Master research Internship

Bibliographic report

Speculative High-Level Synthesis of Instruction Set Processors

Hardware Architecture

Author:
Jean-Michel Gorius

Supervisors:
Steven Derrien
Simon Rokicki
CAIRN

Abstract: In the embedded and IoT spaces, typical computing scenarios traditionally involve processing large quantities of data in regular patterns. However, the trend in specialized computational domains is slowly shifting. New applications such as data mining, graph analytics, and machine learning introduce a new computational framework that requires special-purpose hardware to provide a programmable interface and to operate on control-flow dominated workloads. These workloads are well-suited for Instruction Set Processors, but the design of these complex hardware pieces is costly and very error-prone. This internship aims to study how to take advantage of state-of-the-art work on speculative hardware synthesis to make Instruction Set Processor design amenable to High-Level Synthesis flows. We will study the challenges induced by interwinding multiple speculations and explore fine-grain misspeculation recovery schemes, directing our efforts towards small embedded processors based on in-order pipelined architectures. This bibliographic report introduces the internship's context, alongside issues and challenges related to speculative hardware synthesis.

Contents

1 Introduction

Spearheaded by the personal computer's advent and the rapid development of microprocessor technology, the computing revolution has profoundly transformed our society and the way we interact with the world. Nowadays, computing devices have become omnipresent: from high-performance multi-core machines in high-end servers to small low-power embedded devices in our phones, computers are everywhere. The fast pace at which computers evolved during the past few decades can be explained by a few key innovations in processors' design and architecture, as observed by [Patterson and Hennessy, 2017]. *Pipelining* and *speculation* are two of those innovations that have shaped modern processing cores' performance. Pipelining refers to a technique used by hardware to enable faster program execution by overlapping the execution of multiple instructions. On the other hand, speculation is a method used to uncover parallelism in programs by letting the processor "guess" the outcome of the execution of a given instruction before it finishes its execution.

When combined, pipelining and speculation can drastically increase the execution speed of a processor. However, such advanced optimizations come with a significant design challenge. Hardware Description Languages (HDL) are programming languages that allow their users to write a specification of a piece of hardware at the Register Transfer Level (RTL). RTL provides an abstraction over physical components that models a digital circuit in terms of the flow of digital signals between physical registers and the logical operations and combinations applied to those signals. Using HDLs to design pipelined Instruction Set Processors (ISP) is challenging and involves a significant number of refinements before getting to an accurate hardware description. Starting from a specification of the underlying Instruction Set Architecture (ISA), hardware designers need to make choices regarding, *e.g.*, the pipeline structure, the number of pipeline stages, or the placement of individual registers. Once the hardware description is written, there is no room left for design space exploration. This design process is tedious and error-prone, making ISA and ISP elaboration extremely difficult.

The increasing demand for custom instructions and architectural features for processors in embedded and IoT applications cannot be addressed with currently available tools. Additionally, most existing processors in those spaces rely on proprietary ISAs, preventing third parties from freely customizing the micro-architecture or instruction set. Those restrictions severely hinder innovation. The RISC-V[1] initiative aims at addressing this issue by developing and promoting an open instruction set architecture. The RISC-V ecosystem is quickly growing and has gained much traction for IoT platform designers, as it permits free customization of both the ISA and the micro-architecture. However, going from a simulation model to an actual hardware description is still a daunting task requiring complex verification and debugging steps.

This internship aims at making instruction set processor design amenable to the embedded space by leveraging High-Level Synthesis (HLS) tools. We will examine new ways to take advantage of state-of-the-art work on speculative hardware synthesis to make ISP design amenable to HLS flows. More specifically, we will explore source-level transformations that would allow *in-order pipelined* ISPs to be synthesized from a high-level behavioral description in the form of an Instruction Set Simulator (ISS). Such a simulator often implements a high-level state machine that decodes instructions and executes them sequentially. The intricate control flow and data dependencies between successive instructions make such designs particularly challenging for HLS toolchains. We will focus on improving state-of-the-art program transformation techniques that enable speculative hardware

[1] `https://riscv.org/`

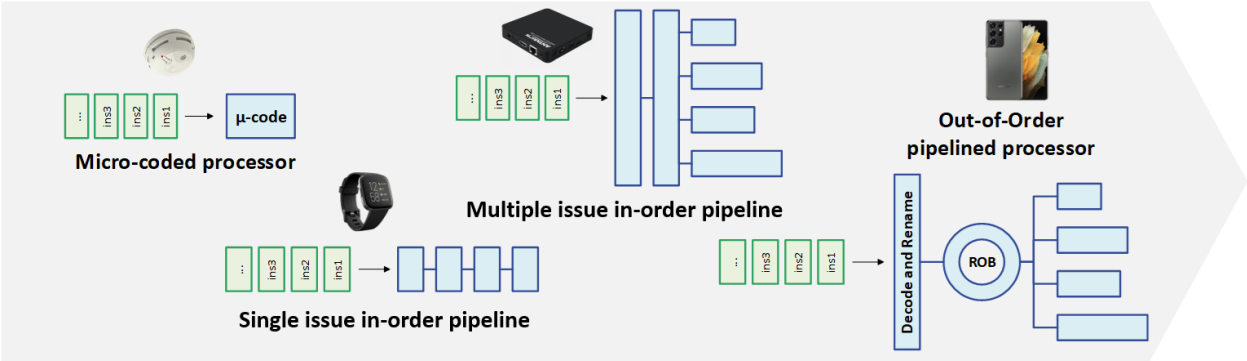


Figure 1: Micro-architectural design space.

synthesis and apply those techniques to instruction set simulators to generate the corresponding processing cores during this internship. We will study the challenges induced by interwinding multiple speculations and explore fine-grain misspeculation recovery schemes, focusing our efforts towards small embedded processors based on in-order pipelined architectures.

