SpecHLS: Speculative Accelerator Design using High-Level Synthesis

Jean-Michel Gorius, Simon Rokicki, Steven Derrien

SpecHLS: Speculative Accelerator Design using High-Level Synthesis

Jean-Michel Gorius
Univ Rennes, Inria, CNRS, IRISA

Simon Rokicki
Univ Rennes, Inria, CNRS, IRISA

Steven Derrien
Univ Rennes, Inria, CNRS, IRISA

***Abstract*—Custom hardware accelerators usage is shifting towards new application domains such as graph analytics and unstructured text analysis. These applications expose complex control-flow which is challenging to map to hardware, especially when operating from a C/C++ description using High-Level Synthesis toolchains. Several approaches relying on speculative execution have been proposed to overcome those limitations, but they often fail to handle the multiple interacting speculations required for realistic use-cases. This paper proposes a fully automated hardware synthesis flow based on a source-to-source compiler that identifies and explores intricate speculation configurations to generate speculative hardware accelerators.**

INTRODUCTION

Modern CPU architectures use aggressive speculation strategies to improve their performance. Speculation uncovers parallelism that can be used to issue more instruction per cycle but incurs significant area overhead. Custom hardware accelerators do usually not implement speculation: its cost and design complexity generally outweigh its benefits.

Custom accelerators are carving out their niche in new domains such as graph analytics, sparse linear algebra, etc. Such workloads are challenging to map to hardware and may benefit from speculative execution. For such designs, RTL-level design methodologies would be impractical.

High-Level Synthesis (HLS) partly address the problem: a designer can start from a C/C++ specification and refine it through a design space exploration step to derive a highly-optimized hardware implementation. However, current HLS tools have limitations for programs with complex control flow or data-dependent behavior.

There have been attempts to better support this type of kernels, notably through the use of dynamic or speculative schedules [1], [2], [3], [4]. Nevertheless, to the best of our knowledge, none of the existing approaches is general enough to support arbitrary speculation patterns.

This work introduces SpecHLS, a source-to-source compiler framework supporting speculative loop pipelining for arbitrary control-flow and memory speculation patterns. The toolchain is evaluated on a set of applications that can benefit from speculative execution. Results show that the proposed approach can provide an order of magnitude performance improvements without suffering from a resource usage explosion.

