A Unified Memory Dependency Framework for Speculative High-Level Synthesis

Jean-Michel Gorius, Simon Rokicki, Steven Derrien

▶ To cite this version:

Jean-Michel Gorius, Simon Rokicki, Steven Derrien. A Unified Memory Dependency Framework for Speculative High-Level Synthesis. CC'24 - ACM SIGPLAN 2024 International Conference on Compiler Construction, Mar 2024, Edinburgh (Ecosse), United Kingdom. 10.1145/3640537.3641581 . hal-04394762v2

HAL Id: hal-04394762

https://inria.hal.science/hal-04394762v2

Submitted on 29 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

A Unified Memory Dependency Framework for Speculative High-Level Synthesis

Jean-Michel Gorius
University of Rennes - Inria - CNRS - IRISA
Rennes, France
jean-michel.gorius@irisa.fr

Simon Rokicki
University of Rennes - Inria - CNRS - IRISA
Rennes, France
simon.rokicki@irisa.fr

Steven Derrien
University of Rennes - Inria - CNRS - IRISA
Rennes, France
steven.derrien@irisa.fr

Abstract

Heterogeneous hardware platforms that leverage application-specific hardware accelerators are becoming increasingly popular as the demand for high-performance compute intensive applications rises. The design of such high-performance hardware accelerators is a complex task. High-Level Synthesis (HLS) promises to ease this process by synthesizing hardware from a high-level algorithmic description. Recent works have demonstrated that speculative execution can be inferred from the latter by leveraging compilation transformation and analysis techniques in HLS flows. However, existing work on speculative HLS lacks support for the intricate memory interactions in data-processing applications. In this paper, we introduce a unified memory speculation framework, which allows aggressive scheduling and high-throughput accelerator synthesis in the presence of complex memory dependencies. We show that our technique can generate high-throughput designs for various applications and describe a complete implementation inside an existing speculative HLS toolchain.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; • **Software and its engineering → Compilers**.

Keywords: High-Level Synthesis, speculation, memory dependencies, code generation

ACM Reference Format:
Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. 2024. A Unified Memory Dependency Framework for Speculative High-Level Synthesis. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3640537.3641581

1 Introduction

The development of increasingly resource-demanding computation domains such as machine learning and graph processing has brought the need for performance optimization to the entire hardware/software stack. Consequently, increasingly heterogeneous hardware solutions are deployed, with programming languages and compilers leveraging such heterogeneity to maximize execution performance [24, 29, 31]. At the other end of the stack, high-performance application-specific hardware accelerators are used to provide fast execution for key operations (*e.g.* machine learning accelerators [19, 42], video encoders/decoders [35]). The increasing complexity of these accelerator's designs has led to the development of design tools that operate at higher levels of abstraction [28, 36].

High-Level Synthesis (HLS) is an example of a hardware design method that leverages compiler techniques to produce hardware. HLS toolchains are *hardware compilers* that transform high-level code (mostly C or C++) into a gate-level hardware description, whose behavior matches the algorithmic behavior of the input program. Since their debut in the late 1990s [10, 20], HLS design flows have become well-established in the industry, with several commercial as well as open-source toolchains available to users [2, 5, 37].

HLS thrives in data-intensive workloads thanks to decades of research by the optimizing compiler community [4, 6, 21, 23, 41]. However, it performs poorly when the input code relies heavily on irregular control-flow decisions. Previous work has identified that such control-dominated code would greatly benefit from dynamic scheduling [18], or even from speculative execution [8, 12, 17], and several new HLS paradigms have emerged from this observation.

In particular, speculative High-Level Synthesis has gained increased attention in the last few years. Recent contributions have brought speculation to elastic circuit design [17] and have made it possible to add speculation to commercial HLS flows [13]. Speculative HLS is a step towards bringing high-performance optimizations previously restricted to general-purpose computing to accelerator design. With its high customization potential, speculation in HLS could

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CC '24, March 2–3, 2024, Edinburgh, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0507-6/24/03…$15.00
https://doi.org/10.1145/3640537.3641581

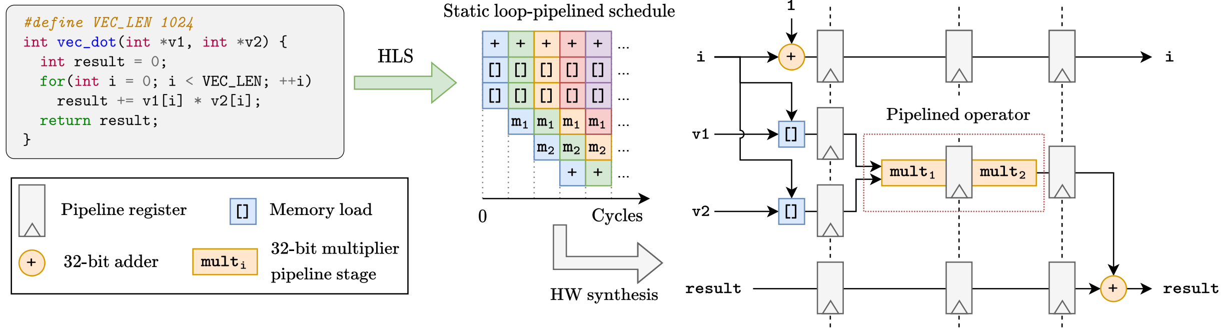

Figure 1. General principle of a High-Level Synthesis toolchain. The input code is transformed into a low-level (RTL) description of the hardware, where each operator is statically scheduled to be executed at a given cycle. Pipeline registers hold intermediate values at cycle boundaries. Some operators, such as the multiplier, may be *pipelined*, i.e., split into more than one execution stage. The number of *pipeline stages* depends on the target frequency, the execution latency of each operator, and the data dependencies between operators.

bring significant performance improvements with a relatively small hardware overhead.

While control-flow speculation in HLS has been described previously [8, 13, 17], existing work does not provide a general way of handling memory accesses in a speculative context. While most work in the HLS field is centered around dynamic dataflow approaches or ad-hoc transformations, such issues deserve a more systematic and compiler-centric approach. This paper introduces a unified framework for handling memory accesses when performing speculative loop pipelining of data-dependent kernels. Our contributions are as follows:

- we introduce a formalized representation of memory accesses in a speculative HLS context;
- we implement memory speculation in a speculative HLS design flow;
- we show experimental results outlining the area-to-performance tradeoffs offered by speculative memory dependency handling in HLS.

To our knowledge, this work is the first to provide a general memory speculation framework for High-Level Synthesis.

This paper is organized as follows. Section 2 presents some background on speculative High-Level Synthesis and on the SpecHLS toolchain on which our work is based, and Section 2.4 motivates our work. Section 3 describes the transformation passes we implement to expose memory speculation opportunities in HLS code, and Section 4 pertains to the code generation phase of our toolchain, which produces code suitable for synthesis using commercial HLS tools from our intermediate representation. Finally, Section 6 discusses some related work, and Section 7 concludes this paper.

2 Speculative High-Level Synthesis

The following section gives a primer on HLS (Section 2.1), while Section 2.2 introduces fundamental speculative HLS principles, and Section 2.3 describes the Intermediate Representation (IR) on which our toolchain operates. Section 2.4 discusses memory access handling in a speculative framework.

