Efficient Design Space Exploration for Dynamic & Speculative High-Level Synthesis

Dylan Leothaud, Jean-Michel Gorius, Simon Rokicki, Steven Derrien

Efficient Design Space Exploration for Dynamic & Speculative High-Level Synthesis

Dylan Leothaud*, Jean-Michel Gorius*, Simon Rokicki*, Steven Derrien*
*Univ Rennes, Inria, CNRS, IRISA

***Abstract*—High-Level Synthesis performs well for compute-intensive loops with regular control but struggles to uncover parallelism in kernels with complex control-flow. Novel scheduling techniques based on dynamic scheduling and speculation have been proposed to address this issue. Although they outperform classical static scheduling techniques, they also come at a significant area overhead. Precisely determining where and by how much to apply these techniques remains an open problem, which we address in this work through an efficient exploration algorithm (combining pruning and search heuristics). We show that our approach can explore large solution spaces while producing efficient solutions.**

I. Introduction

High-Level-Synthesis is a well-established design technique for implementing hardware accelerators. It has been shown that HLS can be competitive with manual designs, but it is also acknowledged that existing tools have a lot of shortcomings. One of their weaknesses is their inability to efficiently deal with complex data-dependent control flow or memory accesses. This stems from the fact that HLS tools rely on static scheduling techniques that do not take advantage of runtime information. This limitation hinders their applicability to emerging application domains such as graph analytics and sparse computation, and has therefore motivated a lot of research work. Most of this work aims at extending HLS tools to support dynamic [1]–[7] and speculative [5], [8]–[10] execution mechanisms. Despite these advances, the problem of determining *when* and *how* to take advantage of speculation and dynamism remains somewhat open (see Section V for an in-depth related work analysis). In particular, introducing dynamicity and speculation can lead to a very large design space, with subtle performance/cost trade-offs that are difficult to evaluate at a high level in the design flow.

We propose to address the latter issue through a combination of static analysis and high-level profiling. To do so, we rely on a unified representation that captures dynamic and speculative scheduling opportunities, including memory speculation [8], [10], [11]. More precisely, the contributions of this work are the following.

- A design-space exploration (DSE) framework enabling rapid exploration of the design space, thanks to an efficient pruning algorithm and search heuristics, based on the SpecHLS design flow [10].
- An experimental evaluation of the approach on several challenging examples, along with a quantitative and qualitative discussion of the results in terms of area and speed.

This paper is organized as follows. Section II provides background information on dynamic and speculative scheduling. Section III presents our approach and Section IV addresses its experimental validation. Section V discusses how our approach differs from existing work, and Section VI concludes the paper.

II. Background

This section recalls key concepts related to HLS, focusing on static, dynamic and speculative scheduling techniques. It provides an overview of the SpecHLS flow and its program IR, and discusses the trade-offs involved in such flows.

A. Scheduling in High Level Synthesis

Efficient custom hardware relies on parallelism and customization to deliver performance. *Identifying* and *exploiting* parallelism is challenging from a compiler perspective. Existing HLS tools build on established optimizing compiler techniques to extract parallelism from user code. In particular, *loop pipelining* [12] has proven to be a key transformation for HLS, thanks to its ability to automatically infer deep and wide pipelined datapaths, even in the presence of loop-carried dependencies. Loop pipelining relies on the existence of a non-unit minimum reuse distance along loop-carried dependencies that is determined at compile-time, through static analysis or user directives. Some kernels cannot benefit from loop pipelining, often because their control flow or memory access patterns are unpredictable. This is the case of the example provided in Figure 1, where the iteration execution path is dependant on the outcome of a data-dependant decision. In this situation, the scheduler will consider the worst-case behavior and derive a schedule with an Initiation Interval (II) of $\mathrm{II} = 4$, as depicted in part ⓐ of Figure 1.

Dynamic and speculative scheduling aim at supporting more efficient implementations, where the precise execution of operations is determined at run-time. These techniques take advantage of some form of dynamic or speculative execution as illustrated in parts ⓑ and ⓒ of Figure 1. Several approaches have been proposed to support such mechanisms within a High-Level Synthesis flow. Dynamic dataflow-based techniques [4] follow a pure dataflow approach, where the control-flow of the program is materialized as tokens within the circuit. This approach is very elegant in that it guarantees *correct by construction* circuits, while enabling very aggressive schedules. These dynamic dataflow approaches have been extended to support speculation [9], but with weaker guarantees. These approaches come at a significant area overhead, which can lead