The remainder of this bibliographic report is structured as follows. Section 2 exposes the motivation and background for our work. Section 2.1 exposes previous work on instruction set processor synthesis while Section 2.2 gives an overview of High-Level Synthesis and motivates its use for complex hardware design synthesis and design space exploration. Section 2.3 discusses loop pipelining and static scheduling, two fundamental aspects of modern HLS workflows, before emphasizing the limitations of those approaches. A complementary approach to static scheduling, dynamic scheduling, is exposed in Section 3.1. Modern general-purpose instruction set processors rely on dynamic scheduling coupled with speculative execution to achieve high performance on heterogeneous workloads. Section 3.2 explores state-of-the-art techniques for the synthesis of speculative hardware. Finally, Section 4 concludes this report by discussing early reflections on the upcoming work.

2 Motivation and Background

In this section, we start by presenting early work on customizing instruction set processors in the context of Application-Specific Instruction Set Processors in Section 2.1 before giving an overview of the fundamental principles of High-Level Synthesis in Section 2.2. Section 2.3 then focuses on two key aspects of modern HLS flows, namely *loop pipelining* and *static scheduling*.

2.1 Customizing Instruction Set Processors

Instruction set processors are intricate pieces of hardware designed to execute a stream of instructions stored in external memory. These processors offer a highly flexible programming interface, which makes them well suited for highly irregular and heterogeneous workloads. Irregular workloads are inherent to desktop, mobile, and high-performance computing, where most applications are control-dominated. On the other hand, in the embedded and IoT spaces, typical computing scenarios traditionally involve little variability and control but focus on processing large quantities of data. Instead of targeting programmability and flexibility, special-purpose hardware is designed to reduce power consumption and increase performance on a single well-defined set of applications

(e.g., signal processing, video encoding and decoding). However, the trend in specialized computational domains is slowly shifting. New applications such as data mining, graph analytics, and machine learning introduce new computational needs that require special-purpose hardware to provide a programmable interface and to operate on control-flow dominated workloads. Addressing these new requirements challenges hardware manufacturers to design programmable hardware with many custom features while still providing fast processing times and reduced energy consumption. The design of ISPs is a tedious and error-prone process that needs to be conducted carefully to prevent the hardware from misbehaving once it is produced.

A single instruction set specification can be used to design a wide variety of ISPs. Figure 1 illustrates some of the different design choices that can be made for the same ISA. This design landscape spans from very low-power devices based on low-energy micro-coded micro-architectures to high-performance Out-of-Order (OoO) processors. Pipelined designs based on single-issue or multiple-issue in-order pipelines are also widespread in connected devices. This implementation diversity leads to a large design space encompassing tradeoffs between die area, power consumption, and performance. The inherent complexity of design space exploration makes it a good target for design automation.

The first approaches aimed at automating the design of ISPs were proposed in the context of Application-Specific Instruction Set Processor (ASIP) design flows. ASIPs are programmable processing cores targeting a particular application domain. As a result of their specialized nature, the ISA of ASIPs is tailored to the application, exposing specially-crafted instructions and focusing on a small set of tasks. Proposed approaches for automated ASIP design often rely on Domain-Specific Languages (DSL) to model the processor hardware structure along with its ISA [Cloutier and Thomas, 1993, Huang and Despain, 1993, Klemm et al., 2007]. The abstraction level provided by ASIP synthesis tools is often very close to that of hardware description languages and asks for explicit management of architectural choices such as pipeline depth and organization, available data forwarding points, and hazard detection logic.

2.2 High-Level Synthesis

High-Level Synthesis (HLS) was first proposed to overcome the many limitations of HDL-based design flows. In an HDL-based design flow, the hardware generated by the HDL synthesis tool may not always meet the designer's constraints. In such a case, large parts of the specification need to be rewritten, which is at least impractical. Contrary to HDL-based hardware synthesis, HLS allows its user to specify the behavior of a given piece of hardware in a high-level language—often C or C++—and to focus on the algorithmic specification of the hardware's operation. An HLS toolchain infers the hardware's structure from this high-level description and the required resources and clock frequency constraints. This approach abstracts the user away from most implementation details and makes changes to the hardware easier to apply, therefore significantly improving the designer's capabilities to conduct design space exploration.

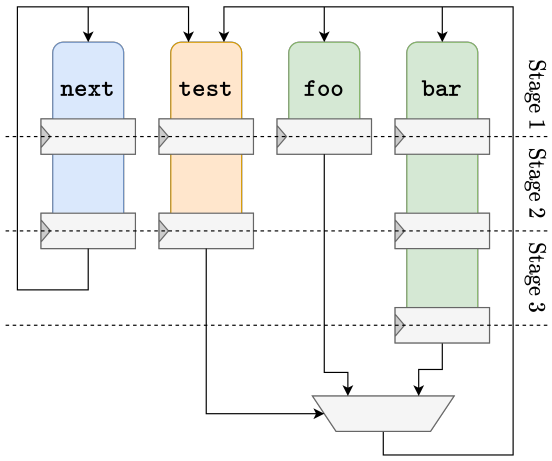
Nowadays, High-Level Synthesis tools are developed both in academia and by major Electronic Design Automation (EDA) vendors. Commercial tools include Xilinx' Vivado HLS[2] and Mentor Graphics' Catapult HLS[3], both based on C/C++. These tools extend the language with implicit semantics and restrict the set of language features that can be used to describe hardware behavior.

[2] `https://www.xilinx.com/products/design-tools/vivado.html`

[3] `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/`

```
1 do {
2   y = next(y);    // 2 cycles
3   if(test(x, y)) // 2 cycles
4     x = foo(x);  // 1 cycle
5   else
6     x = bar(x);  // 3 cycles
7 } while(!x);
```

(a) Sample data-processing code. The number of cycles required for each operation for the target execution frequency is indicated as comments.



(b) Corresponding pipelined datapath.