2.1 High-Level Synthesis

High-Level Synthesis (HLS) is a hardware design process that transforms a high-level programming language specification of the behavior of a circuit into a Register-Transfer Level (RTL) hardware description. The latter can then be synthesized into a hardware implementation. Unlike hardware description languages such as Verilog and VHDL, HLS tools provide a higher abstraction level, enabling fast iteration times and quick exploration of a vast design space.

A key component of an HLS toolchain is its scheduling algorithm: a good operator schedule exposes more parallelism in hardware, leading to faster designs. Traditional HLS scheduling relies on static dependency analysis and leverages optimizations such as *loop pipelining* [8, 26], an analog to software pipelining in traditional compilers [21]. Pipelined loop iterations' execution overlap, with a new iteration starting every II cycles (the *initiation interval*) and executing during Δ cycles (the *pipeline latency*). Hardware designs strive for fully-pipelined executions, aiming for $\text{II} = 1$. Consequently, HLS compilers try to schedule operations to minimize the initiation interval value.

Figure 1 illustrates the general principle of a High-Level Synthesis toolchain. The input C code is statically scheduled by the synthesis tool, which pipelines loop iterations to overlap their execution. In this example, a new iteration starts at every cycle, and each iteration completes after 4 cycles. Therefore, $\text{II} = 1$ and $\Delta = 4$. Note that the multiplication operator is split in two stages to achieve the target execution frequency. The hardware synthesis step maps this operator schedule to actual hardware, placing pipeline register at cycle boundaries to hold intermediate computation results.

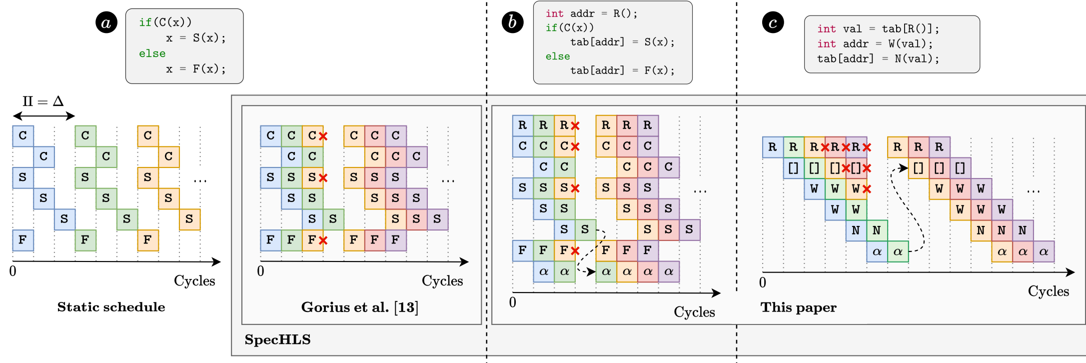

Figure 2. Unified memory dependency handling in speculative HLS allows for transparent handling of array accesses in a speculative context. Part *a* illustrates static scheduling and classical value speculation in HLS [13]. Part *b* and part *c* show the schedules obtained using the techniques presented in this paper in the presence of memory accesses and memory dependencies. The vertical axis represents pipeline stages, and the horizontal axis represents execution cycles. `F` is a *fast* operation (1 cycle), `S` is a *slow* operation (3 cycles), and α denotes an array update.

The bottom left of Figure 2 *a* gives another example of a static schedule, but with $II = \Delta = 3$, since the loop-carried dependency on variable `x` prevents any iteration overlap. The following section describes how to improve this schedule with the help of speculative execution.

2.2 Speculation and High-Level Synthesis

The static scheduling approach falls short when trying to handle control-flow with large latency differentials between the *taken* and the *not-taken* paths, such as in example *a* of Figure 2. In this instance, the static scheduler has to take the worst-case execution scenario into account, which prevents new iterations from starting before each operation of the previous iteration has produced a result. Neither loop unrolling nor user-specified dependency information leads to a satisfactory schedule. In practice, accounting for the longest-latency operation leads to an under-utilization of hardware resources and limits the loop pipelining capabilities of the synthesis tool.

Speculative High-Level Synthesis proposes to solve these inefficiencies by *speculating* that conditionals in the input code always resolve to the fastest path. Such a speculation eliminates the dependency between iteration n and the value of `S(x)` at iteration $n-1$, assuming that `C(x)` is indeed false for the latter. If this assumption happens to be incorrect (i.e., after the result of `C(x)` is known), then all computations that relied on the speculated value need to be aborted (crossed-out operations in Figure 2) and rolled-back. This behavior is similar to the mispeculation handling mechanism in modern CPUs. Applying speculation to the example in part *a* of Figure 2 gives the high-throughput schedule at the bottom right.

This kind of schedule can be obtained by SpecHLS [8, 13], a High-Level Synthesis flow that introduces a preprocessing step that transforms user code to enable speculative execution. SpecHLS builds on the observation that *each conditional in the input code is a potential speculation candidate.* Whenever there is a conditional branch in the input code where one execution path is longer than the other, speculating on the fast path may provide performance improvements. Figure 3 gives an overview of the SpecHLS code transformations applied to our example. The input code is parsed and converted into an internal representation (GSSA), described in Section 2.3. Conditional branches are then checked and selected as speculation candidates if they provide an II improvement during the *speculation exploration* phase. Control-and dataflow are then decoupled by inserting a Finite State Machine (FSM), `nextstate`, that controls the program's execution. Rollback mechanisms are inserted to handle mispeculations. The resulting code, which exhibits explicit reuse distances, is shown on the right-hand side. It is sent to an HLS toolchain that acts as the hardware synthesis backend for the entire flow [8, 13]. This transformation process allows control-flow heavy input codes to produce efficient execution schedules, achieving the coveted $II = 1$ when there is no mispeculation.

The *memory speculation* pass introduced in this paper kicks in when speculation involves interactions with memory. It enables speculative High-Level Synthesis in codes such as the ones presented in Part *b* and *c* of Figure 2.

2.3 Circuit IR

In this paper, we build on top of SpecHLS and the underlying GeCoS compiler framework [11] to bring memory speculation to HLS. We manipulate a variant of the Gated-SSA (GSSA) [38, 39] representation. The latter replaces the SSA form's φ-nodes by *gating nodes*. The type of gating node that replaces a given φ-node depends on the context in which the

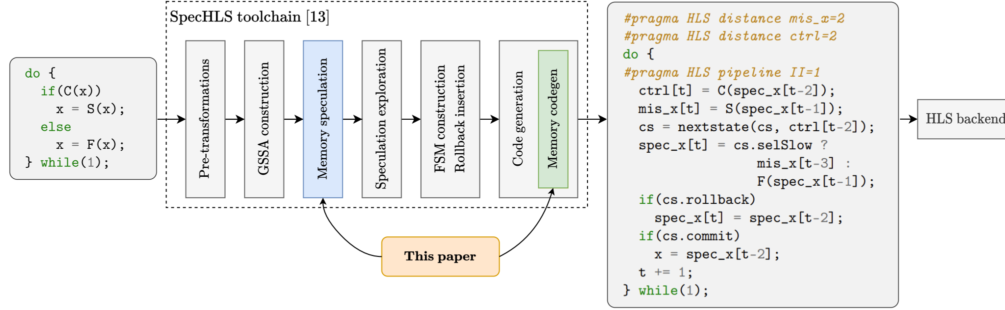

Figure 3. SpecHLS code transformation flow. This paper extends SpecHLS by introducing a memory speculation pass and its accompanying code generation extension.

