2026 Design, Automation & Test in Europe Conference (DATE 2026)

Area Efficient Speculative Loop Pipelining for High-Level Synthesis

Dylan Leothaud*, Simon Rokicki*, Steven Derrien†, Isabelle Puaut*
*Univ Rennes, Inria, CNRS, IRISA
†Université de Bretagne Occidentale, Lab-STICC

***Abstract*—High-Level Synthesis (HLS) allows the automatic generation of efficient circuit designs for computation-intensive kernels, but it lacks flexibility when dealing with irregular control flow. Dynamic and speculative HLS techniques are used to address this issue. These techniques outperform state-of-the-art HLS in kernel execution times but introduce a significant area overhead. In contrast, state-of-the-art HLS easily highlights and exploits resource-sharing opportunities. In this work, we show how to adapt an existing speculative HLS approach to take advantage of well-known static resource sharing mechanisms. Our results show a decrease of the area cost by 34% on average.**

I. Introduction

Classical High-Level Synthesis (HLS) flows provide a convenient way to generate hardware from high-level code. But their performance is fundamentally constrained by the fact that they rely solely on static dependence information. The literature explores two main directions. *Dynamic Scheduling* leverages runtime information to adapt the execution of operations and thereby extract more parallelism than static analysis alone can guarantee [1]–[3]. *Speculative scheduling* goes one step further: it allows the compiler to assume that certain dependences or hazards will not occur optimistically, and to provide recovery mechanisms in case these assumptions are violated [4]–[6]. Both directions enable the generation of higher-performance circuits than with classical HLS, but typically at the cost of increased area.

The *SpecHLS*[1] open-source HLS flow used in this work follows this line of work, but rather than starting from scratch, extends the loop pipelining techniques available in commercial static HLS tools to support forms of speculative and dynamic execution, enabling more aggressive pipelines [5]–[7]. SpecHLS can handle both control-flow and data/memory hazards, but ends up over-provisioning resources to ensure that the pipeline maintains its maximum throughput in the worst-case scenario.

One of the strengths of SpecHLS is its reliance on classical static pipelining techniques. The idea of this work is to use the same principle to resource constraints, which are already well managed in static scheduling. By doing so, we address these issues and thereby reduce the area overhead of speculative HLS, while preserving its performance benefits. More specifically, the contributions of this work are the following:

- We introduce a new technique called *unpipelining* for enabling resource sharing in speculative loop pipelining.
- We introduce two optimizations: *speculative resource sharing*, and *speculative array partitioning*, that build on this new technique.
- We demonstrate the benefits of our approach over a set of representative kernels experimentally.

The remainder of the paper is organized as follows. Section II provides background on pipelining algorithms and resource sharing in HLS flows. Section III motivates and presents the proposed technique. Section IV presents the experimental results. Related work is presented in Section V.

[1]https://lotr.irisa.fr/

II. Background

This section recalls key concepts relevant to scheduling under resource constraints in the context of High-Level Synthesis.

A. Scheduling in High-Level Synthesis

HLS flows rely on parallelism and customization to achieve high performance. From a compiler perspective, *identifying* and *exploiting* parallelism is challenging. Notably, *loop pipelining* [8] has proven to be a critical transformation in HLS due to its ability to automatically generate deep and wide pipelined datapaths, even in the presence of loop-carried dependencies.

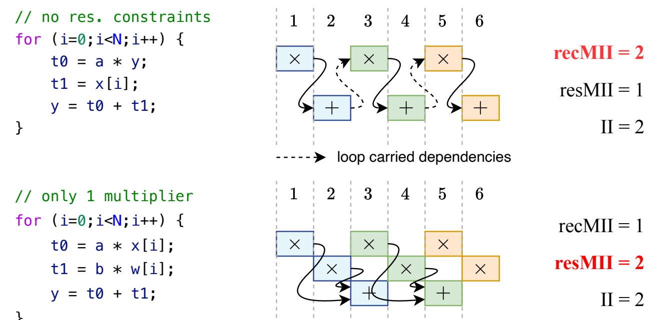


Figure 1. Static schedule of simple examples. The data dependencies constrain the Initiation Interval (II) of the first example. Resource constraints constrain the second II; only one multiplication can be done at once.