Published by the IEEE Computer Society

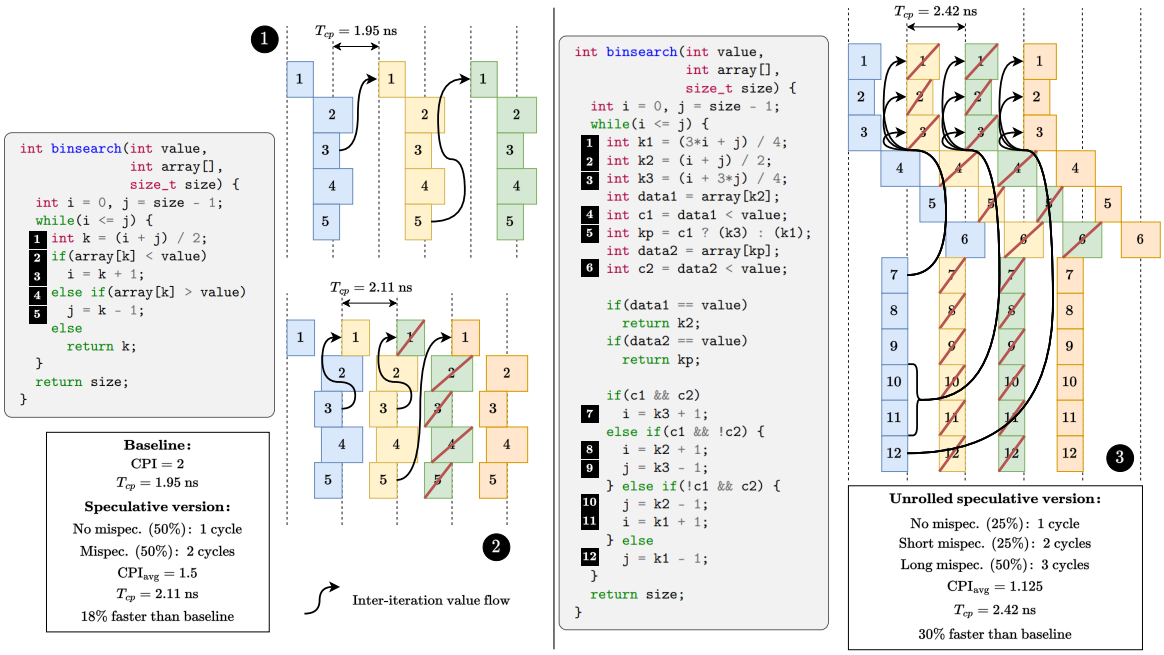

Figure 1. Loop pipelining strategies for a binary search kernel application. The leftmost part represents the C code for the basic kernel and execution traces for the baseline version and for the speculative version. The rightmost part represents the unrolled version of the kernel and the execution trace for the speculative version.

Speculation Considered Useful

As a running example, we use a binary search algorithm shown in the leftmost part of Figure 1. The kernel exposes a loop-carried dependency over `i` and `j` which prevents overlapping consecutive iterations. As a consequence, loop pipelining leads to a minimum initiation interval of $\mathrm{II} = 2$, with a post-synthesis clock speed of $T_{\mathrm{cp}} = 1.9\,\mathrm{ns}$ on an XU280 FPGA. The corresponding execution trace is given in Part ❶ of Figure 1.

However, it is possible to speculate that the first conditional 2 is taken and to immediately start the next iteration, as illustrated by the trace in Part ❷ of Figure 1. The resulting circuit operates at a lower clock speed $T_{\mathrm{cp}} = 2.11$ and suffers from a high mispeculation rate (50%), assuming an unskewed data distribution within `array`. However, mispeculated iterations only take two cycles to execute, just as for the baseline accelerator. The average Cycles Per Iteration (CPI) is therefore $\mathrm{CPI}_{\mathrm{avg}} = 1.5$ instead of 2. Taking into account the increased value of T_{cp}, we can conclude that the speculative accelerator improves performance by 18%.

Let us further consider a manually unrolled version of the kernel, as show on the right of Figure 1. In this instance, it is possible to speculate two iterations ahead, with a mispeculation rate of 75% and an even lower clock speed ($T_{\mathrm{cp}} = 2.42$), as represented in Part ❸. The next iteration starts immediately with the speculated values for `i` and `j`. After one cycle, the first condition is resolved, and there is a 50% chance for the first speculation to be correct. In such a case, we immediately start the next iteration with another speculated value for `i` and `j`. This second speculation also has a 50% chance to be correct. Consequently, 25% of iterations speculate correctly and need one cycle to be executed. 50% of iterations mispeculate once and need two cycles. 25% of them mispeculate twice and need three cycles. On average, the accelerator needs 2.25 cycles to execute one iteration. This duration corresponds to two iterations of the not-unrolled kernel. The unrolled speculative accelerator leads to the most profitable strategy among the three shown in Figure 1.