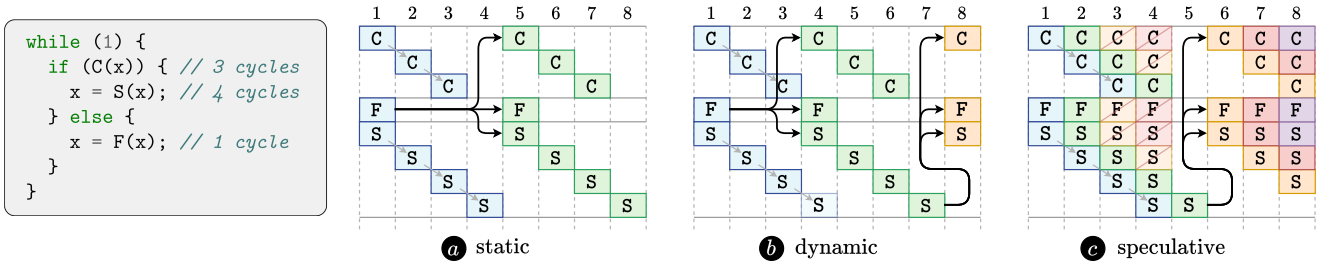

Fig. 1. A summary of High-Level Synthesis scheduling techniques. We assume that `C(x)` evaluates to `true` only for the second iteration. The static schedule (a) is the result of classical HLS scheduling, which employs *speculative code motion* to start the execution of `F` and `S` as early as possible. The dynamic (b) and speculative (c) schedules are the result of state-of-the-art HLS scheduling techniques [4], [8]. Light arrows represent data dependencies between operator stages, while bold arrows represent inter-iteration data dependencies. The latter are omitted from the speculative schedule, except for the mispeculation case (after the second iteration). We represent rolled-back operations using stroke-through operator stages in the speculative schedule.

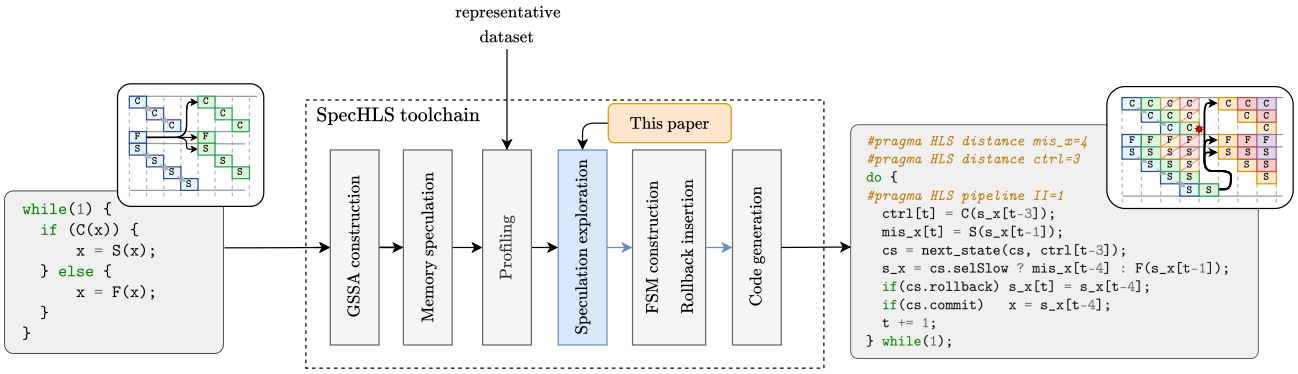

Fig. 2. Overview of the SpecHLS source-to-source compilation flow. The input code is represented on the left, while the code produced by the toolchain is shown on the right. This paper focuses on the design space exploration phase.

to poor area/delay benefits [13]. Several techniques have been proposed to address this issue by resorting to static schedules for groups of operations, where it can be shown that neither dynamicity nor speculation can improve performance [13]–[15].

Operating from dataflow representations requires a complete overhaul of existing HLS toolchains, which are based on static schedules. To address this issue, other approaches propose to extend HLS toolchain capabilities to support some form of dynamicity or speculation, by leveraging existing static loop pipelining capabilities [8], [10]. These approaches incrementally weave dynamicity and speculation in the schedule to improve performance. They have several advantages: they support most dynamic execution patterns, can handle speculation, and take advantage of all existing optimizations in existing HLS tools. The SpecHLS [10] flow follows this strategy: it enables the design of both *dynamic* and *speculative* hardware from C or C++ code. This flow operates as a source-to-source optimizer to be used on top of a commercial HLS tool, as illustrated in Figure 2.

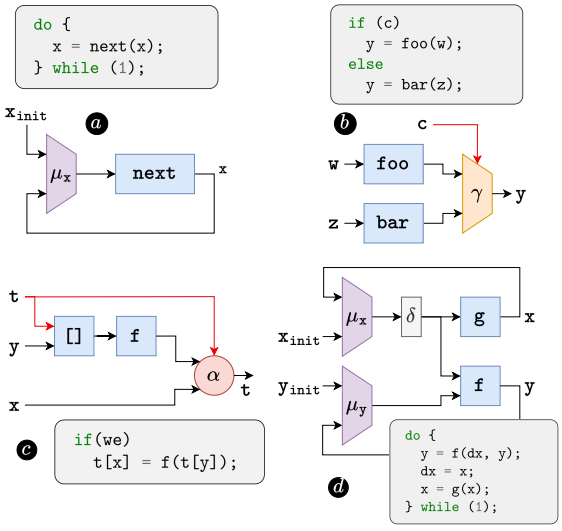

Fig. 3. Extended Gated-SSA operators. μ-nodes (a) represent loop headers, γ-nodes (b) encode control-flow decisions, and α-nodes (c) serve as array updates. δ-nodes (d) act as delays on the datapath, storing a value for one iteration before releasing it.

