occurred, the speculation is valid, and speculative values are committed (see ②). This includes the update of array `tab[]`. When read and write addresses alias, speculative values are simply discarded.
- The pending write address (`d_wr_ad`) either stores the address of an on-going commit (`wr_ad`), or an invalid address (-1) in case of misspeculation (see ③). This address is used in the next iteration to test for aliasing.

Note that in the transformed code, loop iterations do not correspond to the iterations in the original code. The code implements a stall by skipping the commit and re-computing in the next iteration. The condition evaluates to true in the second try because (`wr_ad`) is set to -1 in the first (failed) try.

In this modified code, the reuse distance for the RAW dependence is extended to two iterations, offering additional slack to the pipeline scheduler.

The technique can be generalized to speculate over longer distances (longer read/write latency). Then, the

current read address must be matched against a history of pending write addresses. This is equivalent to a Load Store Queue (LSQ) structure used in speculative Out-of-Order processors. In this case, our implementation becomes similar to that described in Figure 4.

E. Profitability of SLP

The profitability of SLP is a function of the relative area overhead and the misspeculation rate. The effective II increases with higher rates of misspeculation, and at some point the speedup becomes uninteresting with respect to the area overhead of SLP. The effectiveness of SLP for a given loop is determined by comparing the effective II and achievable clock speed against the area overhead caused by speculation. The effective II can be obtained through source-level simulation or computed assuming a certain misspeculation rate. Area and frequency estimates can only be obtained after the HLS synthesis step.

In addition, there are some characteristics of loops that favor SLP, which may be used to further guide the selection of target loops:

- Relatively small number of arrays (in memory with low latency) to avoid excessive cost in LSQs.
- Complex conditions and/or additional computations using the results of fast/slow paths. These computations must be performed regardless of the outcome of the condition (speculation) and takes up its own space. This reduces the relative cost of SLP.

In other words, when the loop body is simple, the relative overhead of SLP becomes large, requiring low misspeculation rates to be effective.

IV. Automation in a Compiler

In this section, we describe how the transformations described in Section III can be automated. We envision our approach to be part of the interactive development flow using HLS tools, where the use of SLP becomes another performance tuning knob for the developer. Thus the focus is on how to transform the code in a systematic manner in a compiler.

A. Program representation

We use a representation based on Gated-SSA [15]. Gated-SSA is an extension of standard SSA [16] where Φ nodes are replaced by either μ or γ nodes, and where arrays are considered as values and updated through α operations. The semantics of these operations are:

- $\mu(x_{\text{out}}, x_{\text{in}})$ nodes appear at loop headers and select the initial x_{out} and loop-carried x_{in} values of a given variable x.

```
int x,rad,awd,v;
int tab[…];
do {
   if (C(x)) {
      x = S(x);
   } else {
      x = F(x);
   }
   rad = RA(x);
   wad = WA(x);
   v = tab[rad];
   tab[wad]=H(v)
} while (!x)
```

Data-dependent control-flow

Data-dependent array indexing

Data-dependent exit condition

Fig. 6. Illustrative C code snippet exposing both control-flow and memory dependence speculation.

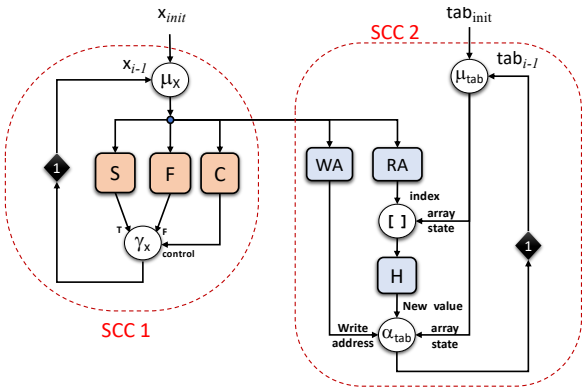


Fig. 7. Gated-SSA program representation. Diamonds identify back-edges. The number inside a diamond indicates the minimum dependence distance associated to the dependence.

