Master research Internship

Internship 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 customized Instruction Set Processors, but the design of these complex hardware pieces is costly and very error-prone. We studied how to take advantage of state-of-the-art work on speculative hardware synthesis during this internship to make Instruction Set Processor design amenable to High-Level Synthesis flows. We looked at the challenges induced by interwinding multiple speculations and explored fine-grain misspeculation recovery schemes, directing our efforts towards small embedded processors based on in-order pipelined architectures. This report introduces the internship's context and discusses our approach to extending Speculative Loop Pipelining to synthesize processor micro-architectures from a high-level behavioral description.

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 wearable products, computers are everywhere. The Internet of Things (IoT) opens many new opportunities for digital products and applications but comes with its own set of challenges for computer designers: devices are expected to handle increasingly large computational workloads while enforcing stringent cost and energy efficiency. The vast majority of IoT platforms rely on low-power Micro-Controller Unit (MCU) families supporting the same Instruction Set Architecture (ISA), e.g., ARM. Different MCUs in the same family expose a wide variety of energy to performance tradeoffs thanks to distinct micro-architectures. Product designers can choose from the available processor designs that best suit their application domain, considering price, energy, and performance constraints.

Most existing MCUs rely on proprietary ISAs, which prevent third parties from freely implementing their customized micro-architecture or deviate from a standardized ISA, thereby hindering innovation. The RISC-V [1] initiative is an effort to address this issue by developing and promoting an open instruction set architecture with its own set of tools. The RISC-V ecosystem is quickly growing and has gained much traction from IoT platform designers, as it permits free customization of both the ISA and the micro-architecture.

The ever-growing success of RISC-V highlights the need for customization in today's digital landscape. With the approaching end of Moore's Law and Dennard Scaling, designers need to consider alternatives to well-established proprietary ISAs as the demand for specialized Instruction Set Processors (ISP) continues to rise. This report exposes our work on the automatic synthesis of custom pipelined processor cores from a high-level behavioral specification. The remainder of this report is organized as follows. Section 2 illustrates the motivation and background behind our work. Section 3 gives an overview of state-of-the-art techniques for the synthesis of speculative hardware, and Section 4 focuses on applying and extending those techniques for the synthesis of ISPs. Section 5 discusses one of the main challenges that we encountered during our work on processor core synthesis, namely the proper handling of multiple intertwined speculations. Finally, Section 6 concludes this report and discusses some preliminary reflections on upcoming work.

2 Motivation and Background

The problem of customizing and retargeting compilers to a new instruction set extension has been widely studied in the late 1990s, and modern compiler infrastructures such as LLVM [Lattner and Adve, 2004] now offer many facilities for this purpose. However, the problem of automatically synthesizing micro-architectures has received much less attention. Although several tools exist for this purpose [Cloutier and Thomas, 1993,Klemm et al., 2007], they are based on low-level structural models of the underlying hardware pipeline. They are not fundamentally different from Hardware Description Language (HDL) based approaches: the processor datapath pipeline organization must be explicit, and hazard management is still left to the designer.

In the meantime, High-Level Synthesis (HLS) technologies, which compile C/C++ code directly to hardware circuits, have continuously improved. Several recent research results have shown that

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

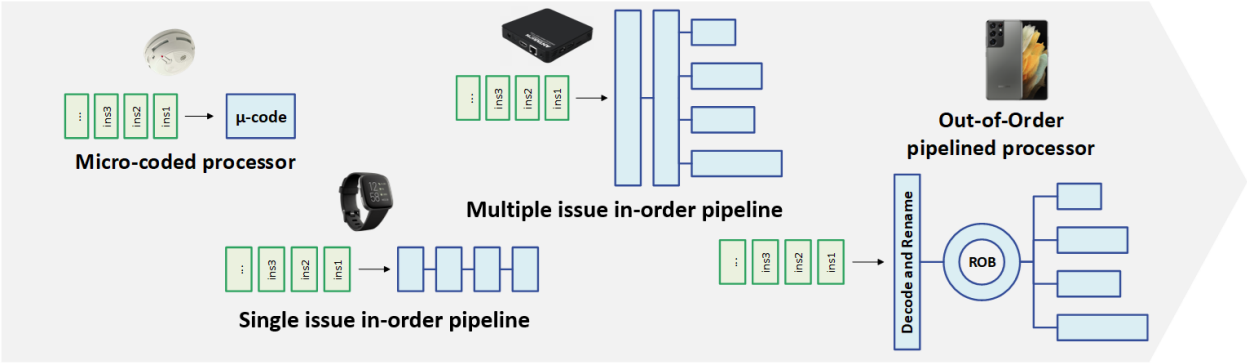


Figure 1: Micro-architectural design space.

HLS techniques could be extended to synthesize efficient speculative hardware structures [Derrien et al., 2020, Josipović et al., 2019]. In particular, speculative loop pipelining [Derrien et al., 2020] appears as a promising approach as it can handle both contr-flow and memory speculations within a classical HLS framework.

In this section, we start by presenting early work on customizing instruction set processors in the context of Application-Specific Instruction Set Processors (ASIP) 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*. Finally, Section 2.4 highlights a few limitations of modern HLS tools for the design and synthesis of ISPs and emphasizes the need for *speculative scheduling* to generate efficient processor cores.

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, making 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. However, 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 an ample 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 Hardware Description Languages (HDL) and asks for explicit management of architectural choices such as pipeline depth and organization, available data forwarding points, and hazard detection logic. We aim at raising the level of abstraction at which designers can specify the working of their desired processor, moving away from a structural description to an entirely behavioral one.

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 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, significantly improving the designer's ability to conduct design space exploration.

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

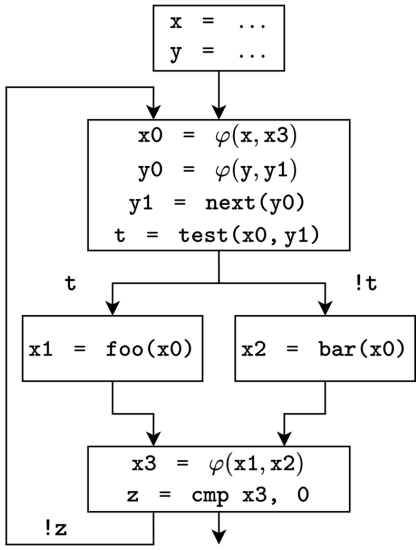
Let us consider an example to get a better feeling for the actual work carried out by an HLS toolchain. Figure 2a shows sample code that we could write in a typical C-based HLS tool to describe a data-processing accelerator. We will use this example as an illustration throughout this report. Given such 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

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

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

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

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



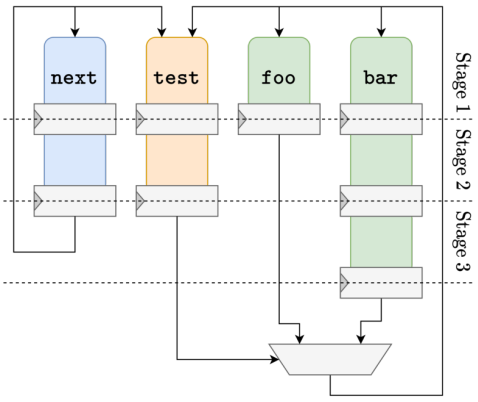
(b) SSA representation of the code in Figure 2a. The HLS toolchain manipulates this internal representation to infer the structure of the hardware.

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

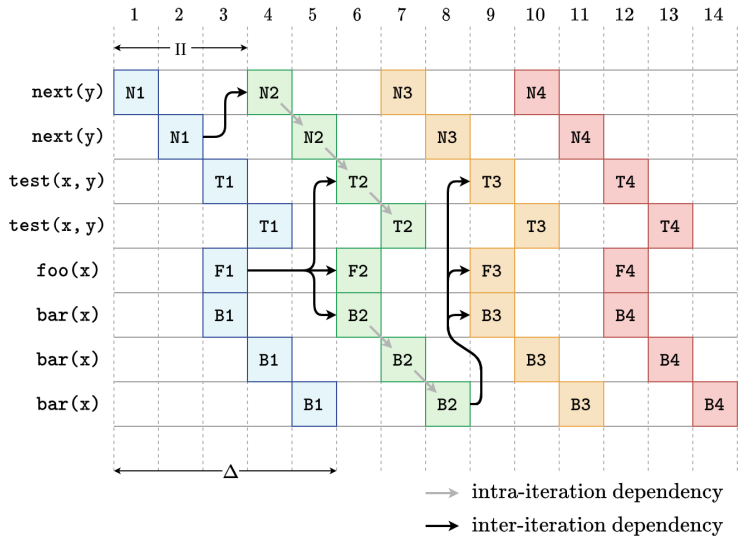
iteration of a loop each cycle. The design's execution frequency depends on each base operator's execution time used in the datapath, thereby determining the clock cycle length. The user can also choose to set the desired clock frequency, thereby constraining the generated circuit layout and potentially increasing the generated hardware area. This area/frequency tradeoff is typical in embedded hardware design. 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 2b).