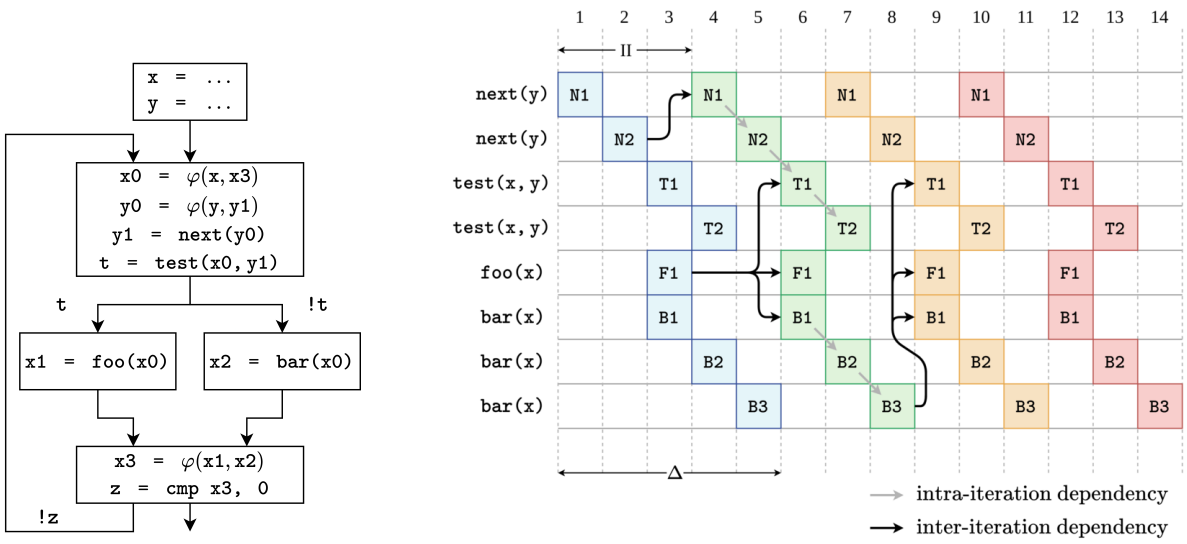
Figure 2: High-Level Synthesis of a data-processing accelerator.

Most notably, HLS introduces strong memory management restrictions by prohibiting dynamic memory management and global variables and providing only limited support for pointer arithmetic.

To get a better feeling for the actual work carried out by an HLS toolchain, let us consider an example. Figure 2a shows sample code that we could write in a typical C-based HLS tool to describe a data-processing accelerator. We will use this example as an illustration throughout this report. Given such an algorithmic specification, an HLS tool will produce hardware in the form of a finite state machine controlling a datapath. Our example code can be processed to produce various hardware layouts depending on the designer's constraints, such as the type of components to use or the target clock frequency. By default, High-Level Synthesis generates a circuit executing one iteration of a loop each cycle. The design's execution frequency depends on each base operator's execution time used in the datapath, determining the length of a clock cycle. The user can also choose to set the desired clock frequency, thereby constraining the generated circuit layout and potentially increasing the generated hardware area. This area/frequency tradeoff is typical in embedded hardware design. Figure 2b shows an example pipelined datapath generated from the code in figure 2a.

2.3 Loop Pipelining and Static Scheduling

When given the code in figure 2a, an HLS toolchain statically schedules the execution of operations on the available resources from the corresponding Static Single Assignment (SSA) [Cytron et al., 1991] representation of the program (figure 3a). It produces a schedule similar to what is given in figure 3b. This structure is akin to the *pipeline* of an instruction set processors capable of overlapping the execution of successive loop iterations and is produced by a standard HLS optimization known as *loop pipelining*. Loop pipelining transforms a sequential loop iteration into a set of independent operations that the hardware can concurrently execute to speed up execution. In figure 3b, cycles are represented on the horizontal axis while the different stages of the synthesized pipeline are represented on the vertical axis. We note that some operations have been divided into multiple stages by the HLS toolchain to accommodate the target execution frequency, *e.g.* `next(y)`



(a) SSA representation of the code in figure 2a. The HLS toolchain manipulates this internal representation to infer the structure of the hardware.

(b) Statically scheduled pipeline. Execution cycles are represented on the horizontal axis while the vertical axis represents the pipeline stages for each operation. The *latency* of the pipeline is given by $\Delta = 5$ cycles while its *initiation interval* is $\text{II} = 3$ cycles.

Figure 3: Internal SSA representation alongside the statically inferred iteration schedule.

executes in two cycles and is divided into two corresponding pipeline stages. Each coloured group on this figure denotes the schedule for an iteration of the loop. The latter's shape is constrained by intra-iteration dependencies, while inter-iteration dependencies constrain the placement of successive iterations in the schedule. One of the significant benefits of loop pipelining is its ability to expose *instruction-level parallelism* (ILP). By dividing individual operations into multiple execution stages, the HLS tool can infer a schedule where multiple instructions can be executed in parallel.

Two metrics characterize a pipelined loop schedule: its *initiation interval* II, which designates the delay between two successive iterations of the same loop, and its *latency* Δ, which corresponds to the time it takes for one iteration of the loop to complete its execution. When synthesizing hardware using HLS tools, designers generally aim at producing pipelined circuits with the smallest initiation interval—often aiming for $\text{II} = 1$. A small value of II allows a new iteration of the loop to start as soon as possible, maximizing throughput and resource usage in the hardware. However, the initiation interval and latency of a given loop is constrained by several factors, including resource availability, target clock frequency, and data dependencies. HLS tools rely on sophisticated compile-time analyses such as *modulo scheduling* [Rau, 1994, Lam, 1988] to compute the best value of II and employ techniques similar to the *software pipelining* to map program instructions to available computational resources.