- $\gamma(c, x_{\text{false}}, x_{\text{true}})$ nodes appear at joint points of a conditional, and act as multiplexers determining which definition reaches the confluence node.
- $\alpha(a, i, v)$ are used for array assignment. A new array "value" is computed from the previous value, a, by replacing the i^{th} element with v. This allows arrays to be considered as atomic objects.

Figure 7 shows the Gated-SSA representation of the program in Figure 6. In this representation, the conditional updates of `x` are captured by the γ_{x} node, and the updates to array `tab` are represented by the α_{tab} node. The loop-carried dependences involving `x` and `tab` are modeled with the μ nodes (μ_{x} and μ_{tab}) and their corresponding back-edges tagged with the minimum reuse distance inferred by the compiler (in this case, a distance of one).

We extend Gated-SSA with an additional node type: **Rollback nodes** are used to model rollbacks to previous iterations. A rollback node uses two inputs (data d and control c). When $c = 0$, the rollback node forwards its most recent data value to the output. When $c > 0$, the

node discards c most recent values and forwards the value stored c iterations ago. During code generation, rollback nodes are lowered either as circular buffers (for scalars) or store buffers (for arrays) as explained in Section III-C.

B. Identifying Speculative Execution Points

In the Gated-SSA representation, conditional branches are modeled as γ nodes. Implementing control-flow speculation amounts to speculating the outcome of a γ node by choosing one of its input data before control data is known. The following criteria define conditionals where speculation is relevant:

- The γ node must be part of a critical loop whose delay prevents pipelining. Otherwise speculation does not improve II.
- The execution paths exposed by the conditional must expose a *fast* and a *slow* path as in Figure 7. If paths are balanced, speculation is useless.

The length (in terms of pipeline stages) of each path is obtained through an analysis based on predefined delays of operators. The delays between pairs of operations in the SCC are computed, assuming no resource constraints, which is part of the computation of recurrence-constrained minimum II (RecII) in iterative modulo scheduling [1]. The delay from the μ node to immediate predecessors of the γ node gives the path latency estimates.

Note that the accuracy of this analysis does not affect correctness. The path latency estimates are used to identify fast/slow patterns and as slacks given to HLS tools in the form of dependence distances. Inaccuracy in the estimates leads to excessive/insufficient slacks, affecting the performance of the design: excessive slack increases misspeculation penalty and insufficient slack constraints loop pipelining and may result in designs with larger IIs and/or slower clock speeds.

In addition, profitability of SLP should be determined as described in Section III-E to decide if SLP is applied or not. In this paper, we assume that the conditionals to speculate are given as programmer input.

C. Constructing the Recovery Logic

We also use latency estimates as described above to compute the parameters of the recovery logic. One difference is that only the dependence edge corresponding to the fast path is considered for the speculative γ node to account for speculation. In addition, we make the following assumptions:

- The latency of the fastest path is one; the latency estimates are normalized to satisfy this assumption. The HLS tools are capable of pipelining the resulting code with larger II if desired.
- The transformation to expose memory speculation (discussed in Section IV-E) is already applied.
- The γ nodes to speculate are provided by the user as annotations.

The operations corresponding to the (end of) fast, slow, and control paths are called `F`, `S`, and `C`, respectively. The delay between the μ node and when the result of an operation `X` becomes available is denoted as θ_{X}.

All the parameters of the recovery logic are computed based on the *relative difference* between when the output of each operation is produced with respect to a couple of key time stamps:

- θ_{validate}: when the validation of speculation happens. This is the max of θ_{C}, θ_{F}, and θ for all other operations that depend on `F`. When the validation succeeds, all values computed with the speculative value are committed.
- θ_{rollback}: when the rollback in case of misspeculation happens. This is the max of θ_{S}, and θ for all other operations that depend on `S`.