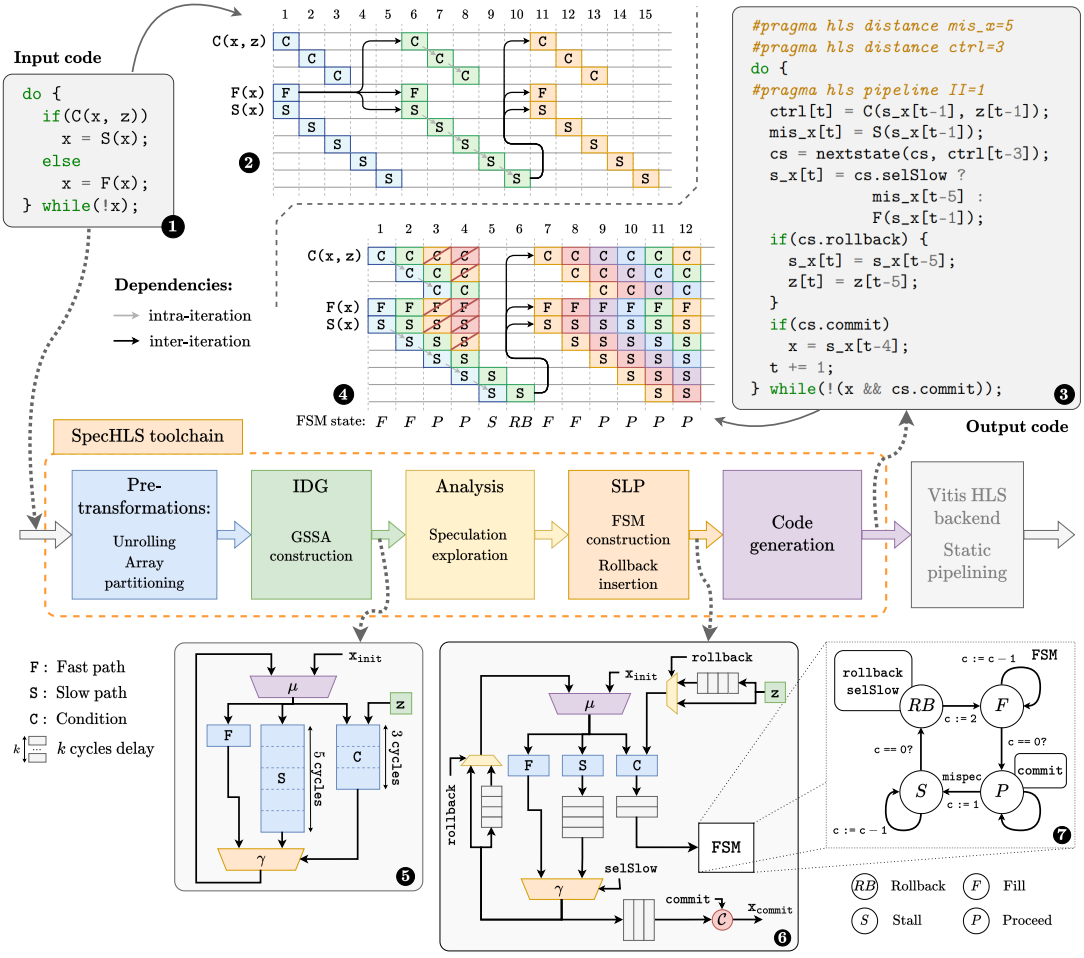

Figure 2. SpecHLS source-to-source transformation flow. The toolchain takes C code ❶ as an input and produces transformed C code ❸. ❷ and ❹ show the respective schedules of the input and output code. The toolchain operates on the intermediate representation depicted in ❺ (before transformation) and ❻ (after transformation). The speculation-controlling FSM is outlined in ❼.

There are three lessons to be learned from this simple example. First, speculative execution does not require high-confidence predictions to be profitable. Second, it is often necessary to resort to intertwined or nested speculations. Last, the best solution is not always the most obvious one.

SpecHLS Compiler Flow

SpecHLS is a source-to-source compiler transformation flow, depicted in Figure 2. SpecHLS accepts C code as input and produces transformed C code with support for speculative loop pipelining. Vitis HLS can then exploit the latter to obtain an accelerator design.

The key idea of speculative loop pipelining [5] (SLP) is that speculative pipelining is considered as a parallelizing transformation rather than a back-end optimization. Consequently, SpecHLS does not address the scheduling of individual operations in the pipeline, and delegates that task to the HLS tool.

The C code produced by SpecHLS from the code in Part ❶ is illustrated in Part ❸ of Figure 2. A corresponding execution trace is shown in Part ❹. The output code ❸ exposes longer reuse distances and implements the pipeline hazard management logic. Increasing the reuse distance allows the HLS pipeliner to work more aggressively to achieve a lower II, as there are less constraints imposed by data-dependencies