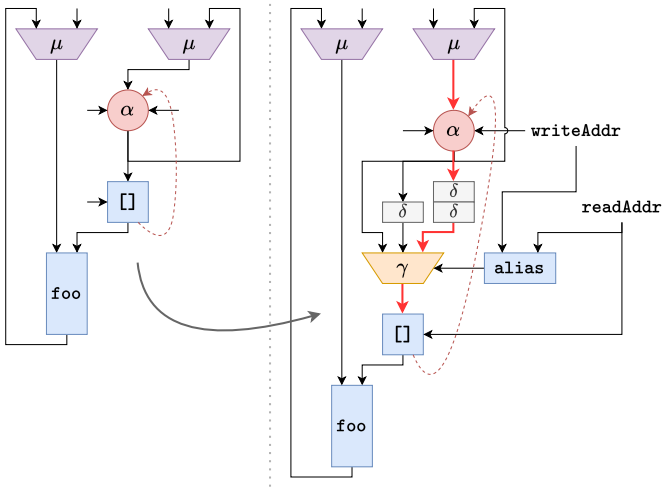

Fig. 4. Recasting memory dependencies as control-flow decisions [11]. The dotted arrow represents an intra-iteration Read-after-Write dependency. The fast path from the start of the iteration to the array read (through two δ-nodes) is indicated by bold red arrows. By exposing new speculation opportunities in our intermediate representation, we also increase the design space size.

B. A Gated-SSA with Speculation Support

The SpecHLS flow uses a variant of the Gated-SSA (GSSA) [16], [17] representation. The latter replaces 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 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, but they also encode the predicate that determines which value is to be selected when execution reaches them. We denote them by $\gamma(c, x_0, x_1, \ldots)$, with c the predicate, and $x_0, x_1, \ldots$ the possible values for variable x.

Until its code generation phase, SpecHLS treats arrays as immutable *values* [10]. We introduce the α operator to model array updates. This operator takes three arguments, $\alpha(t, x, v)$, where t is the array to update, x is the index of the array element to be updated, and v is its new value. This operator produces a new array value, whose element at index x is v, and whose contents are otherwise identical to t. The last operator we introduce is the δ-node. δ-nodes act as delays on the datapath, storing a value for one iteration before releasing it. Figure 3 illustrates these GSSA operators and their corresponding C constructs. The loop exit condition is omitted from μ-nodes to simplify our examples.

Gated-SSA allows us to work with fully-predicated φ-nodes, in the form of μ-and γ-nodes. This predication outlines

```
while (1) {
  if (cond1(y)) {
    next_y = f(y);
  } else {
    next_y = g(y);
  }
  ...
  if (cond2(y)) {
    next_y = h(y);
  }
  y = next_y;
}
```

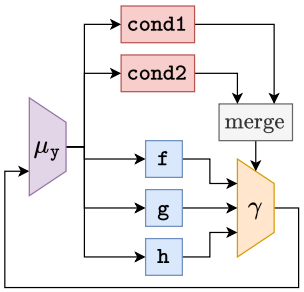

Fig. 5. Leveraging cross-basic block speculation opportunities in SpecHLS. By working directly at the DDG level, SpecHLS can uncover speculation opportunities across multiple basic blocks.

speculation opportunities in the IR. Each γ-node can then be seen as a potential speculation candidate [8]. μ-nodes expose speculation opportunities on the loop exit condition.

C. Memory Speculation

SpecHLS also supports memory speculation by recasting data hazards as a control-flow decision over the existence of a RAW, WAR or WAW memory dependency [11]. This translates into additional γ-nodes in the program IR. The more γ-nodes are present in the IR, the more potential for aggressive speculation and dynamism.

Figure 4 illustrates a transformation that exposes a new γ-node in the GSSA intermediate representation to handle a RAW memory hazard. Since the read and write addresses are not known at compile time, there may be aliasing issues between the α operator on the left-hand side, and the following array read. The key idea behind this transformation is that we can insert a runtime hazard detection mechanism in the form of a γ-node controlled by an `alias` function to check for aliasing between consecutive iterations. The `alias` node keeps a buffer of k past write addresses, and checks the current read address for aliasing, returning the distance of the current iteration to the first aliasing iteration (or $k+1$, if there are no aliases in the last k iterations). The γ-node then selects a *delayed version* of the array to pass on to the read operator. We note that delaying an array through a δ-node is equivalent to ignoring the last update to said array.

Finally, SpecHLS combines this transformation with speculation on the most delayed path, *i.e.*, the one with the largest reuse distance, to efficiently handle data hazards at runtime.