While the schedule in figure 3b reduces the time needed to complete the execution of the loop compared to entirely sequential execution, it is far from optimal for many cases. For example, when most of the executions of `test(x,y)` yield `true`. Computing `bar(x)` at each iteration wastes both time and resources. Additionally, skipping the latter computation would allow more instruc-

tions or iterations to be overlapped and produce a tighter schedule. However, implementing such a schedule would require an oracle. As a consequence, the HLS toolchain schedules for the worst possible scenario. While this pessimistic scheduling approach produces valid schedules for all possible inputs, it cannot be used reasonably to synthesize processor cores. Since an instruction set simulator's control-flow and dependencies are very complex, scheduling for the worst case would lead to a significant under-utilization of hardware resources and drastically hinder the generated core's performance. To derive an efficient processor implementation, we need to generate hardware operating under a *dynamic* rather than a static schedule. The next section of this report explores recent results in hardware synthesis that make dynamic scheduling available to HLS tools and techniques that can be used to generate speculative hardware, capitalizing on dynamic execution to further improve execution performance.

3 Synthesizing Speculative Hardware

A fundamental aspect of modern processor performance is speculative execution. Speculation is an execution method used in high-performance processing cores to unveil more instruction-level parallelism, *i.e.* to enable a broader range of instruction to be executed concurrently. As we will see in Section 3.2, speculative execution comes in two forms: control-flow speculation and memory speculation. Several recent techniques to extend HLS tools to derive speculative hardware have been proposed [Josipović et al., 2018, Josipović et al., 2019, Derrien et al., 2020]. In particular, speculative loop pipelining [Derrien et al., 2020] is a promising approach that can handle both control-flow and memory speculations within a classical HLS framework. The remainder of this section gives an overview of state-of-the-art techniques used to synthesize dynamically scheduled (Section 3.1) and speculative hardware (Section 3.2) from a high-level algorithmic specification.

3.1 Dynamic Scheduling

One of the main reasons why instruction set processors excel in control-flow dominated workloads is their ability to quickly adapt their execution flow to external events or unpredictable changes in the input. This kind of decision has to happen at runtime for the hardware to see the input data and adapt the instruction schedule to potential variations. Compile-time scheduling such as the static scheduling approaches implemented by HLS tools and presented in Section 2.3 are therefore unfit for this type of computations. In this section, we focus on work by [Josipović et al., 2018], which exposes a novel technique for the synthesis of dynamically scheduled circuits in traditional HLS flows. We also look at [Cheng et al., 2020], which aims to combine static and dynamic scheduling into a single HLS framework to generate hardware with a minimal surface area and maximal performance. Dynamic scheduling is the first step towards speculative execution, which we will further discuss in Section 3.2.

3.1.1 Dynamically Scheduled High-Level Synthesis

Static scheduling in HLS tools limits the performance for kernels with dynamic decisions. Dynamic execution is the only way to overcome the overhead introduced by static scheduling in the presence of memory dependencies or on execution paths involving variable-latency operations. In this section, we focus on recent work that showed how dynamic scheduling could be brought to HLS design flows [Josipović et al., 2018]. In this paper, the authors examine the synthesis of dynamically

scheduled elastic circuits from a high-level C description and compare the generated hardware to traditional HLS tools in terms of design complexity and critical path length.

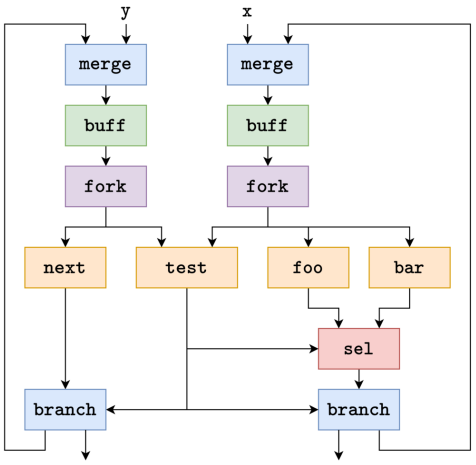
Elastic circuit components are similar to usual datapath elements coupled to a handshaking protocol based on `ready/valid` signal pairs. Handshaking is very common in asynchronous circuit design [Nowick and Singh, 2015], where multiple components need to synchronize without a reference clock signal. Elastic circuits transfer these ideas from the asynchronous domain to synchronous designs governed by a clock. The method described in [Josipović et al., 2018] relies on a small number of elementary building blocks to construct elastic circuits around the concept of token exchange. These basic components include storage units such as elastic buffers and FIFOs, and control-flow components such as branching, merging, and path selection. The authors also introduce elastic components that mimic threaded execution behavior in high-level languages, most notably in the form of fork and join primitives. Figure 4a gives an elastic circuits for the code in figure 2a. The proposed HLS toolchain[4] maps each basic block of the CFG to a set of elastic components.

By relying on elastic primitives, the authors shift scheduling from a centralized FSM to a distributed network of handshake signals. This approach enables fine-grain local decisions to be taken based on circuit input and output. This approach introduces two challenges addressed in the paper:

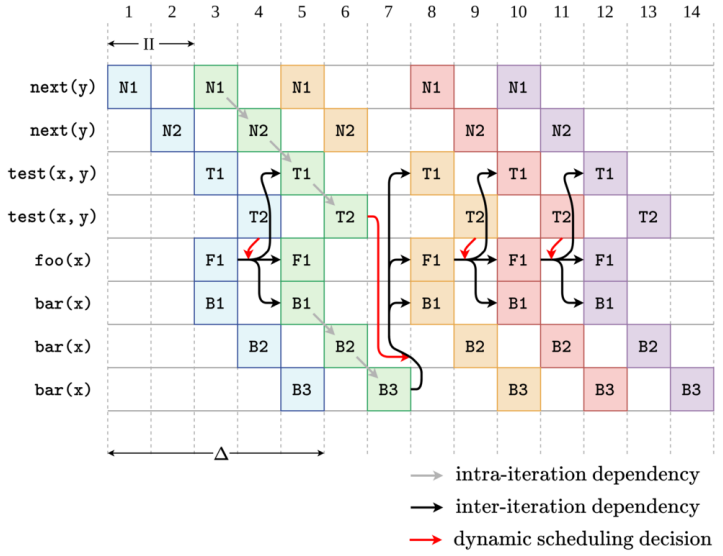
- *Correctness of the generated circuit*: The authors observe that for the synthesized circuit to be semantically correct, tokens propagating in the circuit need to follow the same order as basic blocks in the input program. The proposed implementation propagates tokens through all BBs on a path in the CFG, ensuring that a given basic block only receives data from its immediate predecessors. This approach prevents early token consumption and subsequent deadlocks in the circuit caused by input starvation. Datapaths containing cycles also need to be broken up by elastic buffers to avoid deadlocks.
- *Memory accesses*: When interfacing with memory, the generated circuit needs to ensure the consistency of memory accesses with respect to the input program. The authors introduce an elastic Load-Store Queue (LSQ) component [Josipovic et al., 2017] to maintain consistent memory access ordering. This LSQ keeps track of the current basic block executed by the circuit using a dummy progress indicator token. It allocates new slots in program order, therefore keeping memory accesses ordered even if the dynamic execution were to execute parts of the program out-of-order.

The method described by [Josipović et al., 2018] produces more efficient schedules than traditional static scheduling techniques. Figure 4b illustrates the schedule obtained by applying the technique described in this paper to schedule our example loop code. The interval between two successive loop iterations is shorter when the test succeeds, bringing the effective or average II close to 2 if most of the tests succeed. The circuits generated by the toolchain presented in [Josipović et al., 2018] are such that their effective II never exceeds the II of their statically scheduled counterparts. Additionally, the authors show that the resource cost and clock speed impact of dynamically scheduled HLS is mostly mitigated by the gain in execution performance, thereby presenting an attractive tradeoff for hardware design.

[4] `https://dynamatic.epfl.ch/`



(a) Dataflow circuit. Black arrows represent the flow of data tokens between elastic components. `next`, `test`, `foo` and `bar` are considered as black boxes.



(b) Dynamically scheduled pipeline. The first iteration has a succeeding `test(x, y)` while it fails for the second iteration. The third iteration is delayed until its dependency on `B3` is resolved. II has been reduced to two cycles when `test(x,y)` is `true`.

Figure 4: Dynamic scheduling for the code in figure 2a.

3.1.2 Combining Dynamic and Static Scheduling

Capitalizing on the work presented in Section 3.1.1, a more recent approach [Cheng et al., 2020] exposes a hybrid approach to High-Level Synthesis of hardware under more restrictive resource and area constraints. The authors' key observation is that while dynamic scheduling brings significant performance improvements, it also drastically increases the number of resources needed to propagate and handle handshaking and synchronization between components.

The design method presented by [Cheng et al., 2020] allows parts of a circuit to be synthesized as statically-scheduled components using traditional HLS synthesis techniques. In contrast, the interconnections between those components and the generated design's control logic are dynamically scheduled. This selective replacement of small parts of a dynamic design produces substantial area savings while still providing faster execution times than an entirely static schedule. This hybrid approach leads to highly-effective static scheduling techniques for parts of the circuit that operate on regular workloads while still offering the flexibility of dynamic scheduling parts of the design that need to handle runtime decisions.

3.2 Speculative Hardware Synthesis

As introduced in Section 3.1, dynamic scheduling is the first stepping stone towards the generation of high-performance hardware based on speculative execution. The latter is an essential part of instruction set processor design, with processors needing prediction to enable efficient pipeline usage and execution. This section focuses on state-of-the-art techniques that bring speculative

execution to HLS design flows, allowing hardware designers to synthesize speculative hardware from a high-level description. Section 3.2.1 illustrates speculative execution and highlights some of the challenges introduced by speculative scheduling. Section 3.2.2 takes a look at speculative hardware synthesis by focusing on the work of [Josipović et al., 2019] and [Derrien et al., 2020], describing the different mechanisms introduced by the authors to bring speculation to HLS. Finally, Section 3.2.3 compares speculative dataflow circuits and speculative loop pipelining in the context of ISP synthesis.

3.2.1 Speculative Execution

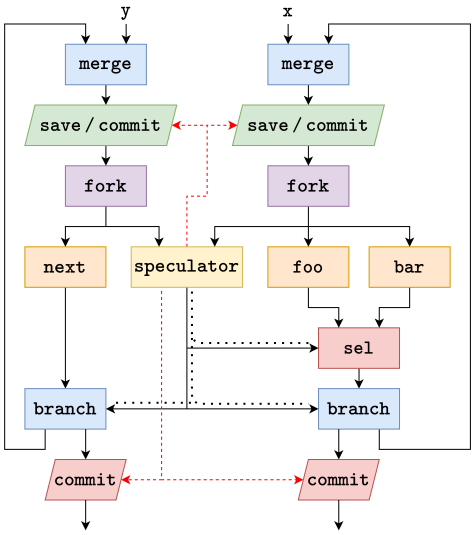
Speculative execution is an execution method used by high-performance computing cores to uncover more instruction-level parallelism. Common wisdom is that speculation is used in superscalar out-of-order processors, but even in-order pipelined processors speculate. **To synthesize an ISP, we need speculation**. Speculation allows the processor to guess the outcome of the execution of a given operation to enable the execution of operations depending on it. We distinguish two types of speculation, namely *control-flow speculation* and *memory speculation*. The former intervenes in cases where, for example, the processor predicts that the condition in a branch is true to start executing the instructions after the branch before the condition is evaluated. The processor may also speculate that a load following a store does not refer to the same address, allowing the load to execute before the store. The latter case is an example of memory speculation. The main difficulty with speculation is that it may be wrong. Consequently, processors supporting speculative execution need to provide both a mechanism to check if a prediction was correct and a mechanism to roll back any effects that the resulting speculation might have had on the program.