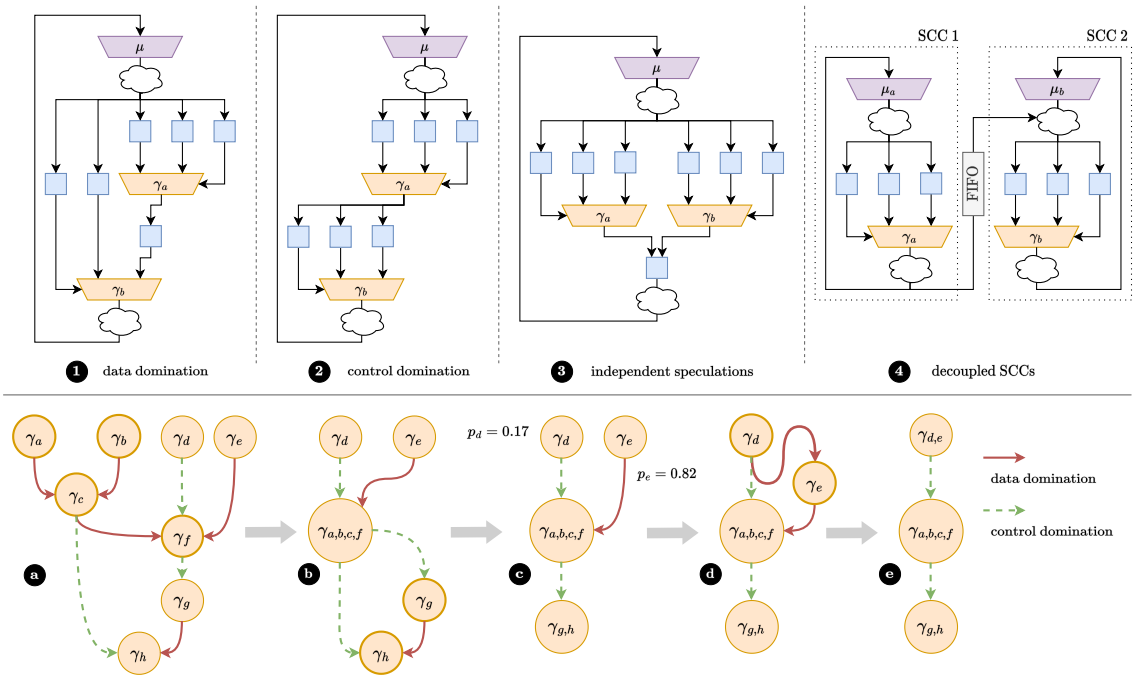

Figure 3. Taxonomy of γ-node patterns handled by our approach (top) and example of γ-node graph reduction (bottom).

between iterations.

The choice of the reuse distance is performed by the SpecHLS transformation flow. The larger the distance, the more aggressive the speculation can be. There is, therefore, a trade-off between (i) values of II and T_{cp} and (ii) area and misspeculation recovery cost. The following describes the SpecHLS transformation flow in more detail.

SpecHLS Intermediate Representation

The proposed compiler flow relies on a program representation based on the Gated-SSA [6] intermediate representation (IR), which is an extension of the classical SSA representation. SSA captures both control-and dataflow dependencies by inserting ϕ-nodes at the program's control-flow joining points.

Gated-SSA distinguishes different types of ϕ-nodes: loop headers are categorized as μ-nodes, and the remaining nodes, which correspond to conditionals, are categorized as γ-nodes. To the difference of SSA ϕ-nodes, γ-nodes include an additional input that exposes the control-flow decision, making them similar to multiplexers.

Part ❺ and Part ❻ of Figure 2 depict simplified Gated-SSA representations of the programs in ❶ and ❸. In this example, the conditional is exposed as a γ-node that selects the value used for the next iteration based on its control port. These values are obtained from three distinct paths: fast (`F`), slow (`S`), and conditional (`C`).

Because they capture conditionals, γ-nodes expose speculation opportunities. For example, speculating that the γ-node in ❺ selects the fast path enables pipelining with $\text{II} = 1$. To be considered for speculation, a given γ-node must (i) expose a long delay over at least one of its data or control inputs, (ii) expose a short delay over one of its input data ports, and (ii) the probability for the latter path to be taken must be higher than a given threshold. The latter probability can be obtained either analytically or by profiling the input code.

The SLP transformation operates on the loop body's IR, and its result is illustrated in Part ❻ of Figure 2. Each loop produces a set of strongly connected component (SCC) in the intermediate representation. Delays are inserted to increase reuse distances on non-speculated paths (e.g., path involving `S` and `C`). Those delays increase

the reuse distance for loop-carried dependencies and provide opportunities for the HLS pipeliner to overlap execution of independent computations. Such delays allow the back-end HLS toolchain to produce a schedule with an initiation interval of $\mathrm{II} = 1$.

Recovery logic is also inserted to handle mispeculations. This logic consists of (i) rollback nodes for restoring past inputs and (ii) commit nodes for filtering invalid outputs. The recovery mechanism is controlled by a Finite State Machine (FSM) depicted in ❼. Previous work [5] gives more details on the way those nodes are inserted.