A pipelined schedule is characterized by its Initiation Interval (II) and its depth (D). The II represents the number of cycles between the start of two consecutive loop iterations in a pipelined schedule. It determines the throughput of the loop: the smaller the II, the more iterations can overlap. HLS compilers usually target a fully pipelined execution with $II = 1$, where all resources are used at every cycle. However, the achievable II is bounded by two theoretical limits. The recurrence-constrained minimum II (recMII) captures the effect of loop-

978-3-9826741-1-7/DATE26/© 2026 EDAA

carried dependencies. If an operation depends on results from earlier iterations, the schedule cannot initiate new iterations faster than this dependency allows. In contrast, the resource-constrained minimum II (resMII) reflects hardware limitations. If only a limited number of functional units are available (e.g., multipliers or memory ports), the schedule must be stretched to avoid conflicts. The effective II of a loop is at least the maximum of recMII and resMII. The two examples of Figure 1 illustrate each of these two limits.

Many kernels cannot fully benefit from classical loop pipelining, often because of unpredictable control flow or memory access patterns, which negatively impact the value of recMII determined by the compiler. Speculative Loop Pipelining (SLP) techniques enable more efficient implementations by anticipating the outcome of conditionals and memory accesses, and help improve recMII.

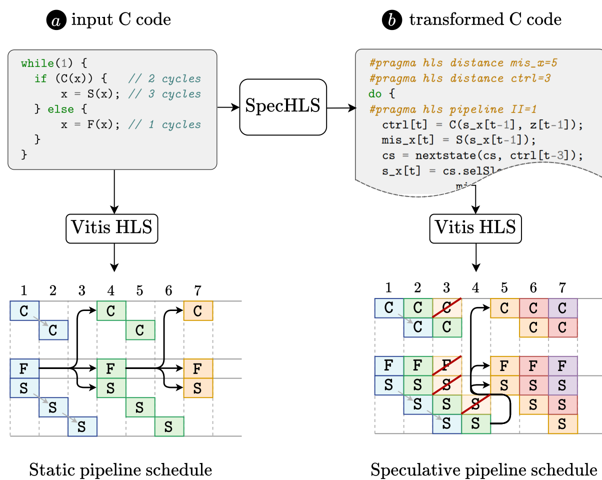


Figure 2. Overview of Speculative Loop Pipelining (SLP) as a source-level transformation in the SpecHLS flow.

This work extends the open-source SpecHLS flow. The SpecHLS flow enables SLP by extending existing HLS toolchains through source-level transformations to support both dynamic and speculative pipelined execution [5], [6]. Figure 2 illustrates the flow on a simple loop where the execution path of an iteration depends on the outcome of a data-dependent decision. In such cases, state-of-the-art HLS tools assume the worst-case scenario, and derive a schedule whose II is limited by the value of $\mathrm{recMII} = 3$, as illustrated in part *a*. In contrast, SpecHLS produces optimized code that takes advantage of speculation by considering that the fast path `F` path is always taken, leading to a design with $\mathrm{recMII} = 1$. When a mispeculation occurs, the pending operations are cancelled and the execution restarts from a valid state, as illustrated in part *b*. Generalizing this transformation and automating it within a compiler framework requires solving several problems, including (i) determining where and when to speculate [9], [10], (ii) supporting hazard management for multiple concurrent speculations [6], and (iii) extending SLP to memory dependence speculations [7].

As with most speculative execution techniques, the performance gains achieved by SpecHLS come at the cost of significant area overhead. The root cause is that in a speculative pipelined schedule, all operations within the loop body are issued — even when their results are unlikely to be needed. In practice, this leads to the overutilization of many resources (e.g., multipliers, memory read ports). Worse, for the resources that cannot be arbitrarily duplicated (e.g., memory write ports), this can limit the achievable II, defeating the initial purpose of speculation. Being able to manage resource constraints in the context of SLP efficiently is, therefore, an issue that needs to be addressed.

B. Managing resource constraints in HLS

Efficient resource sharing is a key challenge in HLS and is at the heart of the scheduling algorithms used in HLS backends. This resource-sharing step can be performed jointly or independently from scheduling.