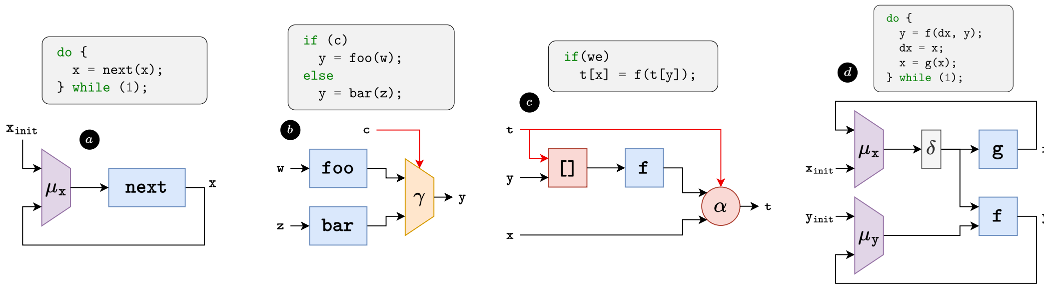

Figure 4. Extended Gated-SSA operators. μ-nodes *a* represent loop headers, γ-nodes *b* encode control-flow decisions, α-nodes *c* serve as array updates, and δ-nodes *d* act as delays on a hardware path.

φ-node appears. The following gating nodes are of particular interest to us:

- μ-nodes are placed in loop headers. They take three arguments, $\mu(p, i_x, l_x)$, where p is the loop exit condition, i_x is the initial value of variable x, and l_x is the value of x after a loop iteration.
- γ-nodes are placed at joining points in the control-flow graph, such as the end of conditional structures. They act as traditional φ-nodes and encode the predicate that determines which value is to be selected when execution reaches them. We denote them $\gamma(p_x, x_0, x_1, \ldots)$, with p_x the predicate and $x_0, x_1, \ldots$ the possible values for variable x.

Gated-SSA allows us to work with fully-predicated φ-nodes, in the form of μ-and γ-nodes. This predication outlines speculation opportunities in the IR. Each γ-node can then be seen as a potential speculation candidate [8].

For the needs of this paper, we extend GSSA with δ-nodes. δ-nodes are akin to *delays* introduced on the hardware path in which they appear. They can be seen as micro-architectural delays in the generated hardware, while μ-nodes can be thought of as iteration delays. δ-nodes access values from previous iterations in the alias detection logic described in Section 3.4. They increase the reuse distance on the path where they are inserted.

Lastly, α-nodes act as store operations into arrays. They take an array, a value, and an index as inputs and produce the updated array. Figure 4 illustrates the GSSA operators we manipulate in this paper. We refer to this representation as an *Instruction Dependency Graph* (IDG). To simplify our examples, the loop exit condition is omitted from μ-nodes.

We extend SpecHLS [13] with a set of transformations that operate on Gated-SSA to expose *memory speculation* opportunities.

2.4 Memory Accesses in Speculative HLS

Hardware support for speculative execution has been explored by previous work in the absence of stateful components, such as memory buffers [8, 13, 17]: a stateless computation that encounters a mispeculation needs only its inputs to be reset to their initial value to handle the mispeculation. The problem of handling mispeculations becomes more intricate when components can retain state that may have been produced by speculative operations.

Existing speculative HLS toolchains choose to either limit speculation to regions of the circuit that do not interact with memory [17], or to treat arrays as immutable values [13]. In the latter approach, α-nodes (*i.e.* array updates) take an array as input and return an updated copy of the array. While the resulting circuit would be correct and behave properly in

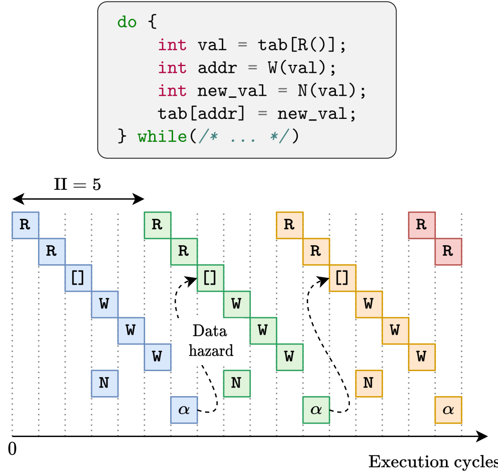

Figure 5. Data-dependent memory access, along with its statically inferred execution schedule. An inter-iteration memory dependency (dotted arrow) prevents efficient overlap of iterations through traditional loop pipelining.

case of mispeculations if such value semantics were to be preserved for code generation, the generated hardware would be far from optimal. It would require array copy operations each time a value may be written in an array, and the cost of storing array states for potential rollbacks in case of a mispeculation would be significant. While the SpecHLS paper does not mention how memory accesses are lowered to synthesizable code, array-as-value semantics is not preserved in the code generation process.

In this paper, we argue that a similar approach can be used to manipulate speculation on memory dependencies at a high abstraction level (Section 3), and that efficient code generation can be derived from such a representation (Section 4) by automatically inferring store queues for arrays that interact with speculative execution. We aim at producing high-throughput schedules for accelerators that interact with memory, such as the ones depicted in examples ⓑ and ⓒ from Figure 2. We present a unified memory dependency handling framework, which generalizes speculative HLS to interact with stateful memory components (example ⓑ), and allows us to speculate on the absence of memory aliases to produce aggressive hardware schedules (example ⓒ).

3 Speculating on Memory Dependencies

Speculating on memory dependencies allows data hazards to be handled in an efficient way, leading to tight operator scheduling in the final hardware. This section starts by describing memory dependency handling in traditional HLS flows (Section 3.1) before illustrating the impact of runtime memory disambiguation logic on the throughput of generated hardware (Section 3.2). Section 3.3 shows how we can integrate memory dependency information into our intermediate representation and Section 3.4 describes our memory speculation framework. Finally, Section 3.5 presents our complete memory speculation mechanism, which is able to handle RAW, WAR and WAW dependencies.

3.1 Memory Dependency Handling in HLS

Data hazards can severely impact the scheduling efficiency of traditional HLS tools. When a possible memory alias is detected in a loop body, the hardware needs to be synthesized while considering the worst case scenario, similarly to how instructions need to be scheduled for the worst case in a VLIW compiler backend [21]. This worst case can have a tremendous impact on the value of II that can be achieved by a hardware design, since the hardware must assume that an alias can happen at every iteration of the loop. Figure 5 illustrates a simple data-dependent memory access example that leads to poor hardware synthesis results. The dependency limits the pipelining capabilities of the HLS toolchain, especially in the presence of non-negligible address computation delays. The bottom part of Figure 5 illustrates the execution schedule that can be inferred by HLS, assuming that `R`, `W`, and `N` take respectively two, three and one cycle to execute. We denote memory loads by `[]` and memory writes by α and further assume that memory accesses take one cycle. We observe that the loop pipelining transformations manage to partially overlap the execution of consecutive iterations of the loop. However, the memory dependency denoted by the dotted arrow in Figure 5 prevents further overlapping because of the potential data hazard if the read and write addresses are the same.