Existing speculation techniques for accelerators [5], [4] are restricted to cases where speculations are independent of one another and do not interfere. In practice, as exemplified in Figure 1, speculations are often intertwined, making recovery logic more complex. Precisely characterizing how dependent speculations interact is therefore necessary.

In this work, we generalize speculative loop pipelining [5] to support arbitrary speculation configurations that involve one or more speculative decisions. We achieve this by partitioning the intermediate representation into a set of *speculation patterns*, *i.e.* disjoint sets of γ-nodes. We propose a taxonomy of speculation patterns, coupled with a set of transformations rules. We use these rules to rewrite arbitrary patterns into a normalized representation supported by our control logic generation flow.

Classification of multiple speculation patterns

Handling patterns involving multiple speculations is challenging: the outcome of a given speculation may depend on the outcome of another one, resulting in intricate recovery schemes. This complexity depends on the dependency graph relating the speculated γ-nodes in the target loop.

This section presents how SpecHLS handles four γ-node patterns that arise when dealing with multiple speculations, namely *data domination*, *control domination*, *independent γ-nodes* and *decoupled SCCs*. The iterative resolution of these patterns from the Gated-SSA intermediate representation is described in a later section.

Data Domination

Data domination (DD) is a speculation pattern where speculative values only flow through the data inputs of γ-nodes. An example of data domination is provided in Part ❶ of Figure 3. In this pattern, the speculation chooses the shortest path through all nodes of the pattern. Once conditions have been resolved, and in case of mispeculation, the control logic rolls back the computation and the speculation chooses the second-shortest path in the pattern. The same process is repeated if another mispeculation occurs.

The execution trace depicted in Part ❸ of Figure 1 is an example of data domination: at cycle 2, an iteration is started using the output of computation 7 from the first iteration; during cycle 2, the condition computed by 4 reveals a mispeculation, and the iteration is started again at cycle 3 using the output of 10 and 11; during cycle 3, the condition computed by 6 reveals another mispeculation, and the iteration is started a third time at cycle 4 using the output of 12.

Data domination is handled using a single Finite State Machine (FSM), which controls all the γ-nodes alongside the rollback mechanisms. This FSM is built by composing the FSMs that correspond to single speculations for each γ-node in the pattern.

Another challenge arises when trying to handle data domination patterns in which values do not flow directly from one γ-node to another. This scenario is depicted in Part ❶ of Figure 3: a block processes the output of γ_a before sending its output to γ_b. The additional computation may de-synchronize the resolution of the conditions driving γ_a and γ_b and would require additional control logic to ensure proper speculative behavior. We choose to simplify this pattern by duplicating this extra computation on all inputs of γ_a, thereby generating additional hardware. Better handling of this particular case of data domination is left for future work.

Control Domination

In a control domination (CD) pattern, the output of a γ-node impacts the control input of another one, as depicted in Part ❷ of Figure 3. The challenge in such a speculation pattern is to correctly handle situations where a mispeculation is triggered based on incorrect data from previous

mispeculations. The latter's condition may not have been resolved when the former mispeculation occurs.

Control domination is handled by inserting a rollback mechanism and an FSM for each γ-node. When a mispeculation is signaled by an FSM, it resets every FSM having a shorter condition and prevents FSMs with longer conditions from handling a mispeculation signal resulting from the mispeculation being handled by itself. This behavior is achieved using a mask system that hides some mispeculation signals. Using control-dominated speculations, we can handle an arbitrary number of speculative γ-nodes, ensuring a safe rollback mechanism in any situation.

Consider the example depicted in Part ❷ of Figure 3, assuming that the condition controlling γ_a is longer than the one controlling γ_b. A mispeculation from γ_a resets the FSM controlling γ_b. A mispeculation from γ_b hides mispeculation signals to γ_a for $\text{latency}(\texttt{Cb})$ cycles, starting after $\text{latency}(\texttt{Ca}) - \text{latency}(\texttt{Cb})$ cycles.

Independent Speculations

Two γ-nodes are independent if they belong to the same loop, but do not interact within a given loop iteration, as depicted in Part ❸ of Figure 3. However, as both nodes are in the same SCC, the values produced by the γ-nodes are used by one another at the next iteration. If one speculation fails at a given iteration, the values used to compute the next iteration are wrong for both iterations. Consequently, this pattern cannot be handled using two independent FSMs. We insert an artificial dependency between independent γ-nodes and consider them as a data domination pattern. We create a link from the node with the lowest mispeculation probability to the node with the highest one, minimizing the mispeculation probability for the resulting pattern. The pattern depicted in Part ❸ of Figure 3 would be equivalent to a single four-input γ-node choosing between all pairs of values from the two original γ-nodes.