Then, the parameters used in the recovery logic are:

- Duration of `FILL`: $\theta_{\text{validate}} - 1$
- Duration of stall (`STALL+ROLLB`): $\theta_{\text{rollback}} - \theta_{\text{validate}}$
- Rollback distance for operation `X`: $\theta_{\text{rollback}} - \theta_{\mathrm{X}}$
- Commit distance for operation `X`: $\theta_{\text{validate}} - \theta_{\mathrm{X}}$

The rollback distance also becomes the size of the buffers used to store speculative values, or the store buffers in case of arrays. Note that the buffers are only necessary for live-out values, and arrays with write accesses within the SCC.

D. Transformation on Gated-SSA Representation

Our speculative pipelining transformation is implemented as a set of simple structural modifications to the Gated-SSA IR, following the principles depicted in Figure 3. These changes include:

- Modifying the distance associated to all input edges of the γ node to expose dependence distances matching the expected number of pipeline stages along each input path.
- Creating **shadow variables** for every speculated live-out variables, with their corresponding multiplexer nodes.
- Instantiating (and wiring) the FSM implementing the misspeculation recovery logic.
- Creating an additional path exporting the commit values out of the SCC.
- Inserting **rollback nodes** along the back-edges of all non speculated live out variables.

We then generate the HLS C code implementing the selected speculation mechanism.

E. Memory Dependence Speculation

As explained in Section III, memory speculation can be expressed directly at the source-level. We show how this speculation mechanism can be exposed as a control-flow speculation opportunity.

Figure 8 shows a transformed loop IR starting from our earlier example (Figures 6 and 7). An additional path involving array `tab` is exposed after the transformation. This path delays the array value by one iteration—a delay is added to the edge from μ_y to $\gamma_{\texttt{tab}}$—and is selected as the outcome of node γ_{tab} when the RAW dependence distance is guaranteed to be greater than one.

This modification exposes a new cycle involving γ_{tab}, in which the cumulative dependence distance is two, enabling a two-stage pipelined implementation for function `H`. From there, we can simply take advantage of control-flow speculation to select the fast path as a speculation target.

Memory dependence speculation should only be used when the dependence distance is data-dependent. Otherwise, memory speculation does not help reduce the critical path length. We assume that the memory accesses to apply memory dependence speculation are annotated by the user.

F. Managing Concurrent Execution of SCCs

In the Gated-SSA representation, a loop involving inter-iteration dependences manifests as a collection of Strongly Connected Components (SCCs), each of them involving one or more μ nodes. This is illustrated in Figure 7 where the loop IR consists of two distinct SCCs.

Since loop-carried dependences only manifest within an SCC, we can safely apply speculative loop pipelining separately for each SCC. This leads to multiple concurrently running SCCs with their own FSMs, where some of the SCCs may be running speculatively.

We apply the principle of decoupled software pipelining [17] to allow these SCCs to run without synchronizations. In decoupled software pipelining, SCCs synchronize with each other through FIFO channels that are used to exchange data. An SCC halts when there is no input available, or when the output buffers are full. In our case, this means that each speculated SCC operates independently and communicates through FIFOs.

In some cases, we also need to rollback and access a previously consumed data token from an external input node outside of the current SCC. We solve this problem by inserting *rollback* nodes to all incoming edges of a speculated SCC, as illustrated in Figure 8.

The absence of deadlocks in our approach is guaranteed by two important properties:

- The graph formed by all the SCCs is by definition acyclic (cycles being embedded within SCCs).
- Within an SCC, all the values corresponding to an iteration of the loop are written to the FIFOs at the same time, once all values are ready (that is, in the commit stage for SCCs with speculation).

The FIFOs between SCCs therefore act as buffers to absorb variability in latency caused by speculation. Note that all SCCs involved should have similar effective II. Otherwise, the SCC with the largest II would become the bottleneck, slowing down all other SCCs.