Most HLS tools address resource sharing during the scheduling phase by solving a resource-constrained scheduling optimization problem. Although the problem is known to be NP-complete, many efficient heuristics have been proposed [11], [12]. In practice, state-of-the-art tools [12] only support resource sharing for so-called *expensive* operators (e.g., floating-point units, DSP blocks, memory ports), and tend to favor speed over area. For example, loop pipelining in Vitis HLS tries to achieve $\mathrm{II} = 1$ by default, and therefore maps each operation of the loop body to its own resource by duplicating operators in the datapath. Some resources, however, cannot be duplicated arbitrarily, and achieving $\mathrm{II} = 1$ is therefore not always possible (the bound being imposed by resMII instead of recMII). This is the case for memory ports (typical embedded memory allows at most two accesses per cycle). This problem can be partially alleviated by some form of redundancy (for read ports), or through specific memory partitioning techniques [13].

Resource sharing can also be implemented after scheduling: when two operations are mutually exclusive, they can safely share the same hardware resource. However, guaranteeing this exclusiveness can be very challenging, especially in the context of complex concurrent schedules. Even then, determining the best sharing strategy is a non-trivial problem, as implementing these sharing mechanisms incurs additional interconnect overhead (multiplexers, arbiters). This problem, known as datapath merging, has also been well studied [14], [15]

III. Proposed Approach

The goal of this work is to exploit resource sharing in speculative loop pipelining to satisfy resource constraints and reduce circuit area, while preserving performance. We follow the principle of making the common case fast [16]: resources are allocated according to their typical utilization, while we speculate that no resource conflicts will occur.

Specifically, we construct a pipelined schedule assuming a likely execution path that achieves II = 1. If a misspeculation occurs, it is treated as a hazard: the pipeline is stalled, and the remaining operations are executed sequentially according to a predefined static schedule. Because this fallback sequence is fully determined at compile time, no dynamic arbitration is

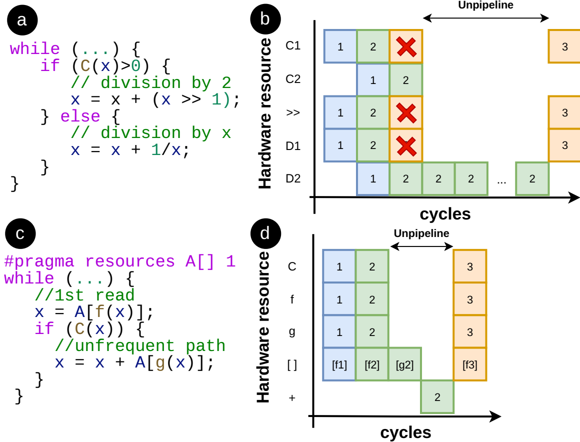


Figure 3. Example of applications that benefit from the proposed approach. The leftmost part represents the two C codes used as the entry of the HLS flow. The rightmost part represents the desired schedule.

needed. Moreover, since the pipeline and fallback executions are mutually exclusive, they can be co-scheduled onto the same functional units by the static HLS scheduler. Throughout the rest of the paper, we refer to this recovery mechanism as *unpipeling*.

The remaining Sections present examples to motivate the approach, describe the SpecHLS flow, and finally give a technical description of the different proposed transformations (*unpipelining* and its two optimizations) integrated in the SpecHLS flow.

A. Motivational examples

Figure 3 illustrates this idea on two motivational examples.

The first example (Figure 3.a) shows a simple loop that accumulates either the result of a shift operation (division by two) or a general division implemented using successive compare/subtract steps. Conventional SLP techniques relax inter-iteration dependencies by speculating that the general division will not be executed. However, they still require a *fully pipelined* division unit, which significantly increases area cost. The schedule in Figure 3.b shows the desired behavior of our approach. The third iteration (shown in orange) starts one cycle after the second, initially assuming the shift operation is selected. When the condition is resolved, it reveals a *misprediction* (we consider here that the condition needs two cycles C_1 and C_2 to be computed).

At this point, the proposed approach differs from the existing SLP techniques, which are based on classical pipelining techniques. Indeed, as stated before, pipelining a loop with $\mathrm{II} = 1$ implies that every operation is executed at every cycle, requiring a dedicated hardware resource. Consequently, SLP instantiates 30 new hardware components for computing the steps of the general division. On the contrary, the proposed idea is to remove those operations from the pipeline (i.e., *unpipelining*) to benefit from existing resource-constrained pipelining algorithms. The resulting datapath needs less hardware for a similar behavior. In the pipeline trace depicted in Figure 3.b, we can observe that during the unpipelining, the whole pipeline is stalled and the different division steps are executed *sequentially* on the same hardware resource. Notably, the initial stages of the division may have already been partially executed speculatively; only the remaining steps are serialized.