Decoupled SCCs

The last pattern consists of γ-nodes in different SCCs, as depicted in Part ❹ of Figure 3. Even if there are interactions between the SCCs, it is possible to schedule each of them using decoupled software pipelining [7]. Consequently, each γ-node has its own FSM, and the different speculations are decoupled using a FIFO between the two SCCs.

Iterative Speculation Pattern Resolution

Because there may be several patterns involved in the same loop, we build a directed acyclic graph capturing the interactions between all γ-nodes within an SCC. The bottom right part of Figure 3 represents such a graph, where edges capture interaction patterns.

We start with one node of the graph for each γ-node in the SCC. We merge nodes in this graph, with each merging operation corresponding to the insertion of an FSM. When merging patterns recursively, we build a product FSM to drive the resulting grouping. At the end of this deterministic process, we obtain a chain of control-dominated nodes for the entire loop body that we handle with an FSM for each node.

In Figure 3, we start from an SCC with eight distinct γ-nodes labeled γ_a to γ_h. Data dominations are represented using plain arrows and control dominations using dashed arrows. Two nodes can only be linked by a single edge. We start by merging DD patterns: in ⓐ, we merge γ_a, γ_b, γ_c, and γ_f, preserving the incoming and outgoing edges for this group of nodes. We merge γ_g and γ_h in ⓑ, but the resulting node would have two incoming edges from $\gamma_{a,b,c,f}$: a control domination and a data domination. Since the former is more generic than the latter, this pattern is resolved as a single control domination to get ⓒ. At this stage, we could either merge γ_d or γ_e into $\gamma_{a,b,c,f}$. We examine the mispeculation probabilities for each node, p_d and p_e, to take a decision and insert an artificial direct data domination between γ_d and γ_e in ⓓ since $p_d < p_e$. Finally, we merge γ_d and γ_e and resolve the two edges going into $\gamma_{a,b,c,f}$ as a control domination in ⓔ.

Experimental evaluation

We evaluate our technique on three examples identified as good candidates for our approach. We generate N_{conf} speculative versions using more or less aggressive speculation configurations and provide the results for the most relevant ones. We compare the performance and resource usage of the speculatively pipelined designs with

Speculation configuration	N_{conf}	II	$F_{\max}$ (MHz)	CPI	Speed-up	Area LUT	Area FF	Area SRL	Area DSP
Binary search									
Baseline	1	2	513	2	1×	128	100	0	0
Speculative	48	1	473	1.5	1.2×	353	371	0	0
Unrolled speculative	108	1	413	1.1	1.5×	813	657	94	0
Hmmer									
Baseline (M=8)	1	4	221	4	1×	5,707	3,006	0	0
Speculative (M=8)	64	1	186	1	3.4×	10,048	10,391	879	0
Baseline (M=16)	1	6	221	6	1×	11,084	5,655	0	0
Speculative (M=16)	64	1	235	1	6.4×	19,756	18,265	2,895	0
SKA gridding									
Baseline	1	8	273	8	1×	2,238	2,951	0	10
Speculative	112	1	268	1	7.9×	34,267	40,309	3,066	80
RISC-V CPU									
Baseline	1	2	287	2	1×	2,047	1,293	0	4
OpStall	12	1	248	1.3	1.3×	2,634	1,773	0	4
Kernels dhrystone				1.29	1.34×				
Kernels matmul				1.18	1.47×				
Kernels median				1.33	1.31×				
Kernels gcd				1.37	1.26×				
RegStall	37	1	256	1.3	1.4×	2,763	1,971	0	4
Kernels dhrystone				1.35	1.33×				
Kernels matmul				1.22	1.47×				
Kernels median				1.34	1.33×				
Kernels gcd				1.37	1.30×				

Figure 4. Results of the experimental study for our examples. The first columns show the different accelerator's performance and their relative speed-up compared to the baseline implementation. The last columns show their area cost.

the non-speculative versions. Our results were obtained with Vitis HLS, targeting an XU280 FPGA.