G. Multiple Speculations

As explained in the above, our approach naturally handles multiple speculations in different SCCs. However, it is also possible to have multiple conditionals in a single SCC that benefit from speculation. These speculations may be intertwined, making scheduling more challenging. In particular, the outcome of a speculated γ node may depend on speculated values. This is illustrated in Figure 9, where γ_{y} indirectly depends on γ_{x}.

In our approach, we handle multiple speculations in a single SCC by viewing them as a single speculation. In other words, we delay the resolution of all speculated γ nodes until the last one is ready (at stage 3 in our example). When a misspeculation is detected, the pipeline is rolled back by multiple iterations, corresponding to the maximum number of speculated γ nodes in the path (two iterations in our example). This approach greatly simplifies the control logic, as misspeculation detection is performed at a single pipeline stage, and the rollback distance is a constant relative to this stage.

We acknowledge that this approach increases misspeculation penalty, when compared to a more fine-grained resolution of speculations. There are many subtle trade-offs on how to manage multiple speculations, exposing an interesting design space of its own. We leave further exploration of this design space as future work.

V. Evaluation

In this section, we evaluate speculative designs produced by our approach. We first evaluate the resulting designs in terms of their characteristics (frequency, II, and area cost) compared with fully static designs. Then, we discuss the final performance assuming different misspeculation rates.

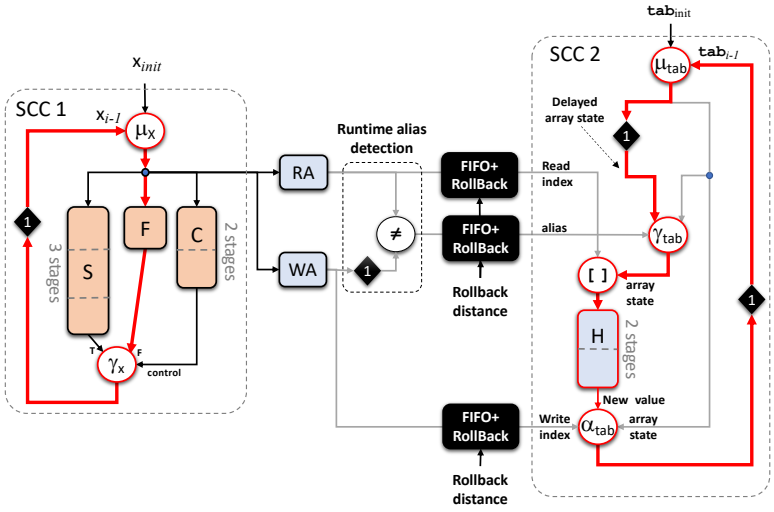


Fig. 8. Transformed IR in which we exposed memory speculation through an additional path exposing a dependence distance of 2. Since speculated paths belong to distinct SCCs, we must insert a FIFO channel with rollback capabilities between them.

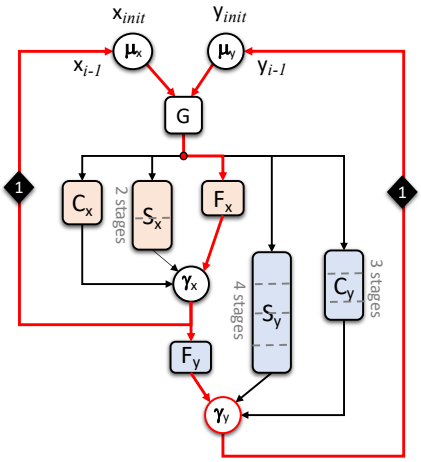


Fig. 9. Gated SSA IR exposing a chain of speculation within the same SCC. Misspeculation may occur for γ_x or γ_y.

For our experiments, we used Xilinx Vivado HLS 2019.1 as a back-end and used a Zynq xc7z020clg484-3 as the target FPGA.