Modern HLS toolchains use static alias analysis passes to identify memory dependencies inside loops. This approach can help identify false dependencies in some cases, with advanced analyses and transformations based on polyhedral compilation techniques [30, 33]. Additionally, HLS users

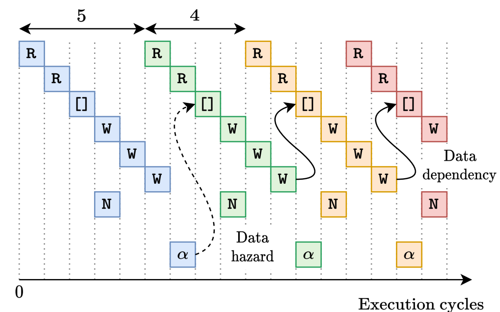

Figure 6. Execution schedule of the example code in Figure 5 with runtime memory disambiguation. The load from memory depends on the result of the address computation `W`, thereby limiting the minimum achievable II.

can override the results of alias analysis with code annotations. The Vitis HLS dependence pragma is an example of such annotation, which allows the user to specify the type of dependency (RAW, WAR or WAW) and the inter-iteration distance between potentially aliasing array accesses [2]. These approaches can cover a wide variety of memory dependency scenarios, but they cannot account for situations where the dependency distance cannot be bound statically. Our approach handles the latter case as well as the former one, leading to higher throughput in the generated hardware at a small area overhead cost.

3.2 Improving HLS Scheduling in the Presence of Memory Dependencies

Existing work has focused on runtime memory access disambiguation [1, 9, 17, 27, 30]. The latter avoids long memory accesses when possible. However, even with runtime alias detection, the address computation latency may still hinder performance. An example is given in Figure 6, with an alias occurring only between the first and second iterations of the loop. The disambiguation logic leads to a dynamically scheduled execution [18], and an improvement in the number of cycles per iteration (CPI). If data hazards represent a proportion $p < 1$ of memory accesses during the entire execution, we now have CPI $= 5p + 4 \times (1 - p) < 5$. In this paper, we show how to further improve the performance of HLS code that exhibits memory dependencies by leveraging speculation.

3.3 Memory Dependencies in Gated-SSA

The SpecHLS flow on top of which we build our memory speculation framework provides limited support for memory operation ordering. For example, while store order is preserved through the immutable array representation, the relative order of stores and preceding loads is not preserved in the GSSA representation. This behaviour can cause the generated circuit to misbehave if a Write-after-Read (WAR)

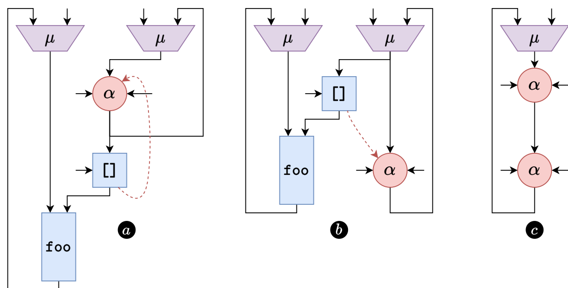

Figure 7. Memory dependencies in extended Gated-SSA. The dotted arrow represents a load-store dependency that needs to be honored during code generation. The immutable array representation already ensures strong store-store ordering.

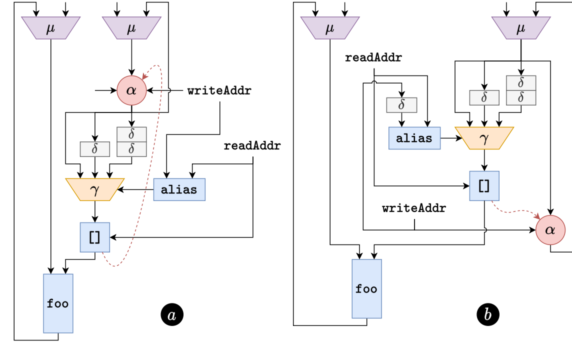

Figure 8. Speculating on a Read-after-Write (a) (see Figure 7 (a)) and Write-after-Read (b) (see Figure 7 (b)) dependency. The shortest path exposes two δ-nodes.

dependency is violated. We extend the SpecHLS intermediate representation to allow for the expression of explicit memory dependencies using non-dataflow edges in the IR graph (see Figure 7).

We note that this extension enforces strong load-store and store-load ordering, but not load-load ordering: loads that in a given order in the input code may be generated out-of-order in the transformed code. The reordering of load operations does not change the behaviour of the circuit and allows us to not over-constrain operation scheduling.

3.4 Exposing Speculation Opportunities

As noted in Section 2.3, speculation opportunities are modeled in our IR as γ-nodes. The latter correspond to control-flow decisions, so we need to recast the memory dependency problem as a control-flow decision problem. This transformation corresponds precisely to runtime alias detection. We introduce a GSSA transformation that materializes runtime alias detection in the IR. The goal of such a transformation is to increase the dependency distance between iterations, allowing the HLS backend to synthesize deeper pipelines. The alias detection window depth w_d (*i.e.* the number of previous iterations to check for aliases with) is configured at compile-time by the user, but it could also be determined through a combination of application profiling and design space exploration. The latter is out of the scope of this paper.

3.4.1 Read-after-Write. We start by considering the case of intra-iteration Read-after-Write dependencies. Before each load, we insert a γ-node that operates on the array *value* from which we would like to read (Figure 8 (a)). The inputs to the alias detection γ-node are increasingly delayed versions of the array (*i.e.* from the current and previous iterations of the loop), and the decision is controlled by an `alias` node. The latter manages an internal buffer of addresses whose size will be determined during the code generation phase (see Section 4). In case of an alias, then the alias node selects the non-delayed input of the γ-node, stalling the load operation

until the α-node completes its operation. Otherwise, if there is an alias $n \geqslant 1$ iterations before the load, then the input of the γ-node with n δ-nodes is selected.

Selecting an array value that comes out of a sequence of n δ-nodes can be seen as ignoring the last n stores to this array. A direct consequence of this observation is that the alias detection γ-node exhibits a slow path on its non-delayed input, and increasingly fast paths on its delayed inputs. This imbalance in path length through the γ-node naturally leads itself to speculation [8]. We speculate that there are no aliases, selecting the most delayed version of the array. If there was an alias $i < w_d$ iterations ago, then we roll back the computation and select the i-th input of the γ-node. Note that, since we speculate, we avoid the issues with runtime memory disambiguation latencies illustrated in Figure 6: a new loop iteration can start every cycle, and the mispeculation handling logic takes care of rolling back any potentially erroneous values. This process is handled transparently by the speculation insertion mechanism in SpecHLS.

3.4.2 Write-after-Read. Let us now consider the case of intra-iteration Write-after-Read dependencies. Similarly to the RAW dependency case, we insert an alias detection mechanism before each memory load. The main difference with the RAW example described above is that write addresses need to be delayed before entering the alias node. Figure 8 ⓑ illustrates this situation.

3.4.3 Write-after-Write. Any Write-after-Write dependency is encoded in our GSSA representation by an edge between α-nodes. This edge is taken into account by the code generation phase (Section 4) to keep array updates in order.