2.3 Loop Pipelining and Static Scheduling

Figure 3a shows an example pipelined datapath generated from the code in Figure 2a. Each operation is divided into several pipeline stages that are executed in parallel by the resulting circuit. The result of the condition pilots a multiplexer, allowing either the result of `foo` or `bar` to be used as the next value of `x`. A typical HLS toolchain produces a schedule similar to what is given in Figure 3b for this datapath. This structure is akin to the *pipeline* of an instruction set processor 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)` executes in two cycles and is divided into two corresponding pipeline stages. Each colored 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 place-



(a) Pipelined datapath for the code in Figure 2a. Each operator is divided in stages, with each stage corresponding to one clock cycle, as can be seen in Figure 3b.



(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: Pipeline datapath alongside the statically inferred iteration schedule.

ment 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, several factors, including resource availability, target clock frequency, and data dependencies, are constrained by the initiation interval and latency of a given loop. 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. State-of-the-art HLS tools use System of Difference Constraint (SDC) Modulo Schedulers to compute the initiation interval for a given design. SDC is a form of Integer Linear Program (ILP), which can be solved in polynomial time with respect to the problem size [Cong and Zhang, 2006]. The SDC form of the HLS scheduling problem can be expressed as

$$\begin{cases} \text{minimize} & \sum_{i \in I} t_i \\ \text{subject to} & t_i - t_j \leqslant -D_i + b_{i,j}\text{II} \end{cases},$$

where t_i is the starting cycle of operation i, D_i is its delay and $b_{i,j}$ is a measure of intra-loop dependencies between instructions i and j [de Souza Rosa et al., 2019].

2.4 The Need for Speculation

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]. *Speculation* is one of those innovations that have shaped modern processing cores' performance. 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 [Patterson and Hennessy, 2017, McFarlin et al., 2013]. However, such advanced optimizations come with a significant design challenge that cannot be solved efficiently with today's hardware synthesis tools.

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 instructions 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 *speculative* rather than a static schedule.

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 of the resulting speculation on the program. The next section of this report explores recent results in hardware synthesis that make speculative scheduling available to HLS tools and techniques that can be used to generate speculative hardware, capitalizing on ahead-of-time execution to improve performance further.

3 Synthesizing Speculative Hardware

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 concur-

rently. Speculative execution is an essential part of ISP 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 3.1 focuses on dynamic scheduling, which is the first step towards the synthesis of speculative hardware. Section 3.2 illustrates speculative execution and highlights some of the challenges introduced by speculative scheduling. Section 3.3 takes a look at speculative hardware synthesis by focusing on the work of [Josipović et al., 2019] and [Derrien et al., 2020], describing the mechanisms introduced by the authors to bring speculation to HLS.

3.1 Dynamic Scheduling: A First Step Towards Speculative Execution

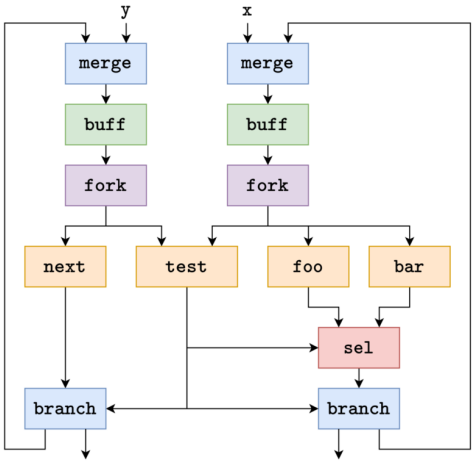
One of the main reasons Instruction Set Processors excel in control-flow-dominated workloads is their ability to quickly adapt their execution flow to external events or unpredictable changes in the input. This kind of decision has to happen at runtime for the hardware to see the input data and adapt 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. 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. Dynamic scheduling is the first step towards speculative execution, which we will further discuss in Section 3.3.

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.

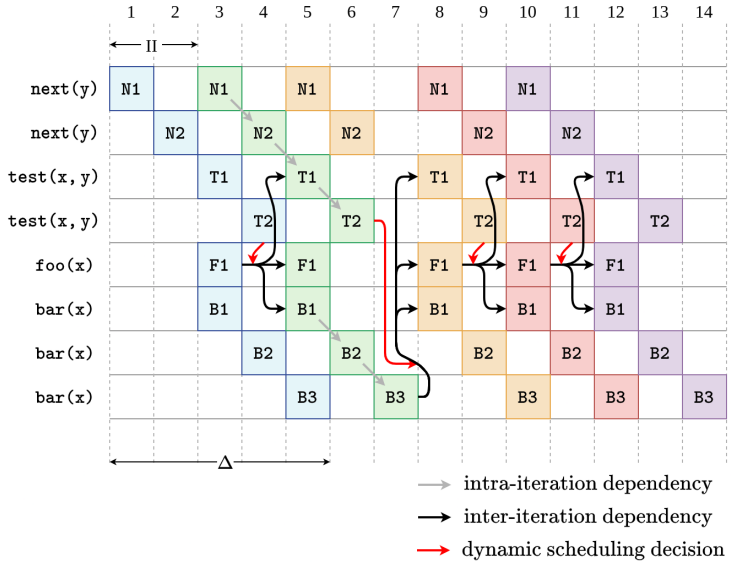
By relying on elastic primitives, the authors shift scheduling from a centralized FSM to a distributed network of handshake signals. This approach enables fine-grain local decisions to be taken based on circuit input and output. This approach introduces two challenges addressed in the paper:

- *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

[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.

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 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 mainly mitigated by the gain in execution performance, presenting an attractive tradeoff for hardware design.

3.2 Speculative Execution

We illustrate a possible speculative schedule for Figure 2a's code in Figure 5a. 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 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 typical 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].

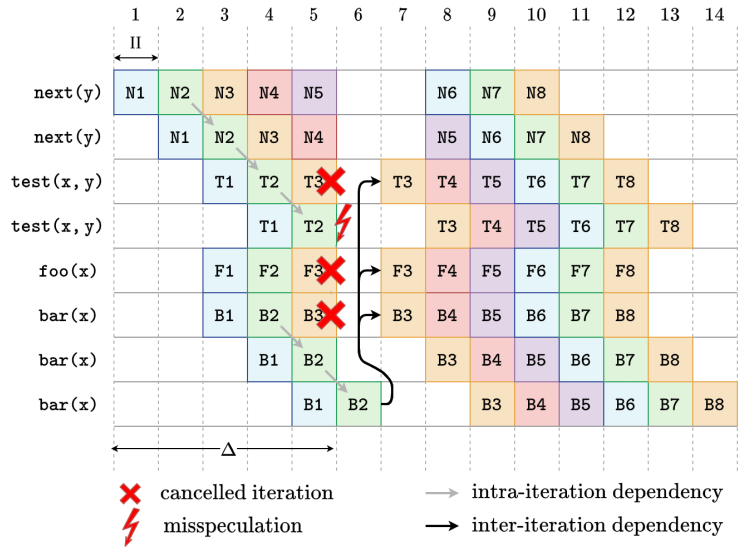
In codes that exhibit both a slow and a fast path, speculation can drastically increase the generated hardware's throughput and performance. In the following sections, we explore techniques that can be used to make HLS tools generate such hardware from a high-level algorithmic specification.

3.3 Speculatively Scheduled Hardware Synthesis

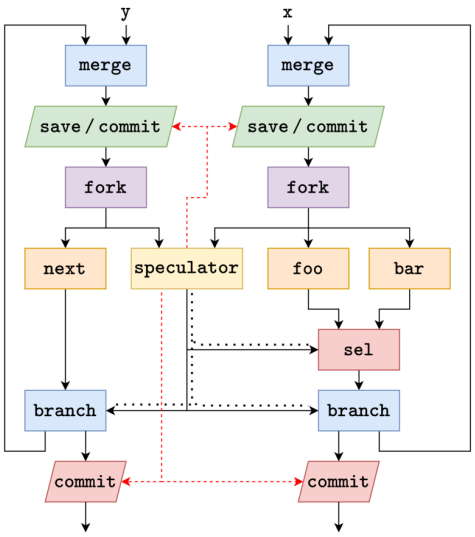
Automating speculative hardware synthesis allows high-performance circuits to be designed more efficiently while also improving hardware designers' ability to sweep through the available design space. Speculative hardware synthesis has been explored by hardware vendors looking to automate the design process of parts of their processor cores, as shown in [Nurvitadhi et al., 2011]. The latter presents a *transactional* model of speculation based on a DSL 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 easily iterate on a few different speculative pipeline designs. 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].