The second example (Figure 3.c) involves a loop where updating the variable `x` may require either one or two memory accesses. In this example, the current SLP transformation issues two memory reads at every iteration, requiring two read ports on that array, even if the second read is unlikely to happen. If only one read port is available, SpecHLS will actually lead to a $\mathrm{II} = 2$. In this case, the idea of *speculative resource sharing* is to speculate that only one read port is needed, and *stall* the pipeline in case of a resource hazard, to perform the missing/pending read access. This is illustrated in Figure 3.d, in which the second iteration (in green) raises such a resource hazard. The read access and its dependent operations have been unpipelined and are executed sequentially.

B. The SpecHLS flow and its intermediate representation

The SpecHLS flow uses a variant of the Gated-SSA representation (GSSA) [17], [18] as its core program Intermediate Representation (IR). In this representation, the φ-nodes of traditional SSA form [19] are replaced with fully-predicated μ and γ-nodes:

- μ-nodes are placed in loop headers and select between the initial value of a variable, and its value from the previous iteration.
- γ-nodes are placed at join points in the control-flow graph, such as the end of conditional structures. They encode a predicate, determining which value to select when execution reaches them.

Figure 4.1 depicts the GSSA representation of the division examples from Figure 3. We can see the μ-nodes for the `x` variable, and the γ-node resulting from the `if` block in the C code. The γ-node uses a predicate which takes two cycles to compute (i.e., C_1 and C_2) and selects a value which is either fast (input 0) or slow (i.e., D_1, ..., D_{32} on input 1). This representation can be derived from the corresponding C code, with timing information to determine the execution latency of the different functions called.

In the SpecHLS flow, γ-nodes play a central role: each γ-node with unbalanced inputs is a speculation opportunity. The first step in SpecHLS, after building the IR, is to determine where to speculate. This exploration step searches for the best combination of speculation, and is discussed in more detail in Section III-E. On the example depicted in Figure 4, a speculation configuration leading to $\mathrm{II} = 1$ involves speculating on input 0 of the γ-node.

C. Scheduling with unpipelining

The proposed technique involves several steps, illustrated in Figure 4 for the division example.

1) **Slicing:** the goal is to determine the set of operations that can be unpipelined. This corresponds to operations that only impact non-speculative inputs of γ-nodes. Precisely, this step is implemented by unconnecting the non-

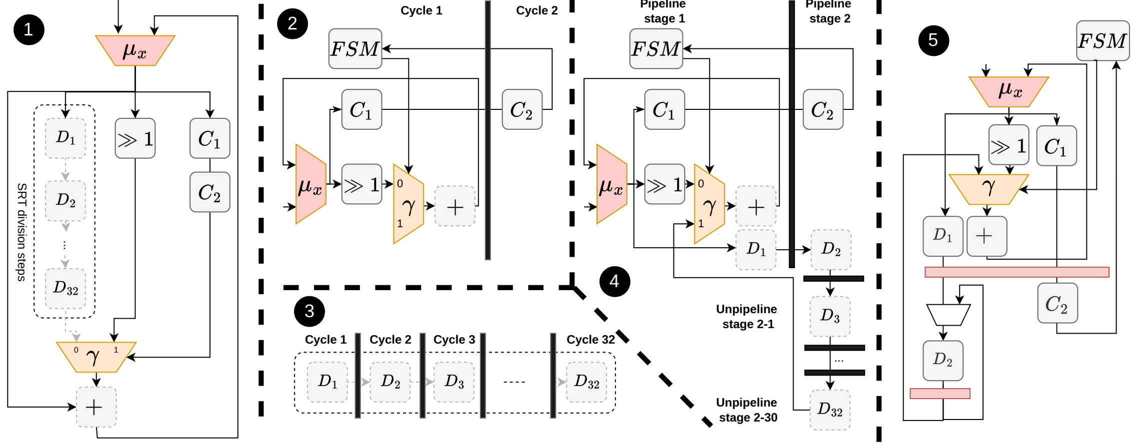


Figure 4. Internal representation of the division example presented in Figure 3, and the steps involved when scheduling with unpipelining.

speculative inputs of the γ-node, and applying dead-code elimination on the graph to slice out all the nodes to unpipeline. After this step, we obtain two different graphs: a cyclic graph containing the core of the speculative pipeline, and a DAG containing the unpipelined instructions.