3.5 Generalizing Memory Speculation

The examples detailed in Section 3.4 form the basis of our proposed memory speculation framework. They are the foundation from which we derive a mechanized procedure to insert memory speculation opportunities in any input code. The main idea behind this procedure is given by Figure 9, and is described in the remainder of this section.

In the following, $n_\alpha(s)$ denotes the number of α-nodes that operate on s in the GSSA representation. For each load l_s from s, $P_\alpha(l_s)$ (resp. $S_\alpha(l_s)$) denotes the set of α-nodes that precede (resp. follow) l_s. Once all memory dependencies are explicitly represented in GSSA, we traverse our IR looking for memory load operations for each array symbol s in our input program. We insert the alias detection logic before each load l_s from s. The structure of the alias detection mechanism can be summarized as follows:

- the `alias` node has $n_\alpha + 1$ entries, one for l_s's read address, and one for each write address in the SCC. Write addresses from nodes in $P_\alpha(l_s)$ are directly linked to the `alias` node, while addresses from nodes in $S_\alpha(l_s)$ are connected through a δ-node (not shown in Figure 9).
- the γ-node has w_d groups of inputs, with each successive group adding a δ-node to its inputs. The latter inputs correspond to the array values produced by all the α-nodes operating on s in the SCC. Similarly to the write addresses, additional δ-nodes are inserted for array values produced by nodes in $S_\alpha(l_s)$.

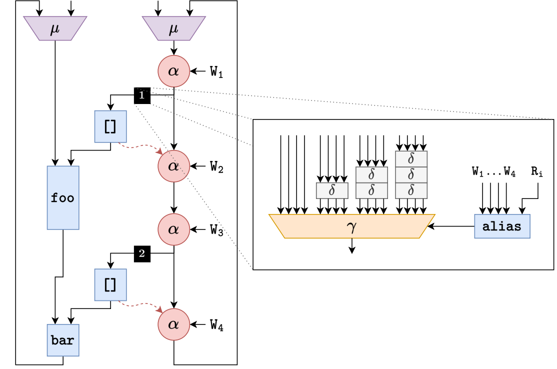

Figure 9. Generalizing the memory speculation mechanism to account for multiple memory dependencies. δ-nodes on the inputs of the alias detection γ-node are grouped for readability purposes.

4 Code Generation

Previous sections have shown how we can introduce a generalized memory speculation mechanism in the SpecHLS toolchain. In this section, we discuss the code generation aspect of memory speculation, which requires special care to make it amenable to efficient synthesis by commercial HLS toolchains. Section 4.1 starts by highlighting the semantic gap that exists between the by-value array semantics that we have been working with up to this point, and the HLS-friendly semantics that we need to generate for our backend. Section 4.2 introduces a couple of common optimizations used in HLS code that are leveraged by our code generation backend. Section 4.3 gives an overview of the code we generate to handle speculation on memory accesses, in particular regarding store queues, and Section 4.4 focuses on our store queue parameter inference pass.

4.1 Code Generation for Memory Speculation

In order for our speculation flow to generate efficient hardware, we need to lower the level of abstraction at which we manipulate arrays. So far, arrays have been considered as immutable values, with α-nodes producing a new updated value after each store operation. While this representation is convenient for high-level transformations, it is unpractical for realistic hardware synthesis. In the code generation

phase, we lower this abstraction to a set of store queues and an underlying array. This step represents an important shift in perspective when considering arrays, as they go from a localized state to a global state that needs to be managed across the entirety of the loop we are trying to pipeline.

The following section highlights the challenges that arise from such a shift of perspective, as we need to guarantee that (i) a read from the global state has to reflect the previously localized state of the array, and needs to take into account potentially pending stores to the array that may not have been committed yet; (ii) a write to the array needs to update the pending write buffer while waiting for the confirmation that the value to be written is correct; (iii) a value is committed to the global state only if is correct, and the array contents (as well as all auxiliary data structures) are rolled-back to their previous state in case of a mispeculation.

4.2 Efficient Code Generation for HLS

Unlike traditional compilers, HLS toolchains target custom hardware with little to no constraints on the type, size, placement and number of operators, registers and memory. Typical HLS targets include ASICs (Application-Specific Integrated Circuits) and FPGAs (Field-Programmable Gate Arrays). We focus on the latter target for our code generation.

Two of the most prominent optimizations in HLS are *loop unrolling* and *array partitioning*. Iterations of an unrolled loop can be mapped to independent hardware operators, maximizing parallel execution at the expense of design area. Array partitioning maps an array in the input HLS code to memory banks or even registers for each array element in the final hardware. This transformation increases the amount of read/write ports on the memory, and can be used to improve execution throughput. However, as for loop unrolling, this optimization is done at the cost of some hardware area.

Because the optimizations discussed in this section introduce a tradeoff between hardware area and performance, HLS toolchains do not apply them automatically. Instead, they are user-guided optimizations, with hardware designers annotating their C++ code using `pragmas` to indicate which parts to optimize. The following section highlights some of the places where we generate annotations in our code to improve hardware generation by the HLS backend.

4.3 Code Generation Structure

For each array for which we inserted the speculation logic described in Section 3 code, we need to generate code for five different functions: `read`, `update`, `commit`, `rollback` and `alias`. They correspond respectively to array reads, α-nodes, commit nodes, rollbacks and the alias detection logic from our speculative IR. These functions operate on plain C pointers representing arrays, and interact with global store queue structures. The following lays out the latter structures and described the operation of each function in more detail. We only discuss operations for a single array, but the code is easily extended to multiple arrays.

4.3.1 Store Queue Structure. Each store queue holds a list of pending values to be stored and a list of their corresponding addresses, with a valid bit for each entry. The size of each of these lists, `maxPendingStores` and `maxPendingAddr`, are computed by a procedure described in Section 4.4. We create a separate partial store queue for each α-node that operates on the array, with the union of all these partial queues forming a store queue structure similar to the one found in processors. Doing so allows us to have fine grain control in the alias detection while still making it easy for the HLS toolchain in the backend to optimize our code.

4.3.2 Reading Values. To read values from an array, we need to either load it from the contents of the array, or index into our store queue structure if an update occurred at the current read address. This operation is achieved by providing our reading stub the value of the read address alongside the array pointer, as well as a list of value ranges. The latter correspond to the value windows to consider in each partial store queue for the current read operation. We do not traverse the entire store queue to avoid data dependency violations, as some values from later iterations may already have been written to the store queue by speculative execution. The parameters of each store queue window are computed by the procedure described in Section 4.4.

4.3.3 Updating Array Contents. Array updates that happen through α-nodes in our intermediate representation are lowered to a function parameterized by the index of the α-node for access to partial store queues, as well as the maximum number of pending values and addresses. Each α-node in our intermediate representation is replaced by a call to this function, passing its unique identifier as the first template parameter. We shift the contents of the store queue to make room for the newly written value and insert the latter in the partial store queue that corresponds to the current α-node. The `we` function argument corresponds to a write-enable signal that is cleared by the speculation Finite State Machine (FSM) if a mispeculation is being handled and invalid values are currently propagating through the circuit.