We illustrate a possible speculative schedule for figure 2a's code in figure 5b. During the execution, parts of a computation have been started using a false prediction's result. The latter leads to a rollback and stall of the pipeline until the dependency is resolved. This situation is one of three kinds of *hazards* that can occur in a pipeline relying on speculative execution. A pipeline hazard occurs when the program's next instruction cannot be executed in the next clock cycle by the pipelined hardware. There are three different types of pipeline hazards:

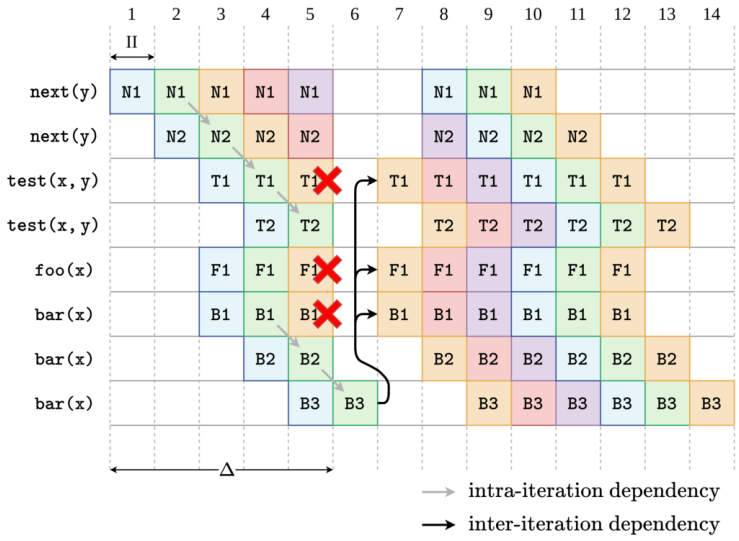
- *structural hazards* occur when the hardware does not support the combination of instructions that the processor wants to execute in the same clock cycle;
- *data hazards* occur when an instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not available yet;
- *control hazards* or branch hazards occur when an instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed. Penalties induced by control hazards are mitigated by the use of prediction and speculative execution.

These pipeline hazards lead to the hardware having to delay the execution of an instruction until the hazard induced constraint is resolved. One way to avoid the penalty of such a delay is to introduce *forwarding*. Forwarding allows parts of a pipelined architecture to bypass some pipeline stages and transfer their result to an earlier stage. This behavior is common in modern processors, where forwarding is used heavily to propagate computational results to subsequent program instructions before writing the result back to main memory [Hennessy and Patterson, 2017].

In codes that exhibit both a slow and a fast path, speculation can drastically increase the generated hardware's throughput and performance. In the next sections, we explore techniques



(a) Speculative dataflow circuit. Black arrows represent the flow of data tokens, with dotted lines used to mark speculative token paths. Red arrows are control signals used by the speculator.



(b) Speculatively scheduled pipeline. The first iteration has `test(x,y)` equal to `true` while the second one has it equal to `false`. The speculation revolves around a control-flow prediction that assumes that the next value of `x` is the result of `foo(x)`. A misspeculation leads to a rollback and pipeline stall after the end of the fifth cycle.

Figure 5: Speculative scheduling for the code in figure 2a.

that can be used to make HLS tools generate such hardware from their usual high-level algorithmic specification input.

3.2.2 Introducing Speculation in Synthesized Hardware

Automating speculative hardware synthesis allows high-performance circuits to be designed more efficiently while also improving hardware designers' ability to sweep through the available design space. Speculative hardware synthesis has been explored by hardware vendors looking to automate the design process of parts of their processor cores, as shown in [Nurvitadhi et al., 2011]. This paper presents a *transactional* model of speculation based on a Domain-Specific Language to describe state components and combinational logic. Though the authors work at a low level of abstraction, their model provides fine-grained speculation support with multiple forwarding and enables them to iterate on a few different speculative pipeline designs easily. This work can be seen as the premise for later developments in speculative hardware synthesis. More recent contributions have paved the way to speculative hardware generation in the context of HLS toolchains. In this section, we take a look at a speculative synthesis applied to elastic circuits [Josipović et al., 2019] and speculative loop pipelining [Derrien et al., 2020].

In [Josipović et al., 2019], the authors build upon previous results presented in Section 3.1.1 to introduce speculative execution support in an experimental HLS toolchain. This paper relies heavily on the framework developed in [Josipović et al., 2018] to provide dynamic execution capabilities to HLS. The authors introduce a new kind of data token to be exchanged with a handshaking protocol

to enable speculative execution in elastic circuits: *speculative tokens*. The latter is generated by a dedicated hardware component named *speculator*, which integrates all the prediction logic required to evaluate a control-flow path speculatively. In addition to issuing speculative tokens in the circuit, speculators also ensure that the predictions made during the execution are correct. If not, they control the rollback logic to revert the current state to a valid one with the help of two additional structural circuit elements, namely *commit units* and *save units*. Commit units are used to retain speculative tokens at critical parts of the circuit until the speculator has validated the corresponding prediction. The speculative token is then converted to a regular data token by the commit unit and forwarded to subsequent computational elements. If the prediction was incorrect, the commit unit simply discards the speculative token. Save units are the counterpart of commit units and are used to store valid data tokens that enter a region of the circuit where speculation may happen. The saved tokens are restored or flushed out depending on the speculator's decision regarding the corresponding speculative decision. Figure 5a shows an example of speculative dataflow circuit generated from the code in figure 2a.

Each time a speculator is inserted into a dataflow circuit, it defines a speculative region. This region is delimited by save units at its entry points and commit units at its outputs. This setup allows the extent of speculation to be well defined in the circuit and avoids possible interferences between multiple speculators. In addition to new elastic components, speculation also mandates that the execution path be marked as carrying a speculative value. The authors introduce a simple marking bit following the datapath and indicating whether a speculator issued the value currently carried by said datapath.