A. Benchmarks

We use the following benchmarks in our evaluation:

- **Ex-Simple:** our first motivating example depicted in Figure 2a.
- **Ex-Rollback:** our example requiring a rollback mechanism, as depicted in Figure 3.
- **Ex-Memory:** an example requiring a store queue to rollback on an array, as explained in Section III-C.
- **CPU:** a simple Instruction Set-Simulator (ISS) for a toy CPU with four instructions: memory load/store, jump, and add, operating on floating-point data. This benchmark aims at demonstrating that our approach can automatically infer an in-order pipelined micro-architecture (with its corresponding hazard detection logic) directly from an ISS specification.
- **Newton-Raphson:** A root-finding algorithm, which was used by Josipović et al. in their work on speculative dataflow circuits [14]. We used the code made available[2] by the authors.
- **Histogram:** Weighted histogram that accumulates floating-point weights in the corresponding bins. We use a sparse variant used by Cheng et al. [18] in their work on combining dynamically and statically scheduled circuits. This benchmark is an example of speculation on a memory dependence.
- **gSum, gSumIf, and getTanh(double):** benchmarks used by Cheng et al. [18]. All of these codes have a simple data-dependent condition that branches to an inexpensive branch (no computation or default value) or an expensive branch with a large number of floating-point operations. We used the code made available[3] by the authors.

B. Area Overhead

Table I summarizes the HW characteristics of Speculative Loop Pipelined designs (SLP) and a baseline design with standard loop pipelining (LP). There are multiple sources of area overhead:

- Cost of reduced II. Improving the II directly increases area cost, simply because of increased parallelism. This overhead is proportional to the complexity of the datapath. When there is a long chain of floating-point operations in the slow path (e.g., gSum and gSumIf), DSP usage is multiplied by large factors correlated with the reduction in II.
- Cost of (Load) Store buffers. These buffers are necessary for programs that have speculative writes to arrays, increasing overhead on some benchmarks (Ex-Array, and Histogram).
- When the design has multiple SCCs we added a FIFO array for synchronizing the two loops,

[2]https://github.com/lana555/dynamatic/

[3]https://github.com/JianyiCheng/HLS-benchmarks/

TABLE I
HW CHARACTERISTICS OF DESIGNS WITH STANDARD LOOP PIPELINING (LP) AND SPECULATIVE LOOP PIPELINED DESIGNS (SLP).

Benchmark	II		$f_{\max}$		LUTs		FFs		SRLs		DSPs		BRAMs	
	LP	SLP	LP	SLP	LP	SLP	LP	SLP	LP	SLP	LP	SLP	LP	SLP
Ex-Simple	3	1	143	137	157	442	119	358	0	0	3	3	0	0
Ex-Rollback	5	1	123	124	335	743	179	581	0	0	6	6	0	0
Ex-Array	6	1	136	130	351	725	393	840	0	0	9	9	0	0
CPU	6	1	128	124	1063	1541	1146	1749	0	0	2	2	0	0
Newton-Raphson	6	1	148	153	502	802	383	956	0	51	9	9	0	0
Histogram	6	1	143	133	695	2198	483	1869	0	66	3	3	0	4
gSum	4	1	111	108	2391	6707	2819	5715	67	237	28	79	0	9
gSumIf	4	1	111	107	4152	11202	4382	8746	34	240	45	152	0	9
getTanh(double)	26	1	130	110	3533	2540	1336	2996	4	143	3	50	0	2

increasing the number of BRAMs used. This is the case for gSum and gSumIf.

- Rollback and delay mechanisms should ideally be implemented using shift-registers, which are either mapped to SRLs or to FFs (the final decision is up to Vivado P&R tool). However, Vivado HLS may fail at inferring precise reuse distances. In such a case, we model our delay lines using circular buffers with modulo indexing. This can lead the tool to infer BRAM primitives even for very short delay lines, as it is the case for our Ex-Simple example.
- Other components of the misspeculation mechanism, such as FSM and rollback mechanisms of scalars. These take a small part of the overhead.