4.3.4 Committing Stores. Once the value computed by a speculative operation is detected as being correct by the speculation logic, the contents of the partial store queues are committed to the underlying array. For each α-node in the code, we check if the oldest value in its partial store queue is valid (*i.e.* the valid bit was set by the update function) and, if so, we write it to the array at the address held in the queue.

4.3.5 Rolling Back. When a mispeculation occurs in the SCC, incorrect data may have already been written to the array's store queue. This incorrect data can either be a value

or an address computed by a speculative operation as a result of a wrong speculation. In such a case, our store queue can simply be rolled back by marking all stores that happened during the wrong speculative execution as invalid. This function is called directly by the speculative FSM.

4.3.6 Detecting Aliases. The alias detection logic presented in Section 3.5 is rewired during the code generation phase. The output of the alias node is now linked to the FSM and informs the speculative execution process of potential mispeculations. Instead of operating on the write addresses, as illustrated in Figure 9, the final alias detection function operates on the partial store queues associated to all α-nodes. It checks if the read address of its corresponding load operator aliases with any store operation registered in the partial queues, and produces an integer value indicating the distance (in terms of memory operations) at which the closest alias occurred.

4.4 Inferring Store Queue Parameters

In our memory dependency handling framework, stores to memory are handled by a set of partial store queues. The latter are parameterized by a store window to ensure temporal consistency during the execution (see Section 4.1). The size of each partial store queue as well as the window parameters are determined by a depth propagation pass, which we chose to encode as a type inference pass on our IR. In the following, we consider the case where there is only one α-node for each array in the input code (Section 4.4.1) before showing how to generalize our approach to multiple array updates (Section 4.4.2).

4.4.1 Single Array Update. We denote the type of an array with elements of type τ as $\mathcal{A}(\tau)$. We add two additional parameters to this type, w and d, to encode the length and the end of the store queue window (*i.e.* the number of values to *discard* at the end of the store queue) that is valid at each node. Consequently, if a node N has type $\mathcal{A}(\tau, w, d)$, then all stores in the queue in the range $[0; w-d]$ are valid when executing N.

The store queue depth inference pass is a forward analysis along dataflow edges, ignoring the back-edges on μ-nodes. The maximum depth of the store queue is computed as one plus the maximum of all rollbacks that can occur for a given array. Intuitively, the depth of the rollback operators is given by the length in cycles of the computations that determine if a speculation is correct or not. During the execution of such an operation, values may be written to the store queue at each execution cycle. These values may be invalid if the current speculation happens to be incorrect. Thus, we need to keep at least `maxRBPending` + 1 values in the store queue at each time, since a mispeculation may require rolling back at most `maxRBPending` values in the queue and restoring the previous state of the array. The user can define a maximum distance (in terms of iterations) at which to look for aliases, Δ_U.

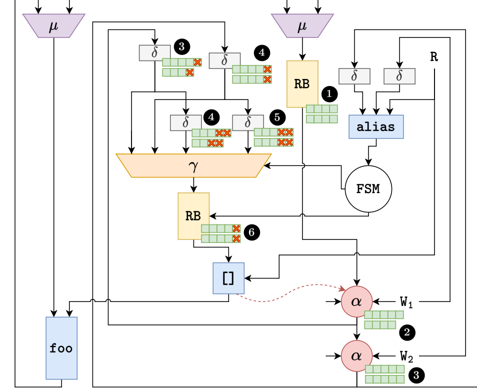

Figure 10. Store queue structure inference pass. The circled numbers show the successive steps followed by the inference algorithm.

The store queue depth inference pass operates on a worklist of nodes, starting with all μ-nodes in the SCC that correspond to arrays. The entire procedure traverses the intermediate representation exactly once, propagating store queue parameters in the `propagate` function. The latter operates according to the following inference rules:

$$\text{Mu}\ \frac{x : \mathcal{A}(\tau) \quad p : \texttt{bool} \quad i_x : \mathcal{A}(\tau)}{\mu(p, i_x, x) : \mathcal{A}(\tau, \texttt{maxRbPending} - 1, 0)}$$

$$\text{Alpha}\ \frac{x : \mathcal{A}(\tau, w, d) \quad i : \texttt{int} \quad e : \tau}{\alpha(x, i, e) : \mathcal{A}(\tau, w + 1, d)}$$

$$\text{Gamma}\ \frac{c : \texttt{bool} \quad e_0 : \mathcal{A}(\tau, w_0, d_0) \ \ldots \ e_n : \mathcal{A}(\tau, w_n, d_n)}{\gamma(c, e0, \ldots, e_n) : \mathcal{A}(\tau, \max_i w_i, \min_i d_i)}$$

$$\text{Mux}\ \frac{c : \texttt{bool} \quad e_0 : \mathcal{A}(\tau, w_0, d_0) \ \ldots \ e_n : \mathcal{A}(\tau, w_n, d_n)}{\text{mux}(c, e0, \ldots, e_n) : \mathcal{A}(\tau, \max_i w_i, \min_i d_i)}$$

$$\text{Delta}\ \frac{x : \mathcal{A}(\tau, w, d)}{\delta(x) : \mathcal{A}(\tau, w, d + 1)}$$

$$\text{Rollback}\ \frac{x : \mathcal{A}(\tau, w, d)}{\text{RB}(x) : \mathcal{A}(\tau, w, d)}$$

The store queue window is initialized at μ-node, where the depth inference pass starts. When encountering an α-node, an additional pending store is added to the store queue window. Multiplexers and γ-node are treated in a similar fashion: we need to consider the most pending stores from all of the node's inputs, and we need to discard the least amount of elements. The latter rule encodes the interval union of all input store queue windows. δ-nodes add a discarded store, as they delay the execution by one cycle: stores that are valid before a delay will only be visible at the next cycle for operators dominated by the δ-node. Finally, rollbacks pass the type of their input to their output, as they only affect the maximum size of the store queue windows. These inference

Table 1. Performance and area results for a set of benchmarks that exhibit dynamic or otherwise hard to handle memory dependencies. We observe an average speed-up equal to the average increase in surface area of the circuit.

	Baseline					**Speculative**							
Benchmark	II	F_{max} (MHz)	LUT+FF	BRAM	DSP	II	F_{max} (MHz)	CPI	LUT+FF	BRAM	DSP	**Misp. (%)**	**Speed-up**
SimpleRAW	3	101	579	1	0	1	101	1.2	1584 (2.7×)	1	0	6	2.5×
SimpleWAR	3	101	903	1	0	1	101	1.5	2035 (2.3×)	1	0	3	2.0×
SimpleWAW	7	101	2955	0	0	1	101	1.7	5475 (1.9×)	0	0	9	4.1×
DoubleRAW	9	101	1376	0	2	1	80	1.4	6380 (4.7×)	0	2	17	5.1×
Histogram	4	122	723	0	2	1	101	1.3	2772 (3.8×)	0	2	16	2.5×
SKA-Gridding	4	122	1170	0	2	1	101	2.5	1480 (1.3×)	0	2	40	1.3×
ALU	2	110	608	0	0	1	99	1.3	1336 (2.2×)	0	3	15	1.4×
KulischAccum	3	197	715	2	0	1	141	1.2	959 (1.3×)	2	0	3	1.8×
Floyd-Warshall	6	122	1631	0	0	1	82	1.1	8192 (5.0×)	0	0	11	3.7×
Gauss	6	122	994	1	5	1	87	1	4177 (4.2×)	3	3	0	4.3×
BNN	5	145	330	2	0	1	137	1.1	2379 (7.2×)	2	0	10	4.3×

rules cover all the node types that may interact with arrays in our IR, except for loads. Array reads do not update the array type, but they make use of the type of their array input in the generated code described in Section 4.1.