The speculative hardware synthesis method presented in [Josipović et al., 2019] enables HLS tools to introduce speculation generically by adding speculative components to the intermediate dataflow circuit representation of the toolchain. This approach leverages both dynamic execution and prediction to achieve execution similar to what can be found in modern instruction set processors. However, this technique relies on a custom HLS middle-and backend incorporating all the required components to generate speculative dataflow circuits, making it harder to integrate with existing HLS tools. One way to circumvent this limitation is to rely on input program transformation [Derrien et al., 2020] to expose speculation directly at the source level.

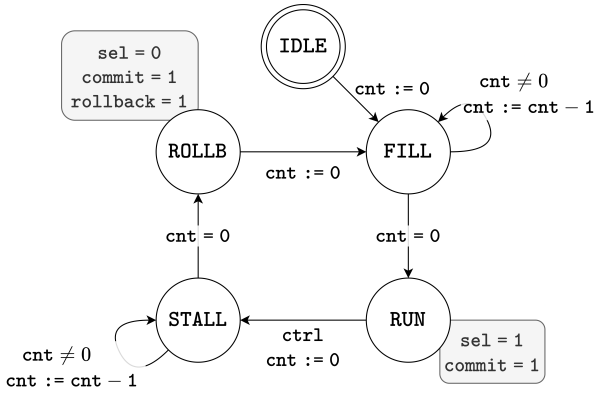
Speculative loop pipelining (SLP) [Derrien et al., 2020] is a hardware synthesis technique that relies on source-to-source program transformations to directly expose speculative behavior in the high-level specification used as an input to the synthesis toolchain. It extends traditional loop pipelining (Section 2.3) with an additional pass aimed at exposing speculation opportunities in strongly-connected components of loops in a program. Figure 6 illustrates this approach on the example code in figure 2a. The initial loop code is transformed to decouple the data and control paths in the execution, mapping data-dependent operations to per-cycle iterations and control decisions to an external finite state machine (FSM). By making each iteration of the loop correspond to one execution cycle, data dependencies and reuse distances become explicit, enabling the HLS toolchain to schedule the speculative circuit efficiently. The entire control path is abstracted away in an FSM represented at the bottom of figure 6b. This FSM handles speculation and triggers rollbacks or commit actions depending on the correctness of the predicted value. It contains four distinct active states:

- the `FILL` state corresponds to the pipeline data fill-up;
- the `RUN` state corresponds to the stationary state of the pipeline, where correct speculations are committed until a misspeculation is detected. In the latter case, the FSM moves to the

```
#pragma hls distance mis_x=3
do {
#pragma hls pipeline II=1
  ctrl[t] = test(s_x[t-2], y[t-2]);
  mis_x[t] = bar(s_x[t-3]);
  s_x[t] = foo(s_x[t-1]);
  y[t] = next(y[t-2]);
  cs = nextstate(cs, ctrl[t]);
  if(cs.rollback) {
    s_x[t] = mis_x[t];
  }
  if(cs.commit) {
    x = cs.sel ? s_x[t-1]
               : mis_x[t];
  }
  t += 1;
} while(!(x && cs.commit));
```

(a) Transformed loop.

```
enum tstate {IDLE, FILL, ...};
struct fsm {
  int3 cnt;
  tstate cs;
  bool commit, rollback, sel;
} cs;
```



(b) Control finite state machine.

Figure 6: Speculative loop pipelining applied to the code in figure 2a.

transient `STALL` state;

- the `STALL` state is used to pause the execution after a misspeculation to wait for the correct value to be available, after which the FSM transitions to the `ROLLB` state;
- the `ROLLB` state restores the pipeline's content in case of a misspeculation, effectively rolling back all computations relying on an incorrect prediction. Once rollback is completed, the pipeline is restarted in the `FILL` state.

The source-to-source transformation described by [Derrien et al., 2020] allows speculative hardware to be generated with regular HLS toolchains. It relies on the HLS toolchain to perform resource allocation and sharing easily. Speculative loop pipelining divides the input program's CFG into strongly connected components (SCC) and applies speculation to each SCC. To automate the detection of potential speculative execution points, SLP relies on a derivative of SSA form to represent programs, namely Gated-SSA [Tu and Padua, 1995]. Gated-SSA replaces φ-nodes in traditional SSA representation by μ-, γ-and η-nodes, while also considering arrays as singular values updated through opaque α-operations. SLP complements the Gated-SSA representation with ρ-nodes. These new language elements are defined as follows:

- $\mu(x_{\text{ext}}, x_{\text{in}})$ replaces φ-nodes at the head of loops and selects either the initial value x_{ext} or the loop-carried $_{\text{in}}$ value for a variable x;
- $\gamma(c, x_{\text{false}}, x_{\text{true}})$ replaces φ-nodes at confluence nodes after conditional statements, selecting either x_{true} or x_{false} depending on the value of the condition c;

- $\eta(c, x_{\text{out}})$ replaces φ-nodes at loop exits and selects the corresponding value of x_{out} when the loop exit condition c is met;
- $\rho(d, c)$ is used to model a rollback with a data buffer d and control c: when $c = 0$, the ρ-node forwards the most recent value of d to its output, and when $c > 0$, it discards the c most recent elements and forwards the value in d stored c iterations in the past;
- $\alpha(a, i, v)$ acts as an assignment to an array, replacing the i-th element of a with v, thereby allowing arrays to be considered as atomic objects.

Using this representation allows a source-to-source compiler to easily manipulate the input program's control flow and transform it through a series of simple changes to the Gated-SSA structure. The SLP transformation modifies the inputs of γ-nodes in SCCs to expose the reuse distance for each data source, as can be seen at lines 4–7 in the code from figure 6a, and creates a *shadow variable* for each speculated live-out variable. The latter corresponds to `mis_x` in figure 6a and is used to compute values along non-speculatively taken paths in case of a misspeculation. SLP then creates the FSM controlling the speculation logic depicted in figure 6b and creates an additional execution path in the program to commit values out of the current SCC. Finally, ρ-nodes are inserted on back-edges of all live-out variables that are not subject to speculation. These nodes handle the rollback logic to recover from a misspeculation.