Overall, the area overhead is small compared to the theoretical maximum speedup with no speculation, which is equal to the II of the fully static design because we reduce II to 1 in all benchmarks.

C. Effective Throughput

Table II summarizes the impact on performance assuming fixed misspeculation rates. The penalty of misspeculation is `STALL` + `FILL` cycles. The speedup is computed as $\frac{\text{Baseline-II}}{\text{Effective-II}} \times \frac{\text{SLP-}f_{\max}}{\text{LP-}f_{\max}}$.

The benefit of SLP is most visible for the CPU and Newton-Raphson benchmarks where the relative area cost of SLP is low. SLP has relatively low area overhead for these benchmarks, because they satisfy the characteristics discussed in Section III-E, making them ideal targets for SLP. Other benchmarks are not ideal targets due to the datapath being too simple (Histogram, gSum, gSumIf) or having a large portion of the datapath only used in case of misspeculation (getTanh(double)). These benchmarks require low misspeculation rates for SLP to be effective.

VI. DISCUSSION AND RELATED WORK

Table III provides a qualitative comparison of related work on the topic of speculative execution for HLS.

Our work on speculative pipelining has some connections with earlier work on automatic syntheses of pipelined instruction set processor architectures [19], [20]. These approaches operate from a high-level description of the target architecture specified in a Domain Specific Language. Our work is much more general as it operates from a general-purpose language and requires much less user intervention.

Holtmann and Ernst [10] were the first to discuss speculative loop pipelining in the context of HLS. They identified the need for rollback registers and proposed a scheduling algorithm, but did not evaluate the area overhead of their approach, nor address memory access aliasing. This severely limits the applicability of their work. Gupta et al. [8] and Lapotre et al. [9] developed techniques to perform control-flow speculation—branch prediction and speculative code motion—in HLS. However, they do not speculate over multiple loop iterations as we do. Alle et al. [6] and Dai et al. [7] proposed mechanisms to allow pipelining of loops with dynamic memory dependences. Our work unifies both control-speculation and memory-speculation using a single framework.

Speculative Dataflow Circuits (SDC) [14] apply speculation in the context of dataflow execution. They add mechanisms into dataflow circuits to allow nodes to execute with speculated data, and to rollback when necessary. They use LSQs to support memory accesses during speculative execution. Our work has similar goals, but we focus on loops and target mostly static designs in contrast to fully dynamic designs in their work. Our code transformations expose optimization opportunities to the HLS tools by decoupling the control logic from the datapaths. This allows the HLS tools to perform static

TABLE II
EFFECTIVE INITIATION INTERVALS AND SPEEDUPS FOR SPECULATIVE DESIGNS ASSUMING DIFFERENT MISSPECULATION RATES.

Benchmark	Pipeline/FSM Parameters			Effective II given Misspec. Rate			Speedup given Misspec. Rate			Normalized Area Cost		
	Δ	STALL	FILL	1%	10%	30%	1%	10%	30%	LUT	FF	DSP
Ex-Simple	3	2	1	1.0	1.3	1.9	2.87	2.21	1.51	2.82	3.01	1.00
Ex-Rollback	5	4	1	1.0	1.5	2.5	5.04	3.36	2,02	2.21	3.25	1.00
Ex-Array	6	5	1	1.1	1.6	2.8	5.41	3.58	2.05	2.07	2.14	1.00
CPU	5	0	4	1.0	1.4	2.2	5.59	4.15	2.64	1.45	1.53	1.00
Newton-Raphson	8	0	8	1.1	1.8	3.4	5.74	3.45	1.82	1.60	2.50	1.00
Histogram	12	0	8	1.1	1.8	3.4	5.17	3.10	1.64	3.16	3.87	1.00
gSum	7	4	1	1.1	1.5	2.5	3.71	2.59	1.56	2.81	2.03	2.82
gSumIf	7	4	1	1.1	1.5	2.5	3.67	2.57	1.54	2.70	2.00	3.38
getTanh(double)	32	26	4	1.3	4.0	10.0	16.92	5.50	2.20	0.72	2.24	16.67