2) **Scheduling:** The two graphs are scheduled separately. The first one is scheduled using the standard SLP algorithm, deriving the different pipeline stages. The unpipelined graph is scheduled using a resource-constrained scheduler that aims to balance latency and resource utilization. The scheduler implemented is a heuristic loosely based on list scheduling; however, an ILP-based scheduler could also be considered. Part ❷ in Figure 4 depicts the scheduled core of the pipeline, with the two different pipeline stages. Part ❸ illustrates the 32-cycle schedule obtained for the unpipelined graph.
3) **Merging:** The two schedules are combined to obtain the final one. For each γ-node, if its condition requires n cycles to be computed, the first n cycles of the unpipelined schedule are reintegrated in the pipeline. The other cycles are integrated as unpipeline stages and will be executed when a mispeculation occurs, driven by the FSM, while the previous stages of the pipeline are stalled. This combination is illustrated in part ❹ of Figure 4: the Pipeline stages are represented horizontally, while Unpipeline stages are represented vertically, aligned with the pipeline stage where the mispeculation is detected. We can observe that the first two steps of the division are executed in pipeline stages 1 and 2. Then, unpipeline stages are created for all the other steps.

After building the final schedule, the design can be simplified by applying a datapath-merging transformation over mutually exclusive operations as discussed in Section II. By construction, we know that operations in an unpipeline stage are mutually exclusive with all operations in previous stages (i.e., operations in other unpipeline stages, or operations at an earlier pipeline stages). In the final schedule of Figure 4, every D_2, ... D_{32} and their delays can be merged. This results in the datapath represented in part ❺, which corresponds to the pipeline trace

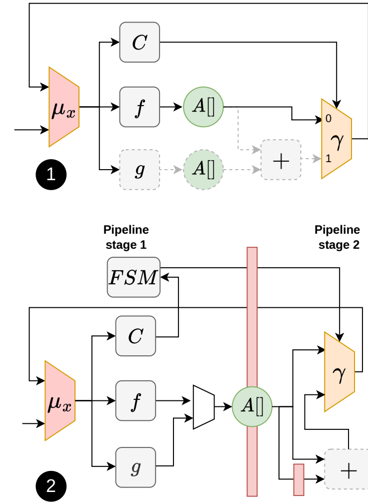


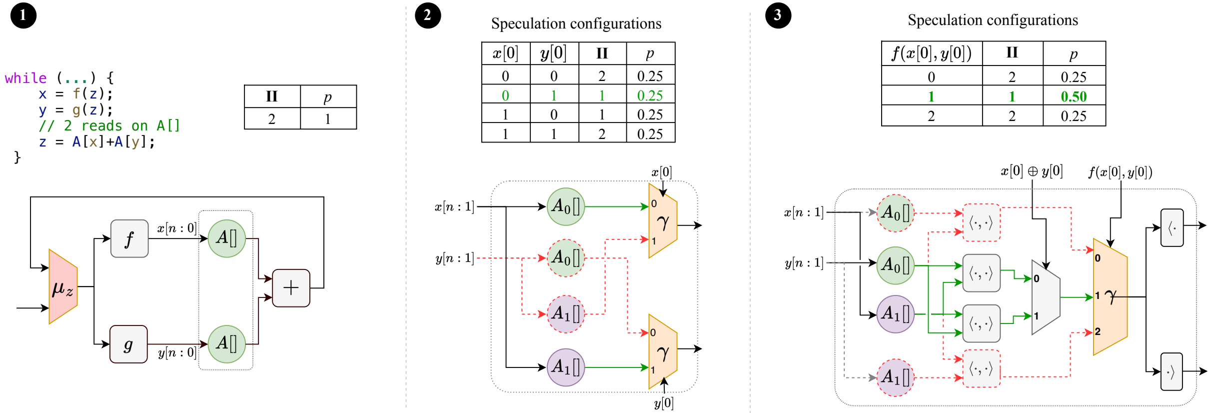
Figure 5. Initial IR for the second example of Figure 3, and the circuit generated after unpipelining.

represented in Figure 3, where all operations are executed on the D_2 resource.

D. Speculative resource sharing

The main challenge for handling resource constraints in SpecHLS is to guide the exploration step (which is in charge of deciding where to speculate) to select a speculation configuration capable of solving the resource conflict. Then, unpipelining is used to enable resource sharing.