3.3.1 Speculative Dataflow Circuits

In [Josipović et al., 2019], the authors build upon previous results presented in [Josipović et al., 2018] 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



(a) Speculatively scheduled pipeline. The first iteration has `test(x,y)` equal to `true` while the second one has it equal to `false`. The execution assumes that the next value of `x` is the result of `foo(x)`. A mispeculation leads to a rollback and pipeline stall after the end of the fifth cycle.



(b) 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.

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

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 store valid data tokens that enter a circuit region where speculation may happen. The saved tokens are restored or flushed out depending on the speculator's decision regarding the corresponding speculative decision. Figure 5b 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.

3.3.2 Speculative Loop Pipelining

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:

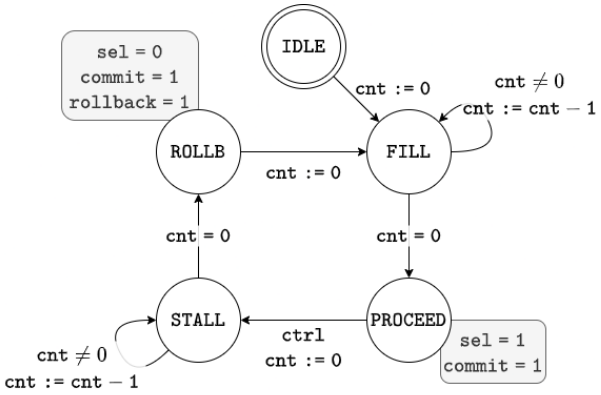
- the `FILL` state corresponds to the pipeline data fill-up;
- the `PROCEED` state corresponds to the stationary state of the pipeline, where correct speculations are committed until a mispeculation is detected. In the latter case, the FSM moves to the transient `STALL` state;
- the `STALL` state is used to pause the execution after a mispeculation 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 mispeculation, 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:

```
1  #pragma hls distance mis_x=3
2  do {
3  #pragma hls pipeline II=1
4    ctrl[t] = test(s_x[t-2], y[t-2]);
5    mis_x[t] = bar(s_x[t-3]);
6    s_x[t] = foo(s_x[t-1]);
7    y[t] = next(y[t-2]);
8    cs = nextstate(cs, ctrl[t]);
9    if(cs.rollback) {
10     s_x[t] = mis_x[t];
11   }
12   if(cs.commit) {
13     x = cs.sel ? s_x[t-1]
14                : mis_x[t];
15   }
16   t += 1;
17 } while(!(x && cs.commit));
```

(a) Transformed loop.

```
1  enum tstate {IDLE, FILL, ...};
2  struct fsm {
3    int3 cnt;
4    tstate cs;
5    bool commit, rollback, sel;
6  } cs;
```

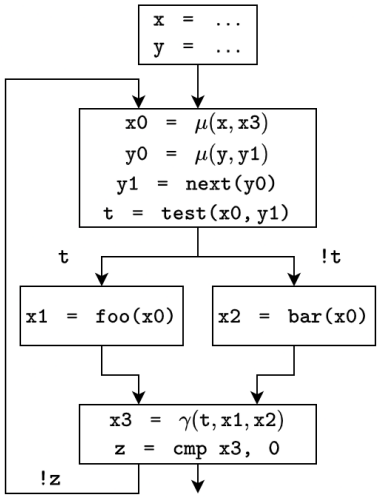


(b) Control finite state machine.

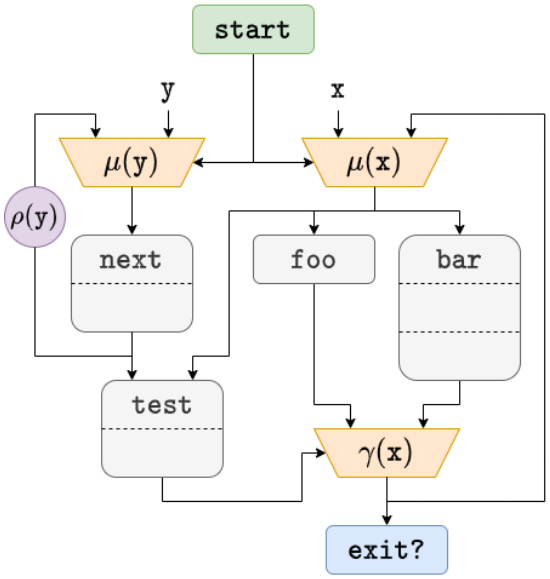
Figure 6: Speculative loop pipelining applied to the code in figure 2a.

- $\mu(x_{\text{ext}}, x_{\text{in}})$ 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 implemented in the GeCoS compiler [Floc'h et al., 2013] modifies γ-node inputs 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 mispeculation. 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 mispeculation.



(a) Gated-SSA representation of the code in Figure 2a.



(b) IDG built from the Gated-SSA representation. The `start` symbol signals the beginning of the loop while the `exit?` node checks for the termination condition.

Figure 7: Gated-SSA and corresponding Instruction Dependency Graph (IDG) for the program in Figure 2a.

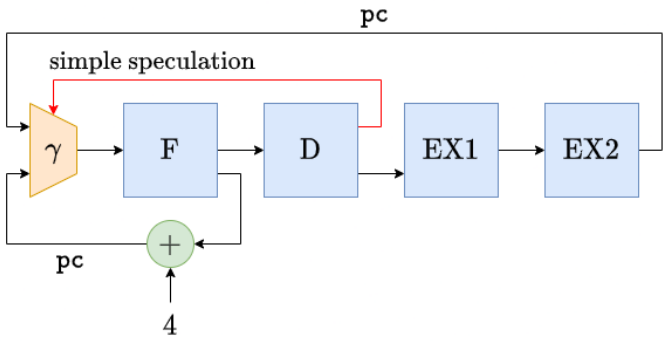
Figure 7a shows the Gated-SSA representation of the program in Figure 2a. φ-nodes have been replaced by their Gated-SSA counterparts and act as delimiters for the loop structure in the SSA graph. Figure 7b gives the resulting Instruction Dependency Graph (IDG), with operation delays as given in the comments of Figure 2a. We note that only x is subject to potential speculation through the γ-node at the end of the loop. If a mispeculation were to occur, y is rolled back to a previous value by the ρ-node on the back edge of the loop.

4 Instruction Set Processor Synthesis

Modern processors are still designed using low-level Hardware Description Languages, which put a significant burden on designers who need to rewrite large parts of the hardware specification when exploring the available design space for the same architecture in different application domains. We aim at replacing this tedious manual work with an automated approach based on HLS flows to automatically synthesize pipelined micro-architectures from an ISS written in C. This section focuses on the generation of ISPs from this high-level behavioral description. More specifically, we look at the typical organization of an ISS and its impact on the speculative structure of the generated processor. Section 4.1 discusses Instruction Set Simulators while Section 4.2 exposes different processor structures, increasing the complexity and the performance of the generated hardware as we add more speculation opportunities.

```
while (1) {
  ir = fetch(pc);
  if (decode(ir) == BR) {
    pc = exec(ir); // slow
  } else {
    pc = pc + 4;   // fast
  }
}
```

(a) Basic ISS.



(b) Structure of the 4-stage pipelined processor.

Figure 8: Pipelined processor with a single speculation on PC.

4.1 Instruction Set Simulators

Since our work focuses on synthesizing Instruction Set Processors, we look at Instruction Set Simulators and their structure to generate speculative hardware. A typical ISS is modeled around the execution flow of a traditional 4-stage pipelined processor:

1. Fetch the next instruction from memory, according to the value of the Program Counter (PC);
2. Separate the retrieved instruction into its elemental parts, decoding opcodes, immediate value, and other elements needed for the proper execution of the instruction;
3. Execute the decoded instruction, matching the opcode and operands against those defined in the ISA specification to take the corresponding action;
4. Write the result of the execution back to the register file or main memory.

This structure naturally exposes multiple speculation paths for us to consider. We can speculate on the next value of PC, fetching instructions before actually knowing if the execution will divert execution, e.g., through a jump instruction. Additionally, the matching operations performed by the execution stage can also be speculated on, speculatively selecting the opcode and even the corresponding operands before the decoding phase ends. Finally, writing results back to memory implies memory speculation to rule out potential aliases and preserve Read-after-Write dependencies during execution.