TABLE III
COMPARISON BETWEEN PREVIOUSLY PROPOSED APPROACHES ON SPECULATIVE EXECUTION FOR HLS.

previous work	loop pipelining	control flow speculation	memory access disambiguisation	resource sharing	source-level
Holtmann and Ernst [10]	yes	yes	no	yes	no
Gupta et al. [8]	no	yes	no	yes	no
Lapotre et al. [9]	no	yes	no	yes	no
Alle et al. [6]	yes	no	partial	yes	yes
Dai et al. [7]	yes	no	yes (LSQ)	yes	no
Josipovic et al. [14]	yes	yes	yes (LSQ)	no	no
Our Approach	yes	yes	yes (LSQ)	yes	yes

pipelining that has many benefits over fully dynamic circuits, including resource sharing.

In addition, our designs are expressed completely at the source level, unlike some of these works that use external HDL components. This gives an important practical advantage to developers by allowing testing/verification to be performed at the source level.

Dynamic and Static Scheduling (DSS) [18] is closely related to our work, although their approach does not perform speculation. The main idea is to separately schedule different parts of the code that do not benefit from dynamic scheduling, and only perform dataflow execution among these statically scheduled components. Their work targets programs that have patterns similar to ours (branch into fast and slow paths) and further optimize area by reducing the II for a path that is taken infrequently. When the condition is simple, their approach achieves small II, since the correct path to take can be determined almost immediately. In contrast, our approach starts the next iteration without waiting for the evaluation of the condition to finish, achieving better II on programs that have long conditions, such as Newton-Raphson or CPU in our benchmarks.

Table IV compares the area cost of our work with the most recent related work [14], [18]. Note that we took the published values for the other work, but the experiment setups are not exactly the same. We target the same FPGA as DSS [18].

SLP achieves similar or better performance with respect to DSS in the benchmarks we used. This is partly because they assumed that the slower path is always taken when scheduling, which gives the most compact design but higher II. Our approach speculates the opposite: the fast path is always taken, which minimizes II but uses more area. For these benchmarks, we believe that the DSS approach gives similar results to our work if they assume that the fast path is always taken. However, this is only true for these benchmarks that have inexpensive conditions. Our work is fundamentally different in that we speculate, and thus is able to achieve shorter II when the condition is complex.

Our approach gives similar area and performance to SDC for the Newton-Raphson benchmark. Since both designs achieve II=1, there is little room for resource sharing, and hence the differences are mostly in the control logic. We did not include other benchmarks used in their work because they all speculate on the exit condition of a loop. Although our approach handles such programs—our approach also executes invalid iterations at the end of the loop, which are not committed—we focus on programs with data-dependent branches in the loop body for our evaluation.

TABLE IV
COMPARISON WITH DYNAMIC AND STATIC SCHEDULING (DSS) [18], AND SPECULATIVE DATAFLOW CIRCUITS (SDC) [14].

Benchmark	II		Fmax		LUTs		FFs		DSPs	
	SDC	SLP	SDC	SLP	SDC	SLP	SDC	SLP	SDC	SLP
Newton-Raphson	1	1	181.8	153	1181	802	603	956	9	9
Benchmark	DSS	SLP	DSS	SLP	DSS	SLP	DSS	SLP	DSS	SLP
Histogram	1	1	111.4	133	990	2198	809	1869	3	3
gSum	5	1	84.9	108	4514	6707	3960	5715	23	79
gSumIf	5	1	85	107	5222	11202	5188	8746	37	152
getTanh(double)	1	1	111.4	110	2579	2540	2797	2996	50	50

In our opinion, the main benefit of SLP with respect to dataflow based approaches [13], [14] is that it builds on a classical HLS framework (centralized control through a single FSM, static pipelining). This makes its integration in existing HLS tools much easier than for dynamic dataflow based approaches, which follow a radically different execution model.