3.2.3 Discussion

Designing speculative hardware is an inherently challenging problem, as it requires a lot of careful considerations about where to speculate and how to handle potential misspeculations. In more advanced speculation schemes, multiple speculations may also interact during the execution of the same program. This scenario is typical in instruction set processors, where predictors are used to predict the next instruction to fetch for execution or disambiguate register and memory dependencies.

Speculative dataflow circuits [Josipović et al., 2019] and speculative loop pipelining [Derrien et al., 2020] are well-suited for the generation of speculative hardware using HLS design flows. Both approaches perform efficient loop pipelining and support both control-flow speculation and memory disambiguation using a *load-store queue* (LSQ). However, while speculative dataflow circuit synthesis requires a custom HLS backend to generate hardware, SLP is fully compatible with existing HLS tools and can exploit the latter's resource-sharing capabilities where appropriate. Additionally, speculative dataflow circuits do not allow designers to specify the desired pipeline depth: the circuit's operations constrain the latter. SLP brings more flexibility for such design space exploration steps. It also relies solely on static analyses, while speculative dataflow circuits need heuristics for elastic buffer placement and sizing.

The work presented in [Josipović et al., 2019] briefly mentions that the implementation natively supports multiple speculations from the same speculation unit because the design preserves the relative order of tokens, thereby preventing any interchange between speculative decisions in commit and save units. According to the authors, interleaving speculations from multiple speculators should be relatively straightforward by employing a tagging system to identify each speculative token's origin. However, the paper does not describe any concrete example of such an approach. Speculative loop pipelining [Derrien et al., 2020] also provides limited support for multiple speculations in the same strongly connected component of a loop, grouping all predictions and considering them as a single unit for the misspeculation recovery path. The authors acknowledge that this approach

simplifies the overall control logic of the generated design at the cost of an increased misspeculation penalty. Both speculative hardware synthesis techniques explored in this report lay the groundwork for multiple speculation support, but further work is required to refine them to make them applicable to instruction processor design.

4 Conclusion

Speculative execution is an essential part of instruction set processor design, even for simple in-order pipelined architectures. This report has explored several techniques that can be used to bring speculative hardware generation to high-level synthesis toolchains, providing the foundations to make ISP design amenable to HLS. However, before synthesizing an efficient processor core from a high-level description, state-of-the-art speculative hardware synthesis techniques need to be refined to handle multiple interleaved speculations and finer-grain recovery schemes.

We will focus on speculative loop pipelining and its application to instruction set processor synthesis during this internship. Our goal is to generate an in-order pipelined processor core from an instruction set simulator written in the C language. We will explore program transformations in the same framework as SLP to guide the underlying HLS toolchain towards the successful generation of a processor instead of interacting with the HLS backend. The first step of this internship will be to study the speculative elements interacting in an ISS for the RISC-V architecture. We will then explore how SLP can be leveraged to synthesize a processor pipeline from this ISS. Speculative hardware synthesis plays a key role in this last step, as even in-order pipelined processors speculate.

References

[Cheng et al., 2020] Cheng, J., Josipović, L., Constantinides, G. A., Ienne, P., and Wickerson, J. (2020). Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 288–298, New York, NY, USA. Association for Computing Machinery.

[Cloutier and Thomas, 1993] Cloutier, R. J. and Thomas, D. E. (1993). Synthesis of pipelined instruction set processors. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, page 583–588, New York, NY, USA. Association for Computing Machinery.

[Cytron et al., 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

[Derrien et al., 2020] Derrien, S., Marty, T., Rokicki, S., and Yuki, T. (2020). Toward speculative loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4229–4239.

[Hennessy and Patterson, 2017] Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition.

[Huang and Despain, 1993] Huang, I.-J. and Despain, A. M. (1993). Hardware/software resolution of pipeline hazards in pipeline synthesis of instruction set processors. In *Proceedings of the 1993*

IEEE/ACM International Conference on Computer-Aided Design, ICCAD '93, page 594–599, Washington, DC, USA. IEEE Computer Society Press.

[Josipovic et al., 2017] Josipovic, L., Brisk, P., and Ienne, P. (2017). An out-of-order load-store queue for spatial computing. *ACM Trans. Embed. Comput. Syst.*, 16(5s).

[Josipović et al., 2018] Josipović, L., Ghosal, R., and Ienne, P. (2018). Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 127–136, New York, NY, USA. Association for Computing Machinery.

[Josipović et al., 2019] Josipović, L., Guerrieri, A., and Ienne, P. (2019). Speculative dataflow circuits. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 162–171, New York, NY, USA. Association for Computing Machinery.

[Klemm et al., 2007] Klemm, R., Sabugo, J. P., Ahlendorf, H., and Fettweis, G. (2007). Using LISATek for the Design of an ASIP Core including Floating Point Operations. In *MBMV*.

[Lam, 1988] Lam, M. (1988). Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 318–328, New York, NY, USA. Association for Computing Machinery.

[Nowick and Singh, 2015] Nowick, S. M. and Singh, M. (2015). Asynchronous design–part 1: Overview and recent advances. *IEEE Design Test*, 32(3):5–18.

[Nurvitadhi et al., 2011] Nurvitadhi, E., Hoe, J. C., Kam, T., and Lu, S.-L. L. (2011). Automatic pipelining from transactional datapath specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–454.

[Patterson and Hennessy, 2017] Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Rau, 1994] Rau, B. R. (1994). Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, page 63–74, New York, NY, USA. Association for Computing Machinery.

[Tu and Padua, 1995] Tu, P. and Padua, D. (1995). Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, page 414–423, New York, NY, USA. Association for Computing Machinery.