Speculative Loop Pipelining handles multiple speculations in the same SCC as a single speculation, delaying all γ-nodes until the last one is ready. If a mispeculation occurs, the pipeline is rolled back by a number of iterations corresponding to the maximum number of speculated γ-nodes on the execution path. While this approach simplifies the overall mispeculation recovery and detection logic, it incurs an increased mispeculation penalty compared to more fine-grained approaches. The complex interwinding of speculations implied by the structure of an ISS imposes finer-grain handling of multiple speculations to generate efficient hardware. We proposed an extension to the SLP model during this internship to efficiently handle multiple speculations in generated hardware. We improved the existing memory speculation infrastructure to handle aliases in an ISS.

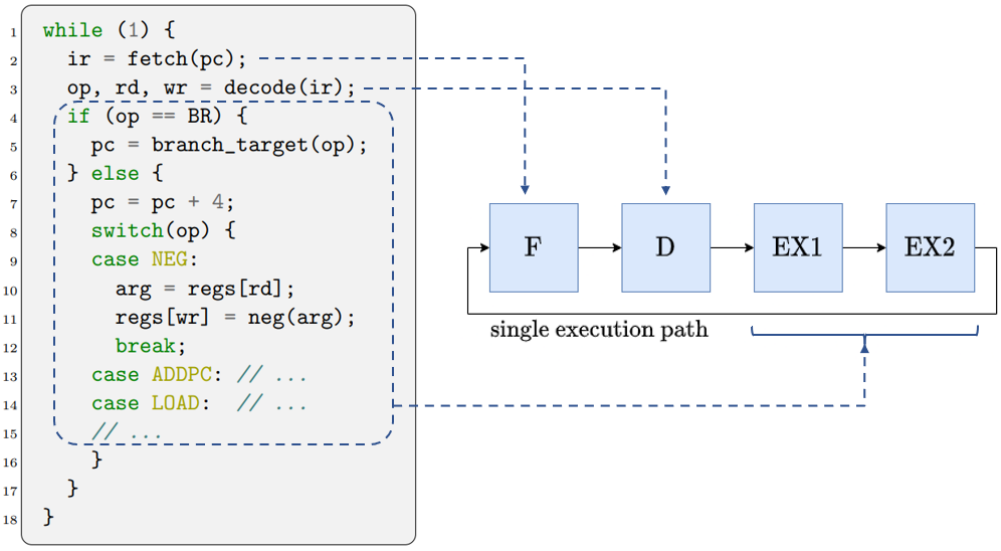


Figure 9: Full Instruction Set Simulator for our 4-stage pipelined processor.

4.2 Speculative Processor Structure

While speculation is commonly associated with high-performance OoO processor cores [McFarlin et al., 2013], even in-order pipelined processors need speculation. In this section, we build the speculative structure of an ISP from the ground up, starting from a simple speculation on the value of the Program Counter (Section 4.2.1) and going through memory speculation, most notably on aliases in the register file (Section 4.2.2). We end by adding forwarding (Section 4.2.3) to our processor core. Section 4.2.4 then discusses the combination of dynamic and speculative decisions inside of an ISP and shows that dynamism can be considered as a particular case of speculation.

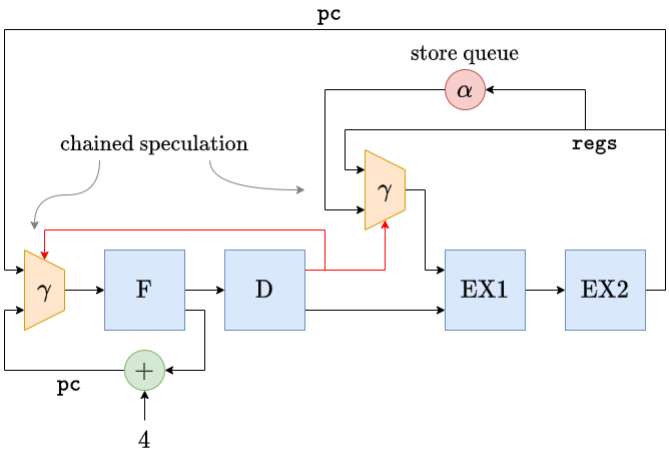
4.2.1 Speculating on the Program Counter

We start building our processor with a simple Instruction Set Simulator shown in Figure 8a. While this processor only supports branch instructions and does not do any helpful computation *per se*, we recognize a speculation opportunity similar to the one in Figure 2a discussed in Section 3.2. The processor's execution can take either a "slow" path, decoding the branch target and setting the value of PC accordingly, or a "fast" path that increments the Program Counter to point to the next instruction in memory, four bytes after the current one. The resulting structure of the processor core is shown in Figure 8b. In this section and the next ones, we will focus on a 4-stage pipeline consisting of a fetch (F) stage and a decode (D) stage followed by two execution stages (EX1 and EX2).

If we were to give the code in Figure 8a to a traditional HLS toolchain, static analysis would determine that an iteration of the outer loop cannot start before the hardware section devoted to the slow path has finished execution. Since the compiler cannot know where branches will be located in memory, the resulting design would be suboptimal. We would need to pause the entire execution to wait for the end of EX1 and EX2 when a simple increment of PC already gives the correct value for the next iteration in most cases. This simple example demonstrates the observation highlighted in Section 3.2 that processors without speculation cannot perform reasonably and that speculative

```
while (1) {
  // ...
  case NEG:
    if (rd == prev_wr)
      arg = ex2;
    else
      arg = regs[rd];
    regs[wr] = ex2
             = neg(arg);
    prev_wr = wr;
    break;
  // ...
}
```

(a) Exposing register alias speculation in the source code.



(b) Structure of the 4-stage pipelined processor with register alias speculation. For memory speculation to work, we need to insert a store queue to buffer pending stores on the register file.

Figure 10: Pipelined processor with a speculation on PC chained with a speculation on the register file.

hardware scheduling is fundamental to ISP synthesis using HLS tools.

4.2.2 Speculating on Register Aliases

The next step to create our processor is to add support for some additional instructions to perform practical computations. To add those instructions, we modify the ISS in Figure 8a to perform a match on the opcode of the decoded instruction with the set of opcodes defined in the ISA specification for our architecture. Figure 9 shows the resulting simulator. A standard optimization in Instruction Set Processors is to speculate on memory aliases to determine if an instruction can be executed ahead of time in the absence of a Read-after-Write dependency. Our source-to-source compiler implements a pass that allows us to expose a second speculation opportunity, on top of the speculation on PC, in the code of Figure 9.

The transformation that our compiler performs on the input source code is based on the following observation. Suppose we read a value from a location in memory and write to the same location at the previous iteration. In that case, we can avoid having to access memory and directly use the previously written value in the current iteration. Figure 10a shows the code corresponding to the `NEG` opcode in the ISS of Figure 9 after said transformation. In C, we introduce a temporary variable `ex2` which is used to hold the result of the previous execution of the EX2 stage. Then, by checking whether the current read location and the previous write location are the same, we can use the cached result instead of querying the register file again.

We note that, contrary to the speculation on PC illustrated in Figure 8a, there is no clear distinction between the fast and slow path in Figure 10a. Using `ex2` instead of querying the register file may be slower than directly accessing the appropriate register. The latency of the `if` path in the ISS is directly linked to the latency of the execute stage of the generated pipeline,

while the `else` path's latency is the latency of a memory access into the register file. Our compiler chooses either one depending on the statically determined latency of the execute stage and on the register access time.

Figure 10b shows the IDG representation of the transformed code in Figure 10a. There are now two successive speculations in our processor, which require special care when scheduling the execution, as we will see in Section 5.3. We handle alias speculation on the register file by introducing an α-node in the design, buffering pending stores on the register file, and updating the register file as if it were a single value thanks to the α-node. The latter acts as a Store Queue (SQ) and allows us to revert stores in case of a mispeculation. Changes to the underlying memory buffer are only applied when the computation results need to be committed. The same principles can be used to speculate on aliases when accessing main memory.

4.2.3 Data Forwarding

Nearly all processors make use of some form of *data forwarding* between different stages in their pipeline. Forwarding allows the result of one stage of the pipeline to be directly used by another stage without propagating through the entire pipeline structure first. Processor design tools use forwarding extensively to reduce pipeline stalls caused by RaW dependencies [Nurvitadhi et al., 2011]. We introduce a compiler pass that enables forwarding in the hardware generated by our toolchain without additional user interaction. The forwarding transformation coupled to the register file transformation discussed in Section 4.2.2 produces the code in Figure 11a for the `NEG` case of the ISS in Figure 9. The resulting pipeline structure is given in Figure 11b.

There are three γ-nodes in our final processor, each one exposing a different speculation opportunity. These intertwined speculations produce complex interaction patterns that we need to understand to generate the FSM controlling the execution. We discuss the challenges introduced by multiple interacting speculations in section 5.