As mentioned earlier, the SpecHLS exploration step determines the speculation configurations with $\mathrm{recMII} = 1$. This algorithm has been slightly improved to integrate resource constraints using the resMII: whenever a speculation configuration is tested, we also slice the operations that would be unpipelined, and compute the number of operations of a particular type, divided by the constraint on this resource. This results in a minimal achievable II for resources. Figure 5 depicts the GSSA representation for the second motivational example in part ❶. If the exploration tries to apply no speculation, the



II	p
2	1

$x[0]$	$y[0]$	II	p
0	0	2	0.25
0	1	1	0.25
1	0	1	0.25
1	1	2	0.25

$f(x[0], y[0])$	II	p
0	2	0.25
1	1	0.50
2	2	0.25

Figure 6. Array partitioning implemented as an IR transformation that enables speculative array partitioning. The operator $\langle\cdot,\cdot\rangle$ represents pairing.

recMII is one, while the resMII on memory accesses is two. When speculating over the first input of the γ-node, the second memory access is sliced out before computing the resMII, leading to $\text{recMII} = \text{resMII} = 1$. In such a situation, the second memory access, along with other operations impacting the non-speculated input of the γ-node, is unpipelined. Part ❷ in Figure 5 represents the final datapath obtained when performing scheduling with unpipelining on this example.

Note that the scheduling of the unpipelined graph has been slightly modified to accommodate the new resource constraints. The idea is to prevent the first cycles (which are reintegrated in the pipeline) from causing a resource conflict. Consequently, when applying the resource constraint schedule, additional constraints are added to the first cycles.

E. *Speculative Array Partitioning*

The ability to manage resource constraints speculatively opens new optimization opportunities, particularly when combined with established HLS techniques. Consider a loop where each iteration performs two memory accesses, but only a single memory port is available. When the access pattern is regular, classical HLS optimizations such as *static array partitioning* can distribute data across multiple banks, ensuring that the two accesses target different banks and allowing an initiation interval (II) of one. However, with random or unpredictable addresses, conflicts cannot be prevented, and conventional HLS tools must conservatively enforce an II of two, issuing one access per cycle. This is illustrated in Part ❶ of Figure 6.

We introduce *Speculative Array Partitioning*, which dynamically splits the array into two banks and speculates that the two accesses target different banks. If the speculation holds, both accesses proceed in parallel (II = 1). If a conflict is detected, the pipeline stalls for one additional cycle to issue the second access, similar to branch misprediction recovery.

The transformation is illustrated in Part ❷ of Figure 6. First, consider a loop iteration with two accesses to the same array. By applying a block-cyclic partitioning with a factor of two, the array is split into two banks. When the index expressions cannot be resolved at compile time, each original access may target either bank. In the IR, this is represented by two γ-nodes per access, each directing the operation to a specific bank.

As briefly mentioned in Section II, during the exploration phase, our framework evaluates possible speculative configurations and selects the one with the highest expected performance gain. This exploration is based on the concept of speculation configurations. A *speculation configuration* captures speculation strategies. It consists of a list of γ-nodes inputs, which identify all active speculations in that scenario, and a probability score based on its occurrence obtained by profiling. The flow then explores the space of configuration to determine the speculation configurations that lead to $\text{II} = 1$, and which have the highest chance of occurring [9].

In Part ❷ of Figure 6, the set of all possible speculation configurations is represented, along with their corresponding resMII. Here, the exploration algorithm will select the configuration that can achieve $\text{resMII} = 1$ (i.e., conflict-free) with a probability of $p = 0.25$. This results in a circuit with an average II of 1.75, a modest 12.5% throughput improvement over the baseline case with $\text{II} = 2$.

Greater benefits can be achieved by resorting to a modified implementation of partitioning, as illustrated by Part ❸ of Figure 6. Here, the two independent binary γ nodes are merged into a single four-input node, enabling speculation across both accesses jointly. With this enhanced IR, our SLP framework identifies two outcomes that allow II = 1 with a probability of 50%, improving throughput by 25%. Finally, increasing the partitioning factor to four banks would further reduce the conflict probability, yielding a theoretical throughput improvement of up to 37.5%, at the price of an increased area overhead. It should be noted that this modified partitioning transformation is easy to automate, as it simply involves enumerating and isolating conflict-free access patterns into their own execution paths (details are omitted due to space constraints).