The results are provided in Figure 4. The first columns summarize the performance results, where II is the Initiation Interval achieved by the HLS tool and $F_{\max}$ is the maximal frequency of the generated hardware. The third column represents the average number of cycles needed to execute one loop iteration (CPI). The speed-up displayed in the table summarizes the performance improvement w.r.t. the baseline implementation, considering both the maximal frequency and number of cycles per iteration. The last values of the table represent the area cost of the different solutions.

In the following, we describe each use-case and explain how speculative execution helps improve the resulting accelerator's performance.

HMM-Based Sequence Comparison

The HMMER software package is used for profile HMM searches in biological sequence databases. Its kernel exposes a dependency pattern preventing parallelization using static techniques, as illustrated in the upper-right part of Figure 5. However, it is possible to speculate with very high confidence (99.9%) over the outcome of a *max* operation that drives `test`. This speculation removes the critical dependency, enabling wavefront parallelization [8]. Whenever a mispeculation occurs, the last safe state is

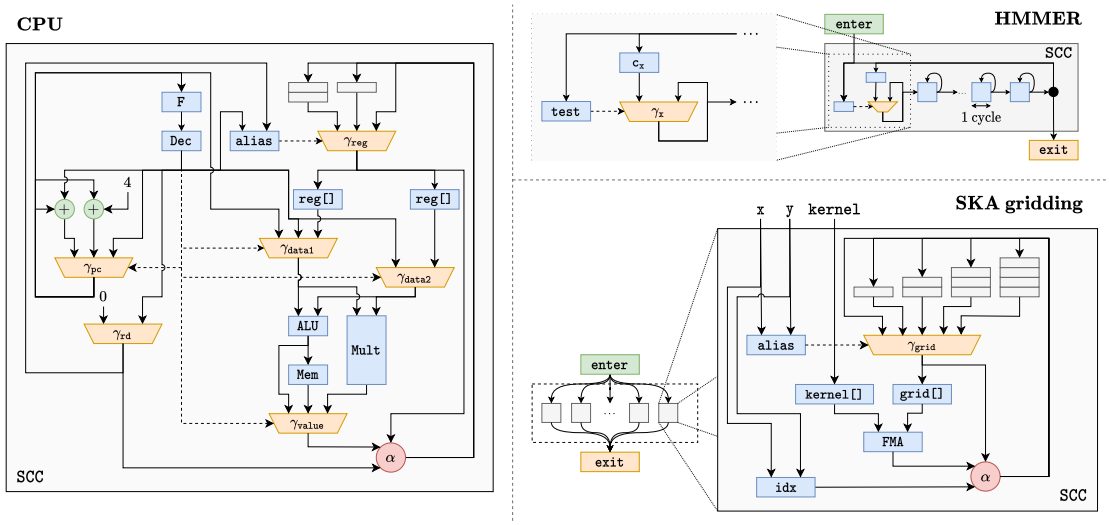

Figure 5. Simplified view of the internal representation of the different use-case applications.

restored, and execution resumes when the correct value is available. Our flow can take advantage of this speculation opportunity and automatically generates a fully parallel speculative accelerator. Results show that the speculative version outperforms the non-speculative one by up to $6.4\times$ for a profile of size $M = 16$, at the price of a $2\times$ increase in LUT count.

Gridding in Square Kilometer Array

The Square Kilometre Array (SKA) aims at becoming the world's largest radio telescope. SKA relies on compute-intensive image gridding and de-gridding stages, where velocity measurements are back-propagated to reconstruct sky images [9].

The gridding algorithm's Gated SSA program IR is summarized in the lower-right part of Figure 5. It consists of two parallel loops (both with low trip counts) which perform the parallel updates of independent pixels, as illustrated in Figure 5. However, the pixel update operation involves a data-dependent Read-After-Write (RAW) loop-carried dependency for array `grid`, which prevents static pipelining.

We use the ability of SpecHLS to perform memory speculation across multiple iterations, extending the technique described in previous work [5]. Memory speculation is achieved by inserting runtime alias checks whose role is to stall the computation in case of an alias, enforce the loop-carried dependency, and speculate that no such alias occurred. The bottom-right part of Figure 5 represents the alias check mechanism for the pixel update operation. The mechanism stalls whenever the RAW reuse distance is lower than four iterations in order to accommodate for the FMA latency and achieves $\mathrm{II} = 1$.