4.2.4 Combining Dynamic and Speculative Decisions

Our processor's ISS contains an additional conditional statement in addition to the speculation paths depicted in Figure 11, namely the `switch` statement matching the opcode of the current operation. We could speculate on which opcode is to be executed next and start executing the corresponding `case` block before the end of the test. However, we note that such a matching operation can be guaranteed to take only one execution cycle. Applying our source-to-source transformations to the opcode matching code exposes an edge case in the SLP model: speculation with a unit conditional latency behaves as *dynamic execution*.

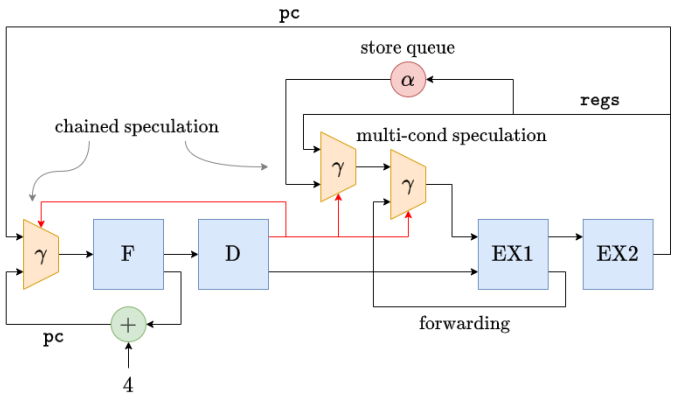
We extend the existing SLP model to handle unit condition latencies in the generated FSM properly. This extension allows us to support the entire range of dynamic and speculative execution scenarii, which gives us greater flexibility when generating hardware. In particular, most `switch` statements in an Instruction Set Simulator can be translated to dynamic decisions, adding additional speculation opportunities to the ones illustrated in Figure 11.

5 Handling Multiple Speculations

Multiple intertwined speculations mechanically occur when trying to synthesize even quite simple ISPs. This section gives an overview of the design space exploration opportunities around multiple

```
while (1) {
  // ...
  case NEG:
    if (rd == prev_wr &&
        ex1_data_avail(rd))
      arg = ex1;
    else {
      if (rd == prev_wr)
        arg = ex2;
      else
        arg = regs[rd];
    }
    ex1 = exec1(arg);
    set_ex1_data_avail(rd);
    ex2 = exec2(ex1);
    regs[wr] = ex2;
    prev_wr = wr;
    break;
  // ...
}
```

(a) Adding forwarding to the processor directly in the source code.



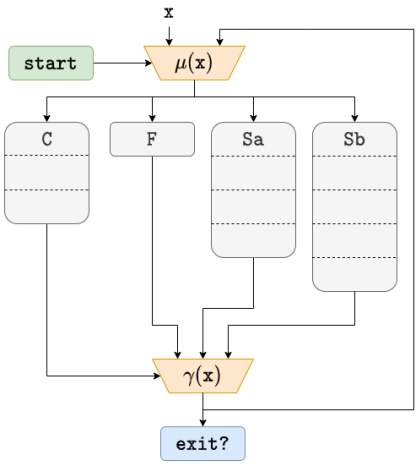
(b) Structure of the 4-stage pipelined processor with register alias speculation and data forwarding. The three γ-nodes in the IDG interact with one another to produce chained and multiple intertwined speculations.

Figure 11: Pipelined processor with a speculation on PC, speculation on the register file and forwarding.

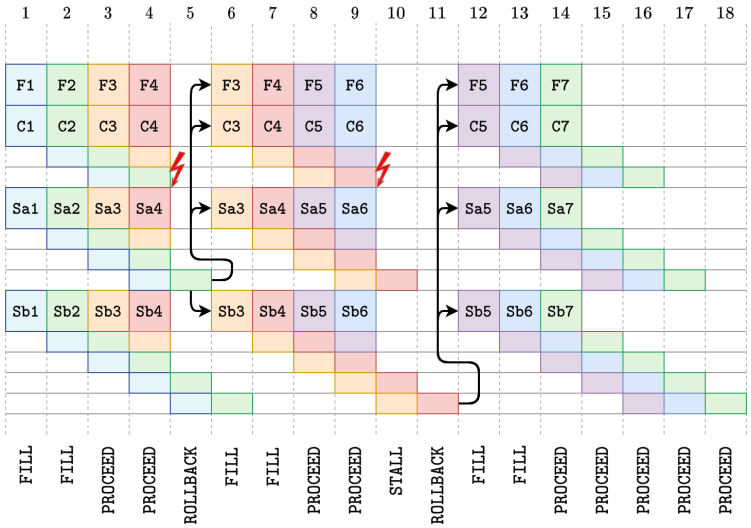
speculation handling exposed by our work on the GeCoS source-to-source compiler. Section 5.1 discusses an extension to the SLP model that allows us to merge multiple speculative paths. The following sections give an overview of multiple speculation scenarii that arise during the synthesis of ISPs. Combining multiple speculations in the same hardware design is an intricate problem, as mispeculations and the corresponding rollbacks need to be carefully handled to avoid trashing useful data in the pipeline during the execution. We distinguish two classes of multiple speculations: *multi-conditional speculations*, where chained γ-nodes are controlled by different conditional operations, and *chained speculations*, where the result of one speculation is used to compute the inputs to subsequent speculations. Both speculation types appear in the 4-stage pipelined processor given in Figure 11b. We give an overview of multi-conditional speculations in Section 5.2 and focus on chained speculations in Section 5.3. Section 5.4 details a few optimizations that we developed to improve multiple speculation handling in synthesized hardware.

5.1 Fusing Multiple Speculation Paths

Speculative Loop Pipelining assumes that each γ-node in its internal IDG representation corresponds to a conditional operator in the input code with a fast and a slow path attached. A consequence of this assumption is that `switch` statements, for example, need to be converted to nested



(a) Multi-path speculation IDG.



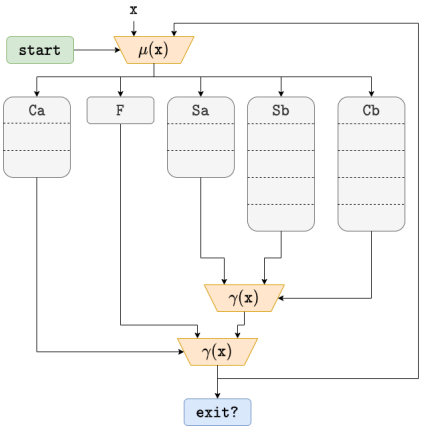
(b) Simple speculation with multiple slow delay values.

Figure 12: Single speculation with multiple execution paths controlled by the same conditional.

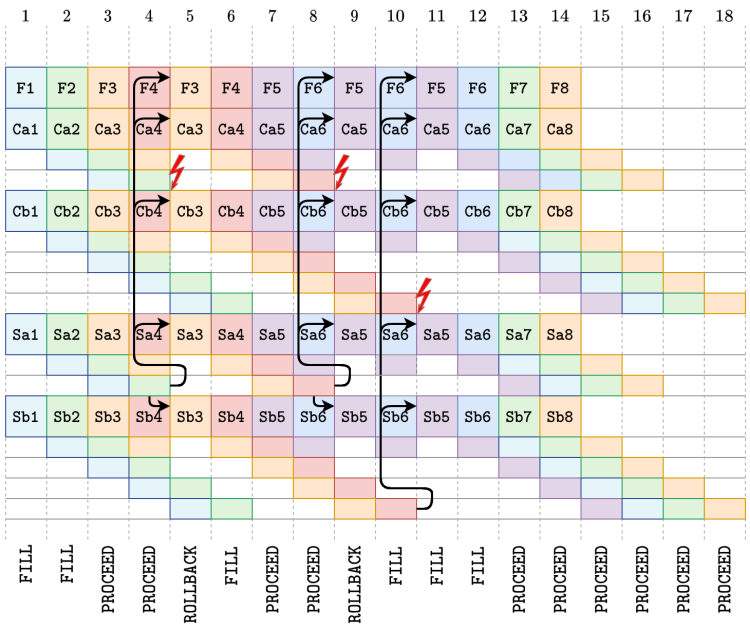
`if/else` statements that are subsequently translated to multiple γ-nodes in the IDG. In order to simplify the control logic, we merge γ-nodes controlled by the same conditional node in the IDG. This transformation enables our FSM to handle multiple speculations on the same condition as a single speculation with multiple possible speculative paths. We sort inputs to the merged γ-nodes by increasing the value of II, increasing the speculated input number with each mispeculation. In the following, we will call such speculation scenarii *multipath speculations*.