D. Fine-Grained Speculation Discovery

SpecHLS operates on a data-dependency graph (DDG) to transform its input to a speculatively-scheduled output [10]. This approach exposes fine-grained interaction between operators, which existing approaches based on basic block analysis fail to uncover [9]. By not confining themselves to basic block boundaries, SpecHLS transformations can leverage inter-block speculation opportunities, e.g., by merging conditions with similar latencies. Figure 5 illustrates such a situation, where the calls to `cond1` and `cond2` are merged into a single

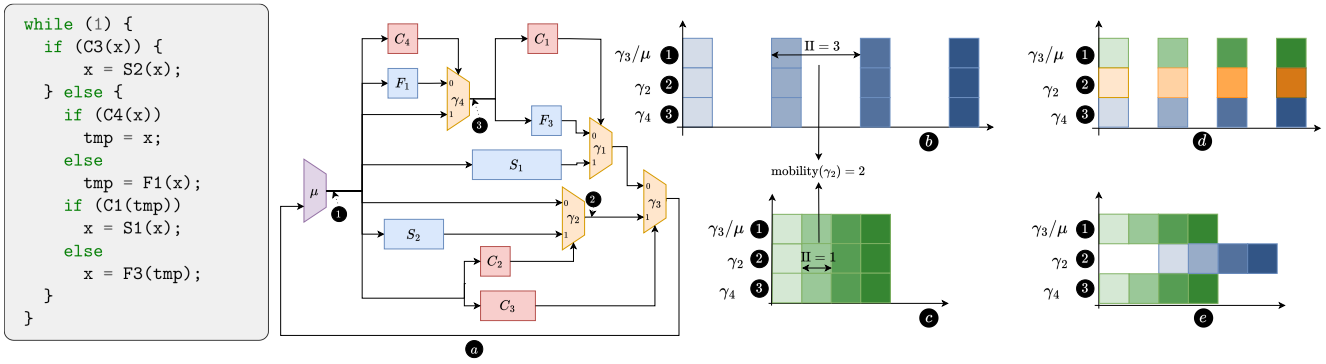

Fig. 6. Illustrative example for our exploration algorithm. Part a illustrates the Gated-SSA representation of the circuit, where we assume $T_{C_{1,2,4}} = T_{F_{1,3}} = 5\,\text{ns}$, $T_{C_3} = T_{S_2} = 15\,\text{ns}$, $T_{S_1} = 25\,\text{ns}$ and $T_\gamma = 1\,\text{ns}$. Part b presents a schedule trace for the circuit, unrolled four times. Part c exhibits a schedule trace for the unrolled circuit, where each γ-nodes is controlled by an oracle that chooses the first available input. Part d displays a schedule trace for the unrolled circuit, where γ_1 and γ_4 are controlled by an oracle and γ_2 always chooses its first input. Part e depicts a schedule trace for the unrolled circuit, where γ_1, γ_3 and γ_4 are controlled by oracles.

control signal across multiple basic blocks, exposing a three-way speculation opportunity on `next_y`.

E. Problem Statement

One of the main strengths of SpecHLS lies in its ability to uncover complex fine-grained speculation opportunities that would be missed by other approaches. This raises the challenge of determining *where* (i.e., on which γ-nodes) and *how* (i.e., for which input) to speculate. An exhaustive search is excluded, since it is common for control-heavy kernels to reach several hundreds of thousands of potential speculation configurations. More information on the design space size can be found in our experimental results (Section IV-B).

III. Proposed Approach

This work proposes a design space exploration algorithm that uses a combination of static analysis and profiling data. Static analysis is used to prune the search space whereas profiling data is used to guide the exploration. Since our approach heavily relies on loop pipelining techniques, we first recall some principles of pipelined scheduling.

A. Determining the RecMII in a Pipelined Schedule

To understand our approach, it is first necessary to introduce the notion of recurrence-constrained minimum initiation interval (RecMII). The RecMII corresponds to the best achievable II, without any resource constraint considerations. This metric captures the maximum amount of available pipeline parallelism within a loop. It uses both operator delays and dependence distances to obtain the best II along all elementary cycles in the graph, as illustrated in Figure 6. The II is determined by the following formula, where $T(\text{cycle}_i)$ corresponds to the cumulative delay along elementary cycle i, and $D(\text{cycle}_i)$ to the cumulative dependence distance along the same elementary cycle.

$$\text{RecMII} = \max_{i \in \text{cycles}} \frac{T(\text{cycle}_i)}{D(\text{cycle}_i)}$$

TABLE I
Profiling data and γ-configurations for Figure 6.

Scenario	γ_1	γ_2	γ_3	γ_4	CP (ns)	Prob.
configuration 1	1	0	1	0	2 ns	0.34
configuration 2	0	0	1	1	2 ns	0.21
configuration 3	0	0	0	1	8 ns	0.17
configuration 4	1	1	0	0	27 ns	0.10
configuration 5	0	1	0	0	13 ns	0.08
configuration 6	1	0	0	1	27 ns	0.04
static	-	-	-	-	27 ns	1
speculative 1	0	-	0	1	8 ns	0.17
speculative 2	-	0	1	-	2 ns	0.58

Since the number of elementary cycles can be exponential in the number of operations in the graph, directly computing the RecMII using the former approach is impractical. However, it has been shown by Leiseron and Saxe [18] that the RecMII of a given loop can be computed in polynomial time, since it amounts to a flow problem.