4.4.2 Multiple Array Updates.

The type inference pass presented in the previous section can be extended to multiple array updates by replacing the type parameter w with a list, where each element of the list corresponds to a different α-node. Only the Alpha inference rule needs to be updated, replacing it by

$$\text{Alpha}\frac{x : \mathcal{A}(\tau, w, d) \quad i : \texttt{int} \quad e : \tau}{\alpha_j(x, i, e) : \mathcal{A}(\tau, w[w_j \mapsto w_j + 1], d)},$$

with j the index of the α-node, and $w[w_j \mapsto w_j + 1]$ denoting the in-place update of the j-th element of w.

Figure 10 illustrates the operation of the store queue parameter inference pass on an example involving a load and two stores. The store queue windows are represented for each α-node by a sequence of squares. The output type of a node is attached to its lower-right corner. The length of the sequences corresponds to the value of the elements of the w type parameter list. The d parameter is represented by crossed-out elements in those windows.

5 Experimental Results

Application-specific hardware accelerators are often comprised of loop kernels and do not encompass entire applications, making traditional benchmarks such as SpecInt irrelevant. Although there exist HLS specific benchmarks [14, 43], they are focused on kernels with regular access patterns, as they carry the legacy of traditional HLS application domains.

To address these issues, we make the choice to evaluate our memory speculation framework on a selected set of benchmarks with data-dependent memory dependencies from previously published work [1, 8], as well as statically determinable but infrequent dependencies that prevent pipelining as discussed by Liu et al [27]. In both cases, traditional HLS tools fail to find a satisfying operation schedule and produce hardware with II > 1. *SimpleRAW*, *SimpleWAR*, *SimpleWAW*, and *DoubleRAW* are synthetic benchmarks that exhibit basic memory dependency patterns inside of a loop. *SimpleRAW* (resp. *SimpleWAR*, and *SimpleWAW*) is similar to Figure 7 ⓐ (resp. Figure 7 ⓑ, and Figure 7 ⓒ). *Histogram*, *SKA-Gridding*, *ALU*, and *KulischAccum* exhibit data-dependent memory accesses that cannot be determined statically. *Floyd-Warshall* and *Gauss* exhibit an inter-iteration memory dependency for certain iterations of the loop. Consequently, HLS tools have to generate a pessimistic schedule while pipelining the loop. *BNN* contains both a data-dependent memory access and an inter-iteration memory dependency.

In all our benchmarks, we choose the minimal value of Δ_U such that II = 1. This value is specified to our toolchain through a `pragma` annotation in the input code. We use Vitis HLS 2021.2 to perform the High-Level Synthesis, with an XC7A200 as the target FPGA. Table 1 shows performance and area results for our benchmark set. The baseline (with no speculation applied) is shown on the left-hand side, followed by its speculative counterpart. Mispeculation rates and relative speed-ups are given at the end of each row. The CPI (Cycles Per Iteration) value that we give for the speculative design can be seen as an *effective* II. It is equal to II for the baseline, and is linearly correlated with the mispeculation rate for the speculative version. The hardware utilization is depicted through the utilization of lookup tables and flip-flops (LUT+FF), memory blocks (BRAM), and

hardware multiply/add blocks (DSP). Note that some arithmetic operations can either be mapped to DSP or to LUT+FF elements, which explains some of our area results. For example, the speculative *Gauss* accelerator uses less DSPs than the baseline because some arithmetic operations have been mapped to LUT+FF by the HLS backend's heuristics.

We note that the proposed approach reduces the static II for each application. To measure the speed-up, we compare the CPI achieved by the baseline and the speculative versions of our benchmarks, but also the maximal frequency of the generated hardware. The additional control inserted to detect aliases and handle mispeculation can have a negative impact on the maximal frequency. However, the important gains on the CPI side leads to a speed-up ranging from 1.3× to 5.1×. It is important to note that for synthetic benchmarks and data-dependent applications, the mispeculation rate depends on the data used and may affect the speed-up. However, previous studies on speculative loop pipelining show that even if we always mispeculate, the CPI is never worse than the pessimistic static schedule [8]. Only the F_{max} reduction may reduce the performance of the generated hardware.

For most benchmarks, we observe an area increase that closely matches the speed-up. Part of this overhead is due to the additional logic used to delay pending writes, detect aliases, and correctly handle mispeculations. The cost of this logic depends on the Δ_U used to achieve II = 1, and on the number of read/write operations in an iteration of the kernel. However, note that reducing the II also negatively impacts the area utilization, as it minimizes the possibility of resource sharing. No resource sharing is possible when II = 1.

6 Related Work

Speculation is a fundamental aspect of high-performance computing that has implications from compiler design all the way to low-level hardware design. Some specialized compiler frameworks leverage coarse-grain speculative execution by partitioning programs into sets of speculative threads [3, 34, 40]. This behavior leads to memory hazards between threads that need to be resolved at runtime. Speculative Decoupled Software Pipelining (SpecDSWP) supports speculative memory rollbacks by leveraging a versioned memory [40]. All these approaches leverage speculation on existing hardware, while our approach is aimed at custom hardware synthesis.

Speculation is very efficient at improving performance, but it is equally difficult to implement. Automated Pipeline synthesis tools attempt to streamline speculation insertion from a high-level description of the pipeline. Piper [16] and T-Piper [32] automatically generate speculative pipelines from a high-level hardware description, with T-Piper allowing for automatic design space exploration of data hazard resolution strategies.

Several works have applied varying degrees of speculation to High-Level Synthesis flows, from automatic branch prediction synthesis [15, 22] to generalized speculative execution [8, 13, 17]. The latter often incorporate some form of limited memory speculation. Josipović et al. [17] insert *save* and *commit* operators at the boundary of dataflow circuit regions where speculative computations are to take place. These regions always end before any *store* unit can interact with memory, and no speculative token can escape to memory before the result of the computation is committed. Gorius et al. [13] only handle simple read-after-write loop-carried dependencies in a speculative context. Furthermore, their approach only allows for a single array update to happen in a loop iteration for each array. However, it is general enough to allow for fully automated design space exploration of RISC-V soft-cores targetting FPGAs [12].

Dynamic memory hazard resolution in HLS is an active research area, as it provides a wider range of application to HLS. Alle et al. [1] use source-to-source manipulations to insert runtime alias detection and transform loops with loop-carried dependencies, allowing them to be pipelined by HLS toolchains. Dai et al. [7] insert dynamic hazard resolution logic at compile-time and handle data hazards at runtime through memory port arbitration and squash-and-replay. Several authors have also leveraged polyhedral compilation techniques to insert compile-time/runtime alias detection logic and stall the execution pipeline when an alias occurs [26, 27, 30]. Some of the latter approaches tend to duplicate hardware to handle memory dependencies [26, 27], whereas our approach's area is limited to the speculation and alias detection logic.