Figure 12 illustrates this situation on a simple example with a single variable, `x`, taking on one of three different values depending on the value of a condition, `C(x)`. We extend the regular SLP model to include n-ary γ-nodes such that we can express IDGs in the form of Figure 12a. This structure can be seen as a `switch` statement in C, with the condition determining which `case` branch is taken to compute the next value of `x`. The condition takes three cycles to execute, while `F(x)`, `Sa(x)` and `Sb(x)` take respectively one, four and five cycles to complete their execution. Figure 12b shows an example execution trace for this IDG. By default, the speculation assumes that the condition always selects the fastest path, computing the next value of `x` using `F`. In cycle 4, a first mispeculation happens as the condition for the second iteration resolves and selects `Sa(x)` as the proper value to use for the third iteration. The third and fourth iterations are subsequently canceled and rolled back. The third iteration starts again with the new value of `x`. A second mispeculation happens when the condition of the fourth iterations resolves to select `Sb(x)` instead of the speculatively selected `F(x)`. This time the pipeline is stalled, waiting for the longest path to finish before proceeding to the fifth iteration.

The bottom of Figure 12b shows the state of the control FSM generated by our toolchain at each cycle. The `FILL` state is kept until the first condition resolves in the `PROCEED` state. If there is no mispeculation, we stay in the `PROCEED` state; otherwise, we compute the number of cycles for which the pipeline needs to be stalled before the result of the slow path is available for the next iteration. The pipeline is then stalled for the given number of cycles before rolling back the



(a) Multi-conditional speculation IDG.



(b) Execution trace of a multi-conditional speculation.

Figure 13: Multiple speculation with different conditional latencies.

computations to start over with the correct input value. At cycle 4 in Figure 12b, the pipeline is stalled for one cycle while the FSM transitions from the `PROCEED` to the `ROLLBACK` state, while it is stalled for two cycles for the second mispeculation, at cycle 9.

We note that multipath speculations like the one depicted in Figure 12a can easily be reduced to simple speculations by fusing the `Sa` and `Sb` paths in the generated circuit. However, fusing these two paths leads to a slightly increased mispeculation penalty, as the longest of the fused paths dictates the rollback distance for the considered γ-node. Multipath speculations, therefore, expose a first design space exploration opportunity with a tradeoff between mispeculation penalty and area overhead.

5.2 Multi-Conditional Speculations

The processor structure that emerged from our source-to-source transformations in Figure 11b exhibits an interesting speculative construct that enters the EX1 stage. Two γ-nodes are following each other and are controlled by potentially different control signals, the first one selecting between the store queue and the no-alias path and the second one choosing between forwarding or accessing the result of the previous execution. This pattern, which we call *multi-conditional speculation*, quite commonly arises when translating nested conditional statements to their Instruction Dependency Graph representation. This section discusses the interaction between the two γ-nodes involved in such a speculative pattern.

Successive γ-nodes present us with new challenges related to the generation of the control logic for the speculative circuit. One speculative decision can now be directly dependent on the

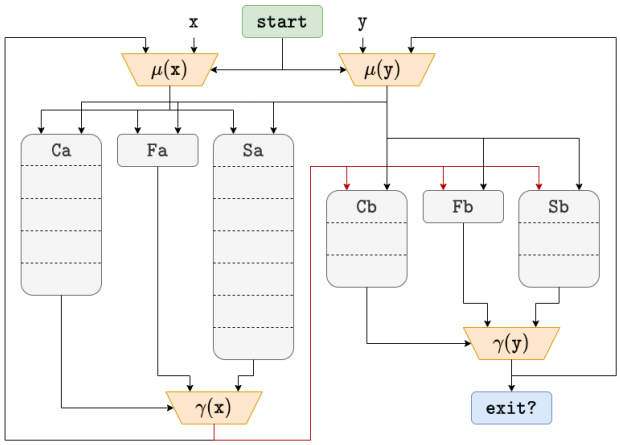


Figure 14: Instruction Dependency Graph for a chained speculation scenario.

outcome of another one. In the following, we will consider the IDG given in Figure 13a. This example illustrates the pattern observed in the processor that we built in section 4.2.3 when adding forwarding to our model. We consider three different data paths, namely `F`, `Sa` and `Sb` with $\Delta_{\texttt{Sa}} < \Delta_{\texttt{Sb}}$. The latency of each path is one, three, and five cycles, respectively. The two successive γ-nodes are controlled by two conditional operators, `Ca` and `Cb`, which are divided into three and five execution stages, respectively. We note that each γ-node follows the basic speculation pattern required by the classical Speculative Loop Pipelining model [Derrien et al., 2020]: there are two paths for each decision with distinct execution latencies. In the standard SLP framework, only the bottom γ-node would be speculated. At the same time, the top one would be translated to a simple multiplexer in the final design, effectively creating an opaque box encompassing `Sa`, `Sb` and `Cb` with a total latency of five cycles. We introduce a finer-grain speculation model and treat both γ-nodes as potential speculation candidates.

Figure 13b gives an example execution trace in our model for the circuit shown in Figure 13a. There are three different possible mispeculation scenarii in this design:

- `Ca` resolves as a mispeculation while `Cb` resolves as a correct speculation;
- `Ca` resolves as a correct speculation while `Cb` resolves as a mispeculation;
- both `Ca` and `Cb` resolve as mispeculations.

The execution trace in Figure 13b only illustrates the former and the latter case. The case where only `Cb` mispeculates is similar to the single speculation discussed in Section 3.2.

In Figure 13b, we suppose that a first mispeculation happens on `Ca2` at the end of the fourth cycle, ruling out `F2` for the computations of the third iteration of the loop. The generated hardware then speculatively selects `Sa2` to be used for the next iteration, waiting for `Cb2` to resolve to confirm this choice. Since the result of `Sa2` is already available at this stage of the pipeline, there is no need to stall the execution, and we can directly proceed with the third iteration. When `Cb` resolves at the end of the sixth cycle, the previous speculation is determined to be correct, and we continue the execution. The same scenario happens at the end of the eighth cycle, but this time `Cb4` also resolves as a mispeculation. The fifth iteration, which was rolled back when the mispeculation happened on `Ca4`, needs to be reset to use the result of `Sb4` instead of `Sa4`. The correct data is injected into

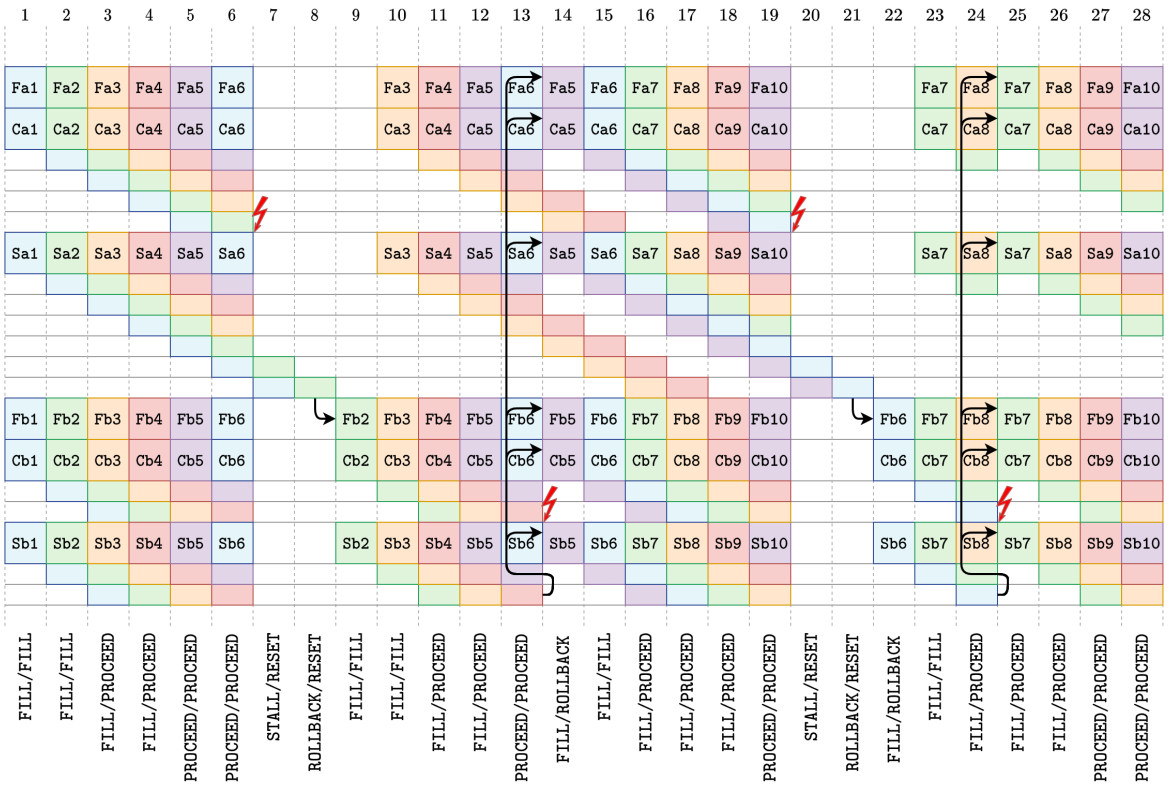


Figure 15: Example execution trace of the chained speculation pattern from Figure 14. The respective states of FSM(x)/FSM(y) are shown on the bottom part. When x is mispeculated over, FSM(y) enters the RESET state.

the computation, and we then resume normal execution. Contrary to the single speculation case discussed in previous sections of this report, this example shows that a rollback can occur even in the middle of the pipeline filling process when a second conditional operator cancels an already rolled back iteration.