We illustrate the II computation on the example shown in Figure 6, in which multiple γ-nodes interact with one another. In this example, assuming a target clock period of $T_{\text{clk}} = 10\text{ns}$, and delays provided by the caption, we would obtain a static schedule with a value of $\text{II} = 3$ (due to the slow path through S_1). This, however, is a pessimistic II. Profiling data may actually show that the path through S_1 only seldom contributes to the actual result.

Table I captures the probability of observing γ-nodes selecting one of their two inputs. Since the probabilities associated to distinct γ-nodes are likely to be correlated, we cannot reason on each γ-node individually, and therefore introduce the notion of γ-*configuration*, which defines the joint outcome of all the γ-nodes in the kernel. Each row in Table I corresponds to such a configuration, along with its probability of occurrence during program execution.

Inspecting the GSSA graph in Figure 6 will reveal that the best achievable II can be 1, assuming that γ-nodes γ_1, γ_4 and

γ_3 select their first input, but also when γ_2 and γ_3 selects their first and second input, respectively. It is important to note that, due to the slow delay along the C_3 path, the latter scenario requires speculative execution (with potential mispeculation penalties) whereas the former can achieve $\mathrm{II} = 1$ with dynamic scheduling only. Given the profiling data in Table I, we can also conclude that configuration *speculative 2* is the most likely among the two. It can therefore be considered the best strategy.

B. Exploring the Solution Space

Although the previous example remains simple, a large number of γ-nodes quickly makes the problem of finding the best configuration intractable. If we denote the set of γ-nodes in our representation by Γ, and the number of inputs for node n by $\mathrm{inputs}(n)$, then our solution space contains up to

$$\prod_{n \in \Gamma} \mathrm{inputs}(n)$$

distinct γ-node combinations to consider. For each γ-node, we need to determine whether the γ-node is selected for dynamic or speculative execution and, if so, which one of the two paths is considered as the *taken* path. However, we are only interested in the subset of configurations that have the following properties:

- The configuration achieves the target II (usually $\mathrm{II} = 1$) by exposing a short path in the GSSA graph.
- The configuration has an occurence probability higher than a reference threshold θ. The latter is set by the user as a parameter to guide the design space exploration.
- The configuration is minimal: removing any speculated γ-node from it increases the achievable II.

A solution meeting all these requirements is called a *valid configuration*. In the context of a full DSE, the goal is to obtain all such configurations. However, because the search space is intractable, we may instead look for the first n solutions. In all cases, we need to be able to (i) prune the search space as early as possible and (ii) explore the space by favoring solutions that are likely to be more profitable, through search heuristics.

C. Static Design Space Pruning

Our pruning strategy consists of determining the set of γ-nodes that *need not* be considered for speculation. We also determine, given a set of speculation candidates, *whether we can reach* the target II. To do so, we rely on a form of abstract schedule, in which we consider each γ-node to operate in one of the following modes:

- In the *static* mode, the node output is considered ready as soon as all of its inputs are ready.
- In the *speculative* mode, the γ-node assumes that one specific data input is always selected: the γ-node output is ready as soon as the speculated input is ready.
- In the *oracle* mode, the γ-node always picks the earliest of its data inputs.

When all γ-nodes operate in *static* mode, the schedule amounts to a fully static schedule (schedule ⓑ in Figure 6). When all γ-nodes operate in *oracle* mode, the resulting schedule assumes the best possible execution outcome for all nodes (schedule ⓒ in Figure 6) and is therefore a lower bound for II. We apply this scheduling step repeatedly until it reaches a fixpoint[1] where the schedule starts to expose a periodic behavior. This period is a lower bound on the achievable II for that configuration.

We illustrate the operation of our exploration algorithm through examples in part ⓓ and ⓔ of Figure 6, where we consider two different γ-node sub-sets: (i) γ_1 and γ_4 as oracles, γ_2 as speculated on its first input and γ_3 as static, and (ii) all γ-nodes as oracles but γ_2 as static.

We use this abstract schedule to identify nodes that need not be speculated (thanks to a mobility metric, detailed in the following), and prune the search space when the γ-configuration is guaranteed not to lead to a viable solution.

We define the schedule difference (for a given γ-node) between the worst case and the best case schedule as the γ-node's *mobility*. Figure 6 shows how we obtain a mobility score of 2 for γ_2. A γ-node with a mobility of zero will not benefit from speculation, and can safely be discarded from the search space. Similarly, γ-nodes with a large mobility are promising candidates for speculation, and should be considered first during the search phase.

We can further exploit these abstract schedules to prune the search space. Consider, for example, a situation where we have determined the speculation decision for a subset of m among n γ-nodes (speculated on first input, second input, or not speculated at all). We now need to determine whether this subset may ultimately lead to a solution enforcing the target II, possibly by speculation on additional nodes. We can use our abstract scheduler to answer this question by simply considering the remaining $m - n$ γ-nodes as operating in *oracle* mode. If the resulting schedule period is higher than the target II, it is safe to conclude that there exists no refinement of the configuration at hand that can lead to the target II.

