Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Utilizing Parametric Systems For Detection of Pipeline Hazards

Lukáš Charvát · Aleš Smrčka · Tomáš Vojnar

Received: date / Accepted: date

Abstract The current stress on having a rapid development cycle for microprocessors featuring pipeline-based execution leads to a high demand of automated techniques supporting the design, including a support for its verification. We present an automated approach that combines static analysis of data paths, SMT solving, and formal verification of parametric systems in order to discover flaws caused by improperly handled data and control hazards between pairs of instructions. In particular, we concentrate on synchronous, single-pipelined microprocessors with in-order execution of instructions. The paper unifies and better formalises our previous works on read-after-write, write-after-read, and write-after-write hazards and extends them to be able to handle control hazards in microprocessors with a single pipeline too. The approach has been implemented in a tool called Hades, and we present promising experimental results obtained using the tool on multiple pipelined microprocessors.

Keywords Microprocessor · Data Hazard · Control Hazard · Formal Methods · Parametric Systems

1 Introduction

As the complexity of hardware designs is growing over the last decades and microprocessor development cycles are facing pressure on fast and low design costs, automation of hardware development has become a crucial need. This trend was further boosted by the recent rise of the so-called Internet of Things (IoT) interconnecting embedded devices, which often require specialised, power-efficient designs to be used. To facilitate the automation, specialized processor description languages [14,25] are used increasingly during the design process. Various tool-chains, such as Synopsys ASIP Designer [26], Cadence Tensilica SDK [8], or Codasip Studio [15] can then take advantage of the availability of such microprocessor descriptions and provide automatic generation of HDL designs, simulators, assemblers, disassemblers, and compilers.

Nowadays, microprocessor design tool-chains typically allow designers to verify designs by *simulation* and/or *functional verification*. Simulation is commonly used to obtain some initial understanding about the design (e.g., to check whether an instruction set contains sufficient instructions). Functional verification usually compares results of large numbers of computations performed by the newly designed microprocessor against a *golden specification*, which must be provided manually by the developers.

However, even extensive functional verification can miss some non-trivial bugs. Therefore, usage of *formal verification* becomes more and more desirable in the recent years. As opposed to testing and bug-hunting techniques that aim at detection of flaws, the goal of formal verification is to rigorously prove that the system is indeed correct. That is, if no issue is found by a formal method, the system is guaranteed to conform to the given specification.

Formal verification is, however, a very demanding task, and even though a lot of progress has been achieved in this area, formal verification is far from being able to fully automatically check all relevant properties of complex designs. Typically, a lot of human intervention is needed in extracting parts of the designs to be verified, constructing abstractions of their environment, and/or helping the automated verification tool in the verification itself. The latter help can come, e.g., in the form of leading a proof when using interactive theorem provers, choosing and fine-tuning the abstraction and verification methods used by model checkers or static

L. Charvát · A. Smrčka · T. Vojnar
Brno University of Technology, Faculty of Information Technology, Božetěchova 2, 612 66 Brno, Czech Republic
E-mail: {icharvat,smrcka,vojnar}@fit.vutbr.cz

analysers, and/or checking possible false alarms produced by such tools.

One way how to mitigate the complexity of automated formal verification is to develop methods specialised in verifying specific features of hardware designs, which is the approach that we take in this work too. The main idea is that, this way, a high degree of automation and scalability can be achieved since only parts of a design related to a specific error are to be investigated.

We, in particular, concentrate on checking the absence of certain kinds of errors in *pipeline-based execution* of instructions in custom-built microprocessors. To be more precise, we restrict ourselves to synchronous microprocessors with in-order execution of instructions that do not contain analog components. Our focus on this area is motivated by that implementation of pipeline-based execution of instructions in purpose-specific microprocessors with customized data and control paths is a rather error-prone task, which implies a special need of proper verification of the resulting designs.