The multi-conditional speculation pattern can be reduced to a simple multipath speculation (Section 5.1) by merging the two conditional operators into a single one. Merging the two operators slightly increases the number of cycles required to resolve mispeculations. However, it can significantly simplify the control logic, going from an FSM that can roll back at any point in the execution to the more regular execution scheme of single speculations. This tradeoff highlights another design space exploration opportunity for hardware designers, as using multi-conditional speculations may only be helpful when the difference of latency between Ca and Cb becomes large. We also note that this speculation pattern is only useful when $\Delta_{\texttt{Ca}} < \Delta_{\texttt{Cb}}$. If we were to have $\Delta_{\texttt{Cb}} \geqslant \Delta_{\texttt{Ca}}$, then merging Ca and Cb would result in no losses in the design and reduce the total required die area after synthesis.

5.3 Chained Speculations

The last speculation pattern that appears in Figure 11b is *chained speculation*. Chained speculation is an intricate speculation pattern where the result of one speculation is transformed and subsequently used in another speculative decision. Two variables, `x` and `y`, interact inside of a chained speculation pattern in the program represented by the IDG in Figure 14. Both variables are used to compute the next value of `x` and the latter is then injected into the computation for the next value of `y`. The red edge in Figure 14 illustrates the latter connection.

We introduce two distinct FSMs to handle chained speculations. The first one is attached to $\gamma(\texttt{x})$ and the second one to $\gamma(\texttt{y})$ and both interact to handle the interleaving of speculative values in the overall circuit. We define a *dominance* relation $\prec_{\text{dom}}$ between FSMs, such that

$$\forall x, y \quad \text{FSM}(y) \prec_{\text{dom}} \text{FSM}(x)$$

if there exists a path in the IDG from node $\gamma(x)$ to node $\gamma(y)$ and both nodes are speculated on. In the example of Figure 14, we have $\text{FSM}(y) \prec_{\text{dom}} \text{FSM}(x)$ since the red path exiting from $\gamma(\texttt{x})$ provides a potentially speculative value that is used to compute the speculative next value of `y` through $\gamma(\texttt{y})$. The dominance relation allows us to define a hierarchical ordering on Finite State Machines that need to interact when handling multiple speculations: FSM(`x`) is allowed to *reset* the execution of FSM(`y`) if `x` is mispeculated over. To this effect, we introduce a fifth state in the FSM defined in SLP (Figure 6b), `RESET`, that interrupts the current execution and transitions to the `FILL` state when triggered by a dominating FSM.

Figure 15 gives an example execution trace for chained speculations. Similarly to the multi-conditional speculation discussed in Section 5.2, we distinguish three different mispeculation cases: either `Ca` or `Cb` mispeculate, or they both mispeculate during the same iteration. Figure 15 illustrates what happens when only `Ca2` mispeculates in the second iteration of the outer loop: FSM(`y`) transitions to the `RESET` state while FSM(`x`) stalls execution and rolls back once the result of `Sa2` is available. The computation of `y` then starts again with the correct value of `x`. At the end of the thirteenth cycle, `Cb4` resolves to a mispeculation. The fifth iteration of the loop is subsequently restarted while `Ca4` and `Sa4` are still executing, waiting to check whether the speculation of `x` that produced the value used to compute `y` was correct. The latter proves to be true at the end of cycle 15. If `Ca4` had been resolved to a mispeculation, we would have needed to go back to the beginning of the fifth iteration of the loop once the result of `Sa4` would have been available. Finally, the last mispeculation on `x` happens at the end of cycle 19, requiring a pipeline stall and a subsequent rollback to restart the computation of `y`. The new `Cb6` that ends on cycle 24 is then also resolved to a mispeculation. The latter leads the seventh iteration to be started again on cycle 25.

Chained speculations come with a large pool of different design choices, further expanding the design space exploration space for ISP synthesis. Using retiming [van Antwerpen et al., 2013] on γ-nodes, the circuit in Figure 14 can be transformed to Figure 16. The strong coupling between the output of $\gamma(\texttt{x})$ and the computations for `y` has been eliminated at the cost of duplicating `Fb`, `Sb` and `Cb` and introducing a few additional γ-nodes. The tree of speculative decisions that emerges in Figure 16 can be further transformed by first re-balancing the γ-nodes and noticing that the path used to compute the conditional operators after this transformation can be merged. By merging the conditional paths used to control each γ-node, we end up with the hardware structure depicted in Figure 17. We merge `Ca` and `Fa/Cb` to control the bottom γ-node since selecting the `Fa/Fb` path in the γ-node tree at the top of the figure requires us to know the value of both of these conditional paths. We end up with a merged conditional operator with a latency of $\Delta_1 = 5$ cycles. Similarly,

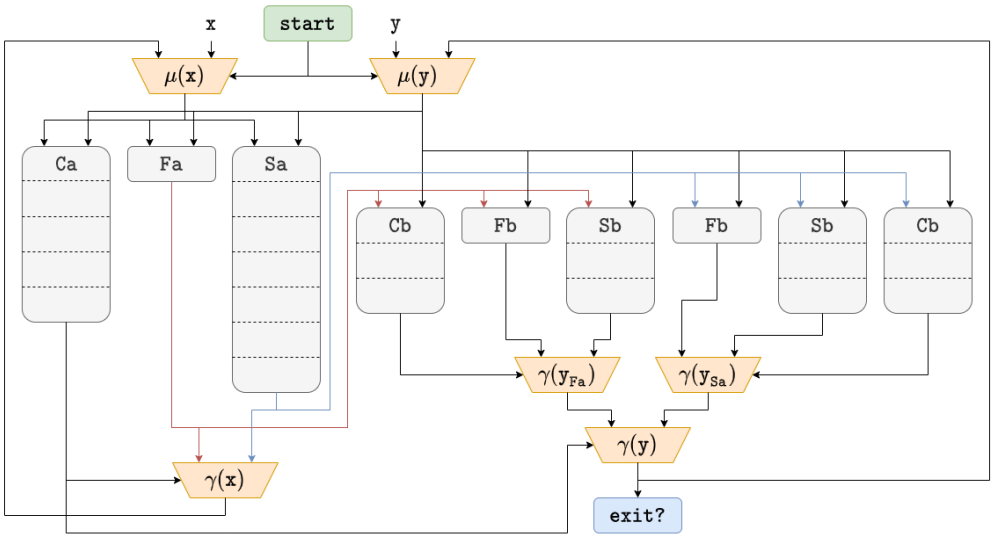


Figure 16: Converting a chained speculation to multi-conditional speculation using γ-node retiming. We can eliminate the need for a second FSM at the cost of duplicating the fast and slow path to compute `y` and by introducing two additional γ-nodes.

choosing between the three entries of the top γ-node in the final design requires a logical combination of `Ca`, `Fa/Cb` and `Sa/Cb`, whose total latency ends up at $\Delta_2 = 10$ cycles. Figure 17 exhibits two of the patterns described in previous sections, namely a multipath speculation (Section 5.1 and a multi-conditional speculation encompassing the two γ-nodes. We note that the condition given at the end of Section 5.2 is verified in the circuit depicted in Figure 17 since the latency of the merged conditional operator controlling the bottom γ-node, Δ_1, is strictly less than the merged conditional latency for the top node, Δ_2. This example shows us that exploiting retiming can widen the design space exploration opportunities we have for micro-architecture synthesis.

5.4 Optimizations

In addition to the extensions we added to Speculative Loop Pipelining to support multiple intertwined speculations and finer-grain mispeculation recovery schemes, we explored several optimization opportunities for the synthesized hardware. An essential design exploration insight lies in the fact that each speculation path in a given hardware design can be explored individually, leading to a set of tradeoffs that a user can explore with the help of our toolchain. This section focuses on four key optimizations to increase the efficiency and cost of the hardware generated by combining our source-to-source compiler and an HLS toolchain. Section 5.4.1 discusses FIFO elimination to reduce the area overhead of the generated hardware, while Section 5.4.2 takes a look at an FSM optimization for multiple speculations and especially chained speculations. Finally, Section 5.4.3 and Section 5.4.4 discuss two techniques that can be used to reduce the cost of the rollback logic in the synthesized design.