Our pruning technique is illustrated in parts ⓓ and ⓔ of Figure 6. Part ⓓ represents the abstract schedule obtained when γ_2 speculate on input 0 and all γ-nodes but γ_3 operate in oracle mode. In this case, the schedule's steady regime shows that it is impossible to achieve $\mathrm{II} = 1$: we are constrained at $\mathrm{II} = 2$ due to a long path through μ_y. We can use this result to prune a subset of the design space since we now know that it is impossible to reach $\mathrm{II} = 1$ by speculating on the first input of γ_2 but without speculating on γ_3. On the contrary, part ⓔ of Figure 6 shows the abstract schedule obtained when all γ-node but γ_2 operate in oracle mode. In that case, the steady regime achieves $\mathrm{II} = 1$, and we need now we may need to further explore the sub-space (by speculating on additional γ-nodes) to possibly reach $\mathrm{II} = 1$.

D. Exploration Heuristics

Pruning alone is not sufficient to enable efficient design exploration. The way we explore the design space is also very

[1] It can be shown that the fixpoint can be reached after at most $2 \times \mathrm{card}(\mathcal{M})$ iterations, where $\mathcal{M}$ is the set of all μ-nodes in the IR for a given circuit.

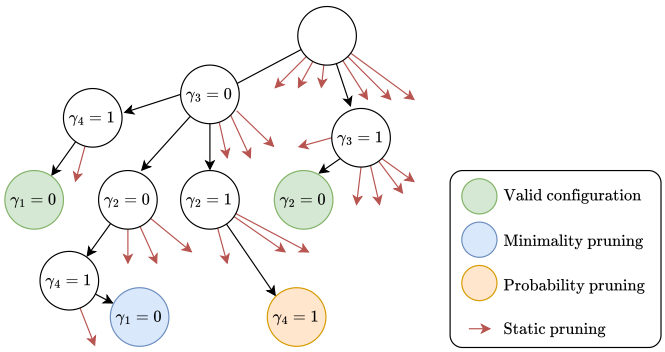

Fig. 7. Exploration tree for our running example, using the total order based on mobility, e.g. $\gamma_3 \prec \gamma_2 \prec \gamma_4 \prec \gamma_1$.

important. Our approach uses a breadth-first search among the different γ-configurations. This corresponds to an exploration tree, as illustrated in Figure 7. Additionally, we use a total order $\prec$ over γ-nodes to prevent exploring the same configuration twice. The total order we use assigns higher rankings to promising γ-nodes. In our current implementation, γ-nodes are sorted based on the mobility value defined in III-C.

In the exploration tree, a child c of a node f represents a γ-configuration $\text{conf}(c)$ where (i) we speculate over the same γ-nodes as the configuration in f, and (ii) we add one speculation over a specific input i of a node γ_n which is smaller than every speculated γ-node in f. More precisely, for each node n in the exploration tree,

$$\text{conf}(n) = \text{conf}(\text{father}(n)) \cup \{\gamma_m \to i\}$$

where

$$\gamma_m \prec \min_{\prec} \{\gamma \mid \exists i \in \mathbb{N},\ \{\gamma \to i\} \in \text{conf}(\text{father}(n))\}\,.$$

The root of the exploration tree corresponds to the fully static schedule. Figure 7 represents an exploration tree for the example in Figure 6, based on a possible ordering of γ-nodes.

For each node in the tree, we apply the following pruning techniques:

1) **Minimality:** We verify whether there already exists a valid configuration, which is a strict subset of the current configuration. In such a case, we can prune the exploration, since we know that the current configuration (and all of its children) will not be minimal.
2) **Probability:** We compute the probability of the current configuration based on profiling information. If it is below the validity threshold, all sub-configurations are also below the threshold, and the branch can be pruned.
3) **Static pruning:** We use the technique described in Section III-C to determine whether it is possible to reach $\text{II} = 1$ using current speculation choices, with all smaller γ-nodes configured as oracles. We prune the branch if it is impossible.

Figure 7 illustrates the pruning steps undertaken during exploration. Configuration (γ_3:0, γ_2:0, γ_4:1, γ_1:0) is pruned, since the valid configuration (γ_3:0, γ_4:1, γ_1:0) is one of its valid strict-subsets. Configuration (γ_3:0, γ_2:1, γ_4:1) is impossible, based on profiling information. Most of the other configurations are pruned by static pruning.

The total order over γ-nodes used for the exploration is critical for efficiently pruning the tree. If speculating only on the smallest γ-node is sufficient to obtain $\text{II} = 1$, then the pruning technique will always return that it is possible to obtain $\text{II} = 1$ by adding speculation over smaller γ-nodes, and will never prune.

Algorithm 1 gives a pseudo-code implementation of our proposed approach.