IV. Experimental validation

Our methodology has been integrated into the SpecHLS source-to-source compiler [6]. All experiments target an Xilinx XC7A25 FPGA using Vitis HLS 2023.1, with a target clock period of 10 ns. Existing HLS benchmarks [20], [21] focus on kernels with regular control-flow that work well with state-of-the-art HLS tools and are not suitable to evaluate our approach. Instead, the proposed approach is evaluated among several kernels

Table I
TIMING AND AREA AFTER HLS AND LOGIC SYNTHESIS FOR THE INPUT PROGRAM (STATIC PIPELING), THE OUTPUT OF SPECHLS, AND THE OUTPUT OF SPECHLS WITH THE UNPIPELINING TRANSFORMATION. T_{clk} IS EXPRESSED IN NS.

	Static pipelining				SpecHLS					Unpipelined				
	II	T_{clk}	LUT/FF	DSP	II	T_{clk}	LUT/FF	DSP	CPI	II	T_{clk}	LUT/FF	DSP	CPI
fastSlow	6	7.3	419/198	0	1	9.26	819/523	0	1.5	1	5.3	217/320	0	1.5
arrayPartition	4	6.5	194/120	1	3	11.3	336/376	2	3.3	1	11.6	319/290	2	1.1
resConstraint	3	5.5	188/115	0	2	4.1	239/316	0	2.4	1	5.6	296/269	0	1.5
getNormTrans	8	27.1	2.4k/1.7k	30	1	27.1	4.2k/2.2k	47	2.6	1	21.3	1.7k/2.2k	5	6.3
BlockFP	2	7.4	302/176	0	1	11.8	727/1k	0	1	1	15.0	550/294	0	1
div	33	11.6	786/491	0	1	9.7	2.2k/1.7k	0	1.5	1	6.9	344/306	0	1.5
FMA	4	6.4	327/238	3	1	10.1	500/397	3	1.6	1	10.7	522/444	0	1.7
histoX2	2	4.8	260/116	0	—	—	—	—	—	1	6.3	897/1.1k	0	1.5

presenting opportunities for unpipelining and resource sharing. The kernels named `div`, `fastSlow`, `getNormTrans`, and `BlockFP` are applications that expose an opportunity to exploit resource sharing. The other kernels (`resConstraint`, `arrayPartition`, `FMA`, and `histoX2`) presents resource constraints on memory ports. Note that kernels `fastSlow`, `resConstraint`, and `arrayPartition` are synthetic examples that aim at validating the transformations presented in Sections III-C, III-D, and III-E, respectively. Each benchmark was implemented in three different versions for comparison: (i) *Static pipelining* version, directly generated with Vitis HLS, (ii) *SpecHLS* version without the proposed techniques, (iii) *Unpipelined* version involving the unpipelining and resource sharing.

Table I reports post-synthesis timing and resource utilization (number of LUTs, FFs, and DSPs). The throughput of the generated circuits is reflected by the average number of Cycles Per Iteration (CPI) and the operating frequency. The Table also reports the static II achieved for loop pipelining. Our approach achieves a decrease of 34% of the circuit area and an increase of 35% of the throughput compared to the SpecHLS-generated circuits. The area reduction stems from the resource sharing enabled by scheduling with unpipelining, and the throughput improvement results from the reduction of II due to better handling of resource constraints. The SpecHLS results for `histoX2` are not reported here because SpecHLS cannot handle two write operations in a single array. Results for `getNormTrans` highlight an important increase in the CPI between SpecHLS and Unpipelined. This loss of throughput is caused by aggressive slicing that unpipelined operations outside the strongly connected component. Consequently, the generated design stalls during the execution of operations that could have been computed in advance, but uses fewer floating-point operators.

V. Related work and discussion

The problem of resource sharing for dynamic and speculative HLS has been addressed in research works. Most of these works rely on the insertion of an external arbiter to handle access conflicts [2], [3], [22], [23].

By contrast, Cheng et al. [24], [25] insert static islands into an otherwise fully dynamically scheduled circuit. These islands reduce the area overhead of dynamic scheduling because no fine-grained synchronization is required inside an island; moreover, their internal static schedule enables classical resource-sharing optimizations. However, sharing cannot occur across islands or between islands and dynamic regions. Our approach lifts this limitation by mapping all operations onto a single, unified datapath, thereby enabling resource sharing across regions.

Other approaches, such as Josipović et al. [2], Li et al. [3], and Dai et al. [22], introduce dedicated arbiters to manage access to shared resources. In Josipović et al.'s approach, two operators may share a resource provided that the instantaneous demand never exceeds the operator's service capacity (i.e., no more concurrent work is issued than can be absorbed within the operator's latency); an arbiter then enforces a safe ordering of requests to prevent deadlocks.