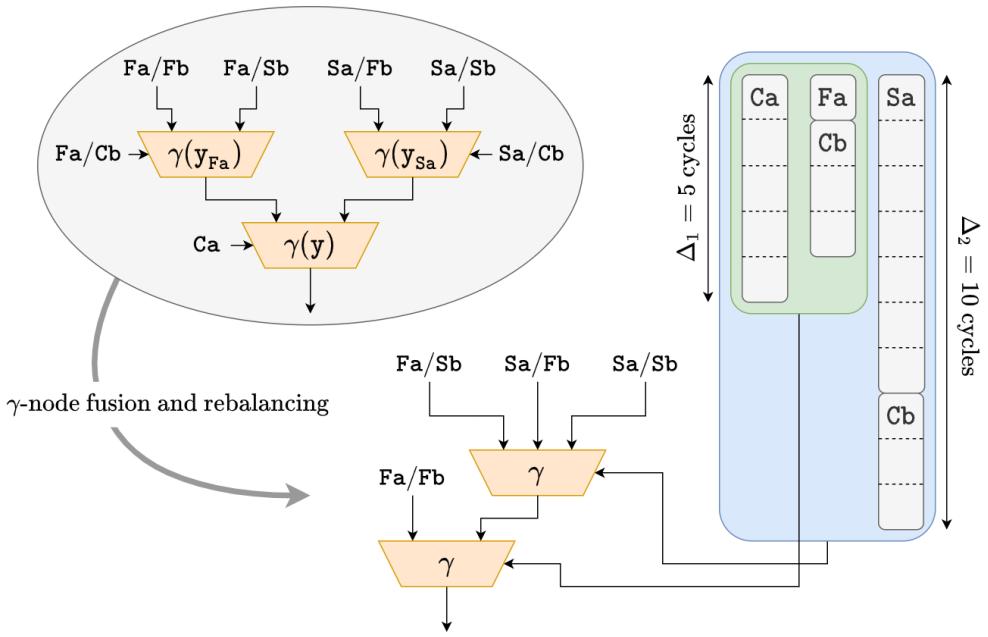


Figure 17: Extracting a multi-path and a multi-conditional speculation from a tree of γ-nodes in the circuit of Figure 16. Inputs to the gamma nodes are not represented and we denote by `Fa/Sb` the fact that `Fa` precedes `Sb` on the path entering the corresponding γ-node.

5.4.1 FIFO Elimination

Speculative Loop Pipelining operates on Strongly Connected Components in the Control-Flow Graph of the input program. Each SCC corresponds to a loop, with some loops that do not expose speculation opportunities. Regardless of the fact that speculation can happen in an SCC, its outputs are connected to other SCCs in the program by FIFOs. The latter incurs a significant area overhead when linking a non-speculative SCC to another SCC in the circuit. We introduce a transformation pass in the GeCoS compiler that merges non-speculative SCCs into speculative ones, thereby reducing the total area used by our design.

The cost of FIFOs can be reduced further by observing that when arrays are involved, a FIFO can be replaced by a rollback buffer. Rollback placement is another instance of the retiming problem that we already encountered in Section 5.3. Consequently, we can exploit retiming in our compiler to place the rollback buffer at a strategic location to minimize the required data handling and control logic.

5.4.2 FSM Optimization

Chained speculations require multiple interacting FSMs to handle the complex interplay of speculative values in hardware. We showed in Section 5.3 and Figure 17 that chained speculation patterns can be transformed to a combination of multipath and multi-conditional speculations. However, this transformation may incur a significant area cost in the generated hardware because of the duplication of parts of the datapath. Another way to transform chained speculation would be to act on the control logic instead of modifying the path that the data travels in the circuit. We can

compute the cartesian product of the FSMs that control the γ-nodes involved in the speculation, eliminating unreachable states while building the combined FSM [Hsieh, 2010]. Further exploration is required to measure the impact of such a transformation on the generated hardware.

5.4.3 Rollback Elimination

We observe that some rollbacks inserted by our toolchain may not be necessary. In the CPU example first introduced in Section 4.2.1, the naive approach would insert a rollback in the final design to revert any changes on the value of the program counter. However, we realize that there is no need to rollback `pc` as it will always get a value either from the increment to the next instruction or from a branch and never go back to a previous value during the execution. Variables with such strong forward progress properties do not require any rollback logic and are characterized by a rollback result being used only by a single γ-node. We can eliminate the corresponding rollback by computing the transitive closure of the adjacency relation in the Instruction Dependency Graph [Nuutila, 1995], starting from the rollback output and checking whether it is used in a single location in the circuit.

5.4.4 Reversible Computations

We say that a hardware operator Ω on variable x is *reversible* if there exists a finite set of hardware operators $W = \{\omega_1, \ldots, \omega_n\}$ such that

$$\exists I = (i_1, \ldots, i_n) \in \mathcal{S}_n \quad \omega_{i_1}(\ldots(\omega_{i_n}(\Omega(x)))\ldots) = x,$$

with S_n denoting the set of all permutations of elements in $[\![1; n]\!]$. Suppose operations executed in the speculative hardware are reversible. In that case, we can avoid using a rollback buffer and instead add operators of W to compute the inverse of an operation Ω on a variable to simulate a rollback. A simple example of this optimization in practice would be to replace the complex rollback and control logic used to revert an arithmetic operation of the form `i += 1`. Instead of keeping a history of the successive values of `i` to revert to an older iteration in case of a mispeculation, we can insert a subtract operator and compute `i -= n` when we need to cancel the last `n` iterations.

6 Conclusion and Future Work

Speculative execution is an essential part of Instruction Set Processor design, even for simple in-order pipelined architectures. During this internship, we 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. Synthesizing an efficient processor core from a high-level description proves challenging, as a typical ISP exhibits complex interleaved speculative patterns. We proposed an extension to the Speculative Loop Pipelining model to handle multiple speculations and fine-grain mispeculation recovery schemes, which will serve as a foundation to efficient processor synthesis from Instruction Set Simulators.

In its current state, our source-to-source compiler can successfully generate elementary Instruction Set Processors that properly handle a few multiple speculation schemes. Some work is still needed to refine and implement the chained speculation model in our toolchain. We hope to provide preliminary results on end-to-end ISP synthesis by the end of this internship. Discussions on

potential future work have also been quite fruitful, with new research directions pointing to further micro-architectural feature syntheses such as branch predictors, and formal verification methods for the designs generated by our toolchain. Existing work on the verification of dynamically scheduled circuits [Josipović et al., 2018] may serve as a good starting point for such work [Cheng et al., 2021]. While we do not have the ambition to generate hardware as optimized as hand-tuned HDL descriptions, we expect to land not far behind such designs in terms of performance and area overhead. We expect to be able to produce code similar to the Comet RISC-V core [Rokicki et al., 2019] soon. Since our approach is based on statically determined features of the input code, we also expect to generate more straightforward and more efficient hardware than other state-of-the-art techniques that rely entirely on dynamic scheduling and execution since many simplifications can already be done at compile time with static analysis of the input code.

References

[Cheng et al., 2021] Cheng, J., Wickerson, J., and Constantinides, G. A. (2021). Probabilistic optimization for high-level synthesis. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 145, 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.

[Cong and Zhang, 2006] Cong, J. and Zhang, Z. (2006). An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, page 433–438, 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.

[de Souza Rosa et al., 2019] de Souza Rosa, L., Bouganis, C.-S., and Bonato, V. (2019). Scaling up modulo scheduling for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):912–925.

[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.

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

[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.

[Hsieh, 2010] Hsieh, S. C. (2010). Product construction of finite-state machines. In *Proc. of the World Congress on Engineering and Computer Science*, pages 141–143.

[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.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86.

[McFarlin et al., 2013] McFarlin, D. S., Tucker, C., and Zilles, C. (2013). Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 241–252, 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.

[Nuutila, 1995] Nuutila, E. (1995). Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124.

[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.

[Rokicki et al., 2019] Rokicki, S., Pala, D., Paturel, J., and Sentieys, O. (2019). What You Simulate Is What You Synthesize: Design of a RISC-V Core from C++ Specifications. In *RISC-V Workshop 2019*, pages 1–2, Zurich, Switzerland.

[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.

[van Antwerpen et al., 2013] van Antwerpen, B., Hutton, M. D., Baeckler, G. W., and Yuan, J. (2013). Register retiming technique (Patent US8806399B1).