Algorithm 1: Exploration of speculation opportunities.

```
sortedGammas ← sortByMobility(Γ)
configsToExplore ← {γ → i | ∀γ ∈ sortedGammas, ∀i ∈ inputs(γ)}
validConfigs ← ∅
while configsToExplore ≠ ∅ do
  config ← pop(configsToExplore)
  for every γ ∈ sortedGamma such that
  ∀γ' → i ∈ config, γ ≺ γ' do
    for every input i of γ do
      newConfig ← config ∪ {γ → i}
      if probability(newConfig) ≥ threshold and
      recmiiWithOracle(newConfig) == 1 then
        if recmii(newConfig) == 1 then
          validConfigs ←
          validConfigs ∪ {newConfig}
        else
          configsToExplore ←
          configsToExplore ∪ {newConfig}
        end if
      end if
    end for
  end for
end while
```

E. Choosing the Final Configuration

Once the exploration is completed, we obtain a set of valid configurations. Since we generate a speculative schedule entirely at compile-time, we are capable of statically computing the iteration latency for each configuration of γ-nodes. Consequently, we can use the profiling information depicted in Table I to derive a precise estimate of the whole loop latency without performing RTL simulation. Additionally, we can build an area score based on the size of the control logic inserted to handle the speculations. Picking the right configuration is then a matter of deciding which trade-offs to make between performance and area.

IV. Experimental Validation

This Section evaluates the relevance of our approach from a qualitative and quantitative perspective. Our aim is to demonstrate that our flow finds good solutions even in large design spaces, and that the latter improve performance compared to

TABLE II
AREA RESULTS AND DESIGN SPACE METRICS FOR LARGE BENCHMARKS.

Benchmark	Type	Area					Design space size				Runtime
		II	T_{clk} (ns)	LUT	FF	DSP	γ-nodes	Baseline	Heuristics	SOTA [7]	
FPU	Baseline	3	7.90	353	283	2	21	30.9B	116k	10.9k	1055s
	Speculative	1	17.6	2 885	1 105	2					
SpMM	Baseline	6	6.27	256	442	4	12	708k	60	1.35k	6s
	Speculative	2	6.71	1 097	1 377	4					
RISC-V CPU	Baseline	5	8.891	1 499	361	0	16	752M	1.46k	11.3M	263s
	Speculative	1	14.35	1630	953	0					
Superscalar	Baseline	4	7.16	3502	2688	0	16	178M	1.19k	5.74M	177s
	Speculative	2	10.451	6865	5682	0					

static scheduling. In the following, we describe our experimental setup and then discuss the results that we obtain.

A. Experimental Setup

The SpecHLS flow is implemented within the GeCoS source-to-source compiler infrastructure [19], and relies on the CIRCT [20] project to provide a fast scheduling infrastructure for our pruning strategy. We use Vitis HLS 2021.2 as a back-end to perform High-Level Synthesis, with an XC7A200 as the target FPGA.

Existing HLS benchmarks [21], [22] focus on kernels with regular control-flow and memory access patterns: they favor applications that work well with state-of-the-art HLS tools. This makes them poor candidates for evaluating our approach. Instead, we choose a selection of benchmarks that expose complex and data-dependant control flow. These benchmarks were obtained from previously published work [1], [10], [15], [23], [24] and are challenging to pipeline, since reaching $\text{II} = 1$ requires a non-trivial combination of speculations that remains out of reach of SOTA dynamic scheduling techniques [4], [7], [15]. We briefly describe them below:

- **RISC-V**: a processor implementing the rvi32 ISA, in the form of an untimed abstract datapath, exposing speculation opportunities on PC, registers, etc.
- **Superscalar**: a simplified $2\times$ unrolled version of the above, exposing the opportunity to issue two instructions per iteration.
- **FPU**: an FPU supporting addition and multiplication, with a fast path in case operands have the same exponent.
- **SpMM**: structured sparse matrix product, with speculation opportunities over the matrix weight structure to enable parallel accumulation.

B. Experimental results

Table II shows the area results and design space metrics for our benchmark set. The baseline (with no speculation applied) is shown first, followed by its speculative counterpart.

1) Scalability evaluation: The rightmost part of Table II provides metrics regarding the size of the design space. The first two columns provide the number of γ-nodes and the size of the full solution space that can be considered. The column labeled *Heuristics* demonstrates the effectiveness of the proposed pruning techniques. These results are obtained with a probability threshold of $\theta = 10\%$. They show that our approach is able to drastically reduce the search space by 5 to 6 orders of magnitude. As a comparison, we show the size of the explored space when following the approach of Szafarczyk et al. [7] in the last column of Table II. Our results show that our approach is much more scalable, while natively supporting both control-flow and memory speculation.

The last column of Table II shows the runtime for performing the DSE alone. Results show that even for complex benchmarks, our approach is able to carry out the whole analysis in less than twenty minutes.

2) Performance evaluation: The left-most part of Table II provides area and performance metrics, comparing the baseline implementation with the one found by our exploration process. As for previous work on speculative loop pipelining, results demonstrate that the II is reduced, at the cost of additional hardware and lower clock frequency. Part of the area overhead is due to the additional logic used to handle mispeculations, the rest is a direct consequence of reaching a lower II, leading to fewer resource-sharing opportunities. Two of our benchmarks reach $\text{II} = 2$, while our exploration found that it should achieve $\text{II} = 1$. This is due to a mismatch between our delay model and that of Vitis HLS, which we hope to fix for the final version of the paper.