As our first step in the above area, we proposed a fully-automated approach for checking whether pipelined microprocessors execute each instruction correctly provided that it is executing in isolation [9]. Later, we complemented the approach by proposing a method for verifying the absence of *read-after-write* (RAW) [11] and, subsequently, *write-after-read* (WAR) and *write-after-write* (WAW) hazards [12] in microprocessors with a single pipeline. In this paper, we present a unified and better formalised approach of treatment of all of the above mentioned hazard types. Moreover, we extend the approach so that it can now address problems that might be caused by *control hazards* as well.

Our approach starts by a *static analysis* of data paths to detect anomalies and possible hazards. This analysis is followed by a transformation of detected problematic data paths to a *parametric system*: in particular, a system of a parametric (and hence not known in advance) number of instructions to be executed over the detected path, which can be viewed as a linear array of concurrent processes. Finally, we verify whether the detected potential hazards are real by means of techniques for formal verification of parametric systems. We have implemented our approach in a tool called *Hades* [10] and present encouraging results from its experimental evaluations on multiple pipelined microprocessors, including real-life ones.

Plan of the Paper Section 2 presents an overview of the related work addressing validation of single-pipeline microprocessors. Section 3 defines the needed notions. In Section 4, we sketch the main idea of the proposed approach. Sections 5 and 6 discuss pre-processing tasks that are needed before the core steps of our verification approach are applied. These core steps are then described in Section 7. Section 8 presents an experimental evaluation of the proposed approach. Finally, Section 9 concludes the paper.

2 Related Work

Showing the absence of pipeline hazards is a native part of checking conformance between a register transfer level (RTL) design and a formally encoded instruction accurate (ISA) description. The perhaps most cited approach to such checking is the so-called flushing technique [7], which has been extended, e.g., in [18,27,16], to handle rather complicated designs with multi-cycle execution units, exceptions, and branch prediction. The main challenge of these works is to overcome the semantic gap between the different levels of a processor description. Dealing with this issue typically requires a significant user intervention in the form of providing various additional assertions about the design (as can be seen, e.g., in the recent technical report [6]) or transforming it to a purpose-specific description language.

In [17], the so-called self-consistency check that compares possible executions of each instruction in two scenarios is introduced. The comparison is made wrt a property given by the user, e.g., a property concerning data hazards which deals with (i) executions of an instruction enclosed by any (random) instructions within the pipeline and (ii) executions of the same instruction surrounded by `NOP` instructions only. If the self-consistency check succeeds, conformance of the RTL and ISA descriptions of a processor can be established by separately showing conformance of the RTL/ISA descriptions of each individual instruction. The main drawback of the approach is that it requires the enclosing instructions from the first run not to violate a so-called consistent state of the microprocessor, which has to be manually defined by the user.

In [1], a formal model based on a notion of stages, parcels (instructions), and hazards has been introduced. Once the user defines predicates needed for describing the pipeline, the design can be automatically formally proven correct under a correctness criterion given in the work. Another, a bit similar approach has been proposed in [19]. The approach introduces an abstract formal model whose components are to be linked by the user with the concrete cycle-accurate implementation through a number of mappings. Afterwards, interval property checking [23] is used to check several properties implying correctness of the pipeline behaviour. Again, both of the above methods require a significant manual user intervention.

Compared with the above approaches, we do not aim at full conformance checking between RTL and ISA implementations. Instead, we address one specific property—namely, the absence of problems caused by data and control pipeline hazards. On the other hand, our approach is almost fully automated—the only step required from the user

is to identify the architectural resources (such as registers and memory ports) and the program counter.

3 Preliminaries

We now introduce various basic notions that we will build on in the rest of the paper.