VII. CONCLUSION

In this work, we propose a complete flow to support speculative loop pipelining in the context of High Level Synthesis. Our approach supports both control-flow and memory access speculation and is implemented as a source-to-source transformation. Experimental results show that speculation can bring significant performance improvement for several important HLS kernels, while allowing seamless integration within existing tools.

REFERENCES

[1] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27, 1994, p. 63–74.

[2] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, Sep. 2018.

[3] A. Morvan, S. Derrien, and P. Quinton, "Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 3, pp. 339–352, 2013.

[4] A. Cilardo and L. Gallo, "Improving multibank memory access parallelism with lattice-based partitioning," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, pp. 45:1–45:25, Jan. 2015.

[5] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14, 2014, pp. 199–208.

[6] 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*, ser. DAC '13, 2013.

[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 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, 2017, pp. 189–194.

[8] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the 16th IEEE International Conference on VLSI Design*, 2003, pp. 461–466.

[9] 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*, Sep. 2013, pp. 1–6.

[10] U. Holtmann and R. Ernst, "Combining MBP-speculative computation and loop pipelining in high-level synthesis," in *Proceedings the European Design and Test Conference. ED TC 1995*, March 1995, pp. 550–556.

[11] D. Herrmann and R. Ernst, "Register synthesis for speculative computation," in *Proceedings of the 1997 European conference on Design and Test*. IEEE Computer Society, 1997, p. 463.

[12] 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 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 78–85.

[13] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, 2018, p. 127–136.

[14] L. Josipović, A. Guerrieri, and P. Ienne, "Speculative dataflow circuits," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, 2019, p. 162–171.

[15] P. Tu and D. Padua, "Gated SSA-based demand-driven symbolic analysis for parallelizing compilers," in *Proceedings of the 9th International Conference on Supercomputing*, ser. ICS '95, 1995, p. 414–423.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

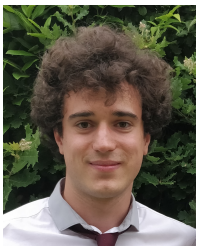
[17] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '04, 2004, pp. 177–188.

[18] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20, 2020, p. 288–298.

[19] R. J. Cloutier and D. E. Thomas, "Synthesis of pipelined instruction set processors," in *Proceedings of the 30th international Design Automation Conference*, 1993, pp. 583–588.

[20] I.-J. Huang and A. M. Despain, "Hardware/software resolution of pipeline hazards in pipeline synthesis of instruction set processors," in *Proceedings of 1993 IEEE International Conference on Computer Aided Design*, ser. ICCAD '93, 1993, pp. 594–599.

Steven Derrien obtained his PhD from University of Rennes 1 in 2003, and is now professor at University of Rennes 1. He is also a member of the Cairn research group at IRISA. His research interests include High-Level Synthesis, loop parallelization, and reconfigurable systems design.



Thibaut Marty obtained his BSc degree and MSc degree from University of Rennes, France, in 2015 and 2017 respectively. He is currently working toward the PhD degree in computer sciences at the University of Rennes, under supervision of Steven Derrien and Tomofumi Yuki. His research interests include High-Level Synthesis and fault tolerance on FPGAs.

Simon Rokicki obtained his PhD degree in computer sciences at the University of Rennes, under the supervision of Steven Derrien and Erven Rohou. He is currently a post-doctoral researcher at ENS Rennes. His research interests include embedded systems architecture, dynamic compilation, HW/SW co-design and High-Level Synthesis.

Tomofumi Yuki received his Ph.D. in computer science from Colorado State University in 2012. He is currently a researcher at Inria, France. His main research interests are in parallel/high performance computing, compilers/programming models for parallel/HPC, and High-Level Synthesis.