As shown in Figure 4, this approach improves performance by a factor of $8\times$. However, since every pixel update process operates independently, control logic and FIFOs must be duplicated, thereby significantly impacting the design's surface area, especially regarding LUT and FF usage.

In-order Pipelined RISC-V CPU

Several accelerator platforms are based on many-core architectures built out of simple in-order CPU micro-architectures with specialized instruction sets. These CPUs are designed at the RTL level and are difficult to evolve or customize.

This use-case shows that our flow can automatically infer an in-order pipelined micro-architecture for the RISC-V RV32IM instruction set, starting from an Instruction Set Simulator (ISS) model entirely written in C++. The simulator's IR is depicted in the leftmost part of Figure 5, where γ-nodes expose speculation opportunities.

As shown in Figure 4, pipelining the baseline

ISS leads to II = 2, due to the loop-carried dependency over `pc` and `reg[]`. We use our flow to explore the set of possible speculation strategies in two cases: with memory speculation enabled (RegStall design), and without (OpStall design), and pick the best solution as identified by our toolchain. OpStall speculates over $\gamma_{\texttt{pc}}$ that no branch is taken and stalls the pipeline whenever a multiplication is issued. RegStall stalls in the case of a RAW dependency over a register.

Contrary to previous work [5], our approach allows for fine-grain speculation resolution by separately handling the speculations that occur in the pipeline. We avoid resolving all conditions in the execute stage of the processor's pipeline, thereby making this stage shorter and improving the overall design frequency. The performance of each design was evaluated over four kernels (`dhrystone`, `matmul`, `median` and `gcd`). The results in Figure 4 show that SpecHLS improves performance for the OpStall configuration. The RegStall configuration shows a slight performance drop that we expect to be an artifact of the way the HLS backend handles our memory speculation code. Future work will address those limitations..

Discussion

The three use-cases discussed in the previous section demonstrate that speculative execution can improve the performance of hardware accelerators, even when the kernel already exposes iteration-level parallelism. However, this technique would provide only limited performance improvements if applied to standard HLS benchmarks. The reason is that current benchmarks focus on computation patterns that are already well supported by existing HLS tools [10].

Our work precisely targets kernels that are difficult to handle for traditional HLS tools. We believe that speculative pipelining should be seen as an enabling optimization, which is likely to find many unexpected usages.

However, several issues need to be addressed to make the approach practical. For example, there is a need to support full design-space exploration when combining speculations. Our approach also raises many issues from a verification perspective, and a formal equivalence checking technique for our transformation would be highly desirable.

REFERENCES

1. 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*, FPGA '17, pp. 189–194, 2017.
2. E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. L. Lu, "Automatic pipelining from transactional datapath specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 441–454, 2011.
3. 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*, FPGA '18, p. 127–136, 2018.
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*, FPGA '19, p. 162–171, 2019.
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, 2020.
6. P. Tu and D. Padua, "Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers," in *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, p. 414–423, 1995.
7. 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*, PACT '04, pp. 177–188, 2004.
8. T. Takagi and T. Maruyama, "Accelerating HMMER search using FPGA," in *2009 International Conference on Field Programmable Logic and Applications*, pp. 332–337, 2009.
9. B. Veenboer, M. Petschow, and J. W. Romein, "Image-Domain Gridding on Graphics Processors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 545–554, 2017.
10. 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," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

Jean-Michel Gorius obtained his BSc degree and MSc degree from University of Rennes, France, in 2018 and 2021 respectively. He is currently working towards a Ph.D. degree in computer science at the University of Rennes, under the supervision of Steven Derrien and Simon Rokicki. His research interests range from hardware design and hardware synthesis to compilation and high-performance applications. Contact him at jean-michel.gorius@irisa.fr.

Simon Rokicki obtained his PhD degree in computer science at the University of Rennes, under the supervision of Steven Derrien and Erven Rohou. He is now an associate professor at ENS Rennes. His research interests include embedded systems architecture, dynamic compilation, HW/SW co-design and High-Level Synthesis. Contact him at simon.rokicki@irisa.fr.

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 TARAN research group at IRISA. His research interests include High-Level Synthesis, loop parallelization, and reconfigurable systems design. Contact him at steven.derrien@irisa.fr.