3.1 Processor Structure Graphs

In what follows, we expect a processor to be described in the form of a so-called *processor structure graph* (PSG) which can be represented by a tuple $G = (V, E, s, t, \omega)$. Here, V is a finite set that is the union $V_r \cup V_f$ of a set V_r of *registers* and a set V_f of *Boolean circuits*, $V_r \cap V_f = \emptyset$. We distinguish two types of registers: namely, *architectural registers* V_a and *pipeline registers* V_p such that $V_r = V_a \cup V_p$ and $V_a \cap V_p = \emptyset$. The set of architectural registers V_a must always contain exactly one program counter v_{pc}. We expect all registers to have a unit write and zero read delay. Longer access times (e.g., for memory ports) can be modelled by introducing sequentially connected registers emulating the required delay. Boolean circuits represent common combinational logic circuits. For the rest of the paper, it is sufficient to distinguish these circuits into *multiplexers* V_{mx} and all other circuits V_g, referred to as *generic circuits* further on. Hence, we let $V_f = V_{mx} \cup V_g$ while requiring $V_{mx} \cap V_g = \emptyset$.

For registers, we use a well-known notation to characterize their connections: namely, we use $\mathtt{d}$ to denote the data-in, $\mathtt{q}$ data-out, $\mathtt{rst}$ reset, and $\mathtt{en}$ write-enable connections. For multiplexers, we denote by $\mathtt{sel}$ the inbound connection that is the selector which selects one of the input cases $\mathtt{c}_i$ to be transferred from the input to the output of the multiplexer, which is again denoted as $\mathtt{q}$. We denote input connections of generic Boolean circuits as generic inputs $\mathtt{a}_i$. As before, the output is denoted as $\mathtt{q}$. Together, this gives us the set $\mathbb{T} = \{\mathtt{d}, \mathtt{q}, \mathtt{rst}, \mathtt{en}, \mathtt{sel}\} \cup \{\mathtt{a}_i, \mathtt{c}_i \mid i \in \mathbb{N}\}$ of all connection types.

Next, we use E to denote a finite set of *transfer edges*. Note that we do not define the set of edges as $E \subseteq V \times V$ since we sometimes need more edges between two nodes. Instead, we simply require that E is a finite set of some abstract edges, and we assign each edge with its source, target, and type. Namely, we use $s : E \to V \times \mathbb{T}$ to assign to each edge its source vertex and its connection type, and $t : E \to V \times \mathbb{T}$ to assign to each edge its target vertex and its type of connection.

The sets V and E and the functions s and t must fulfil the following criteria. For each register $v_r \in V_r$:

- There is exactly one inbound data-in edge $e_d \in E$ such that $t(e_d) = (v_r, \mathtt{d})$.
- There are arbitrarily many outbound data-out edges $e_q^i \in E$ such that $s(e_q^i) = (v_r, \mathtt{q})$ where $0 \leq i < n$ for some $n \in \mathbb{N}$. (For $n = 0$, we get a register that is only written.)
- There is exactly one inbound clear edge $e_{rst} \in E$, also denoted as the synchronous reset edge, s.t. $t(e_{rst}) = (v_r, \mathtt{rst})$.
- There is exactly one inbound enable edge $e_{en} \in E$ s.t. $t(e_{en}) = (v_r, \mathtt{en})$.

Next, for each Boolean circuit, the following criteria must be satisfied.

- For each circuit $v_g \in V_g$ implementing a Boolean function $g(a_0, \ldots, a_{n-1})$, there is exactly one inbound edge for each argument of g such that $t(e_{a_i}) = (v_g, \mathtt{a}_i)$ for all $0 \leq i < n$ where $n \in \mathbb{N}$. (For $n = 0$, we get a constant function without parameters.)
- Every multiplexer $v_{mx} \in V_{mx}$ that implements a case selection function $switch(sel, case_0, \ldots, case_{n-1})$ has exactly one inbound edge for each of its arguments such that $t(e_{sel}) = (v_{mx}, \mathtt{sel})$ and $t(e_{case_i}) = (v_{mx}, \mathtt{c}_i)$ for all $0 \leq i < n$ where $n \geq 2$.
- For each circuit $v_f \in V_f$, there are arbitrarily many outbound result edges $e_q^i \in E$ such that $s(e_q^i) = (v_f, \mathtt{q})$ where $0 \leq i < n$ for