By contrast, our method relies on static resource sharing between operations that are provably never active simultaneously. We do not merge arbitrary operators, but deadlock cannot occur because the global SpecHLS finite-state machine (FSM) deterministically orchestrates execution. A further advantage of unpipelining over arbiter-based schemes is that it requires no additional per-operator arbiters: the existing SpecHLS FSM inherently performs the necessary arbitration for each operator fusion, incurring no extra hardware.

VI. Conclusion

Speculative Loop Pipelining opens up new opportunities for automatic circuit design when combined with High-Level Synthesis. It allows the generation of designs with a more aggressive pipeline. But it can induce a large area overhead. In this paper, we propose a new mechanism to handle speculation in High-Level Synthesis called unpipeline. It separates the circuits into a fully pipelined part and a sequential execution part to exploit state-of-the-art resource-sharing techniques, thereby reducing the area overhead of speculation. On average, our method achieves a diminution of 34% of the circuit area compared to existing speculative high-level synthesis tools.

Acknowledgment

This work is partly funded by the French National Research Agency (ANR), as part of the LOTR project[2] (ANR-23-CE25-0016)

[2]https://lotr.irisa.fr/

References

[1] 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. New York, NY, USA: Association for Computing Machinery, Feb. 2018, pp. 127–136. [Online]. Available: https://dl.acm.org/doi/10.1145/3174243.3174264

[2] L. Josipović, A. Marmet, A. Guerrieri, and P. Ienne, "Resource sharing in dataflow circuits," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 4, pp. 1–27, 2023.

[3] R. Li and R. Manohar, "Pipelink: A pipelined resource sharing system for dataflow high-level synthesis," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2025, pp. 182–182.

[4] 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. New York, NY, USA: Association for Computing Machinery, 2019, p. 162–171.

[5] S. Derrien, T. Marty, S. Rokicki, and T. Yuki, "Toward Speculative Loop Pipelining for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4229–4239, Nov. 2020.

[6] J.-M. Gorius, S. Rokicki, and S. Derrien, "SpecHLS: Speculative Accelerator Design Using High-Level Synthesis," *IEEE Micro*, vol. 42, no. 5, pp. 99–107, 2022.

[7] ——, "A unified memory dependency framework for speculative high-level synthesis," in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, 2024, pp. 13–25.

[8] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 318–328. [Online]. Available: https://doi.org/10.1145/53990.54022

[9] D. Leothaud, J.-M. Gorius, S. Rokicki, and S. Derrien, "Efficient design space exploration for dynamic & speculative high-level synthesis," in *34th International Conference on Field-Programmable Logic and Applications*, 2024.

[10] Y. She, J. Liu, Y. Huang, R. C. Cheung, and H. Yan, "A speculative loop pipeline framework with accurate path modeling for high-level synthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 18, no. 2, pp. 1–33, 2025.

[11] Z. Zhang and B. Liu, "Sdc-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2013, pp. 211–218.

[12] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 433–438.

[13] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 2, Apr. 2011. [Online]. Available: https://doi.org/10.1145/1929943.1929947

[14] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 969–980, 2005.

[15] M. Stojilović, D. Novo, L. Saranovac, P. Brisk, and P. Ienne, "Selective flexibility: Breaking the rigidity of datapath merging," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 1543–1548.

[16] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] P. Tu and D. Padua, "Efficient building and placing of gating functions," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 47–55.

[18] ——, "Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers," in *Proceedings of the 9th International Conference on Supercomputing*, ser. ICS '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 414–423. [Online]. Available: https://doi.org/10.1145/224538.224648

[19] 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 (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[20] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2008, pp. 1192–1195.

[21] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 269–278. [Online]. Available: https://doi.org/10.1145/3174243.3174255

[22] 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*, 2017, pp. 189–194.

[23] V. Lapotre, P. Coussy, C. Chavet, H. Wouafo, and R. Danilo, "Dynamic branch prediction for high-level synthesis," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–6.

[24] J. Cheng, L. Josipovic, 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*, 2020, pp. 288–298.

[25] J. Cheng, J. Wickerson, and G. A. Constantinides, "Finding and finessing static islands in dynamically scheduled circuits," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 89–100.