The issue of store queue sizing in dataflow circuits has been covered by Liu et al. [25] as well as Elakhras et al. [9]. The latter leverage basic-block analysis to tune the size of the load-store queue inserted into their dataflow designs. This approach has only been shown to be applicable in the context of *dynamic* dataflow circuits. Speculative execution would likely require additional logic in the store queue, especially to handle multiple interacting speculations, which are only briefly discussed by previous work on dataflow circuits [17].

7 Conclusion

Speculative High-Level Synthesis brings customizable speculative behaviour to hardware accelerators, and opens up a wide range of possible accelerator designs for control-flow dominated applications. In this paper, we propose a unified memory dependency handling framework for speculative HLS and show that we can generate hardware from C++ code in the presence of speculative memory accesses. Our memory speculation framework allows for handling of memory accesses in speculative loops by rolling back incorrect stores, speculating over intra-and inter-iteration dependencies in Read-after-Write, Write-after-Read and Write-after-Write scenarios, and generate C++ code that is compatible with commercial HLS toolchains.

References

[1] Mythri Alle, Antoine Morvan, and Steven Derrien. 2013. Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis. In *Proceedings of the 50th Annual Design Automation Conference* (Austin, Texas) *(DAC '13)*. Association for Computing Machinery, New York, NY, USA, Article 51, 10 pages. https://doi.org/10.1145/2463209.2488796

[2] AMD Xilinx. 2023. Vitis High-Level Synthesis User Guide (UG1399). https://docs.xilinx.com/r/en-US/ug1399-vitis-hls. Accessed 25-08-2023.

[3] Anasua Bhowmik and Manoj Franklin. 2002. A General Compiler Framework for Speculative Multithreading. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Winnipeg, Manitoba, Canada) *(SPAA '02)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/564870.564885

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[7] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. 2017. 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* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 189–194. https://doi.org/10.1145/3020078.3021754

[8] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. 2020. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (Nov. 2020), 4229–4239. https://doi.org/10.1109/TCAD.2020.3012866

[9] Ayatallah Elakhras, Riya Sawhney, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. 2023. Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 39–45. https://doi.org/10.1145/3543622.3573050

[10] John P. Elliott. 1999. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design.* Kluwer Academic Publishers, USA.

[11] Antoine Floc'h, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic L'Hours, Nicolas Simon, Steven Derrien, François Charot, Christophe Wolinski, and Olivier Sentieys. 2013. GeCoS: A framework for prototyping custom hardware design flows. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 100–105. https://doi.org/10.1109/SCAM.2013.6648190

[12] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. 2022. Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis. In *2022 International Conference on Field-Programmable Technology (ICFPT)*. 1–6. https://doi.org/10.1109/ICFPT56656.2022.9974478

[13] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. 2022. SpecHLS: Speculative Accelerator Design Using High-Level Synthesis. *IEEE Micro* 42, 5 (2022), 99–107. https://doi.org/10.1109/MM.2022.3188136

[14] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1192–1195. https://doi.org/10.1109/ISCAS.2008.4541637

[15] U. Holtmann and R. Ernst. 1995. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *Proceedings the European Design and Test Conference. ED&TC 1995*. 550–556. https://doi.org/10.1109/EDTC.1995.470346

[16] I.-J. Huang and A.M. Despain. 1993. Hardware/software resolution of pipeline hazards in pipeline synthesis of instruction set processors. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. 594–599. https://doi.org/10.1109/ICCAD.1993.580120

[17] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative Dataflow Circuits. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 162–171. https://doi.org/10.1145/3289602.3293914

[18] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/3174243.3174264

[19] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. Motivation for and Evaluation of the First Tensor Processing Unit. *IEEE Micro* 38, 3 (2018), 10–19. https://doi.org/10.1109/MM.2018.032271057

[20] David W. Knapp. 1996. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler.* Prentice-Hall, Inc., USA.

[21] M. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 318–328. https://doi.org/10.1145/53990.54022

[22] Vianney Lapotre, Philippe Coussy, Cyrille Chavet, Hugues Wouafo, and Robin Danilo. 2013. Dynamic branch prediction for high-level synthesis. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–6. https://doi.org/10.1109/FPL.2013.6645540

[23] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[24] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[25] Jiantao Liu, Carmine Rizzi, and Lana Josipović. 2022. Load-Store Queue Sizing for Efficient Dataflow Circuits. In *2022 International Conference on Field-Programmable Technology (ICFPT)*. 1–9. https://doi.org/10.1109/ICFPT56656.2022.9974425

[26] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. 2017. Run fast when you can: Loop pipelining with uncertain and non-uniform memory dependencies. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 126–130. https://doi.org/10.1109/ACSSC.2017.8335151

[27] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. 2018. Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 9 (Sept. 2018), 1802–1815.

https://doi.org/10.1109/TCAD.2017.2783363

[28] D. MacMillen, R. Camposano, D. Hill, and T.W. Williams. 2000. An industrial view of electronic design automation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19, 12 (Dec. 2000), 1428–1448. https://doi.org/10.1109/43.898825

[29] Modular. 2023. Mojo – a new programming language for all AI developers. https://www.modular.com/mojo. Accessed 01-09-2023.

[30] Antoine Morvan, Steven Derrien, and Patrice Quinton. 2013. 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* 32, 3 (March 2013), 339–352. https://doi.org/10.1109/TCAD.2012.2228270

[31] R. Nozal and J.L. Bosque. 2021. Exploiting Co-execution with OneAPI: Heterogeneity from a Modern Perspective. In *Euro-Par 2021: Parallel Processing*. https://doi.org/10.1007/978-3-030-85665-6_31

[32] Eriko Nurvitadhi, James C. Hoe, Timothy Kam, and Shih-Lien L. Lu. 2011. Automatic Pipelining From Transactional Datapath Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 3 (2011), 441–454. https://doi.org/10.1109/TCAD.2010.2088950

[33] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 4–13. https://doi.org/10.1145/125826.125848

[34] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. 2005. Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 269–279. https://doi.org/10.1145/1065010.1065043

[35] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. 2021. Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 600–615. https://doi.org/10.1145/3445814.3446723

[36] A. Sangiovanni-Vincentelli. 2003. The tides of EDA. *IEEE Design & Test of Computers* 20, 6 (Nov. 2003), 59–75. https://doi.org/10.1109/MDT.2003.1246165

[37] Siemens. 2023. Catapult High-Level Synthesis and Verification. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/. Accessed 20-08-2023.

[38] Peng Tu and David Padua. 1995. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 47–55. https://doi.org/10.1145/207110.207115

[39] Peng Tu and David Padua. 1995. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Proceedings of the 9th International Conference on Supercomputing* (Barcelona, Spain) *(ICS '95)*. Association for Computing Machinery, New York, NY, USA, 414–423. https://doi.org/10.1145/224538.224648

[40] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative Decoupled Software Pipelining. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 49–59. https://doi.org/10.1109/PACT.2007.4336199

[41] M.E. Wolf and M.S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (1991), 452–471. https://doi.org/10.1109/71.97902

[42] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[43] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. 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* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 269–278. https://doi.org/10.1145/3174243.3174255

Received 2023-11-13; accepted 2023-12-23