V. DISCUSSION

Many research works have addressed the problem of pipelining complex loops, either through advanced static schedules [23], [25], runtime schedules [3], [4], or both [1], [5], [8]. In this section we focus the discussion on research work that shares the most resemblance to ours.

Szafarczyk et *al.* [7] automatically expose dynamism in HLS code through a compiler analysis that operates at the basic block level. Their approach uses a greedy algorithm that iterates over all basic blocks in every elementary circuit of the control-flow graph to identify dynamic scheduling opportunities. Contrary to our approach, they neither support speculation nor handle data-dependent memory accesses. Our approach supports both, thanks to the use of a unifying model for control flow and memory speculation [11]. Additionally,

the elementary cycle enumeration in the CFG may not scale on large benchmarks, as shown by the results in Section IV-B.

In their Dynamatic toolchain, Josipović et *al.* [4], [9] operate at the basic block level to coordinate execution between dynamically scheduled and speculative regions in the generated hardware. Our finer-grain representation exposes more speculation opportunities but dramatically increases the search space, as demonstrated in Section IV-B. This large design space size motivates the need for a fast and scalable DSE algorithm. Dynamatic also delegates memory speculation and dependency management to an external load-store queue [26]. Our approach treats memory speculation as a particular case of control-flow speculation and can, therefore, integrate it into our DSE process. Speculative execution support in Dynamatic [9] seems to have been put on hold, and the toolchain focuses on dynamic scheduling. Our approach unifies dynamic and speculative scheduling in a single framework.

Recent work has focused on reducing the area overhead of dynamically-scheduled hardware by introducing *static islands* in elastic circuits [13]–[15]. These works tackle a somewhat similar problem, but follow the opposite approach to ours: rather than weaving dynamicity in a static-schedule, they instead try to eliminate dynamicity when it can be shown that it cannot improve performance. The main limitation of these approaches is their inability to take advantage of speculation, and to provide early performance estimates.

Cheng et *al.* [6] propose a probabilistic scheduling model for HLS. Their approach models schedules as stochastic Petri Nets, and relies on control-flow profiling data obtained through Dynamatic [4], coupled to their own memory access profiler. The authors apply probabilistic scheduling to the static islands problem discussed previously. While their approach uses probabilities to drive their scheduling exploration, the model proposed in their paper does not capture the potential correlations between control-flow decisions in the input program. Our approach maps profiling data to execution paths instead of individual control-flow decisions, capturing any potential correlation in the process.

VI. Conclusion

Speculative High-Level Synthesis opens interesting new design perspectives, as it allows the acceleration of complex, control-dominated kernels. Determining when and where to apply speculation raises new issues in term of design space exploration. In this paper, we propose a design space exploration flow that takes advantage of static analysis and profiling information to efficiently explore large design spaces with hundreds of thousands of solutions in a matter of minutes without the need for costly RTL-level simulations. Our approach is scalable enough to automatically infer the set of speculation decisions that enable the automatic generation of an in-order superscalar RISC-V core from an algorithmic description.

Acknowledgements

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

References

[1] M. Alle, A. Morvan, and S. Derrien, "Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2463209.2488796

[2] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS Today: Successes, Challenges, and Opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 51:1–51:42, Aug. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3530775

[3] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 189–194. [Online]. Available: https://doi.org/10.1145/3020078.3021754

[4] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-level Synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, Feb. 2018, pp. 127–136. [Online]. Available: https://dl.acm.org/doi/10.1145/3174243.3174264

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

[6] J. Cheng, J. Wickerson, and G. A. Constantinides, "Probabilistic Scheduling in High-Level Synthesis," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 195–203.

[7] R. Szafarczyk, S. W. Nabi, and W. Vanderbauwhede, "Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 1–9. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/FPL60245.2023.00009

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

[9] L. Josipović, A. Guerrieri, and P. Ienne, "Speculative Dataflow Circuits," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 162–171.

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

[11] ——, "A Unified Memory Dependency Framework for Speculative High-Level Synthesis," in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 13–25. [Online]. Available: https://doi.org/10.1145/3640537.3641581

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

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

[2]https://lotr.gitlabpages.inria.fr/website/

[14] J. Cheng, L. Josipovic, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining Dynamic & Static Scheduling in High-level Synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 288–298. [Online]. Available: https://doi.org/10.1145/3373087.3375297

[15] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "DASS: Combining Dynamic & Static Scheduling in High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 628–641, 2022.

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

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

[18] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[19] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys, "GeCoS: A framework for prototyping custom hardware design flows," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013, pp. 100–105.

[20] "CIRCT: Circuit IR Compilers and Tools," https://github.com/llvm/circt.

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

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

[23] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, Sep. 2018.

[24] J.-M. Gorius, S. Rokicki, and S. Derrien, "Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–6.

[25] A. Morvan, S. Derrien, and P. Quinton, "Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 3, pp. 339–352, Mar. 2013.

[26] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, sep 2017. [Online]. Available: https://doi.org/10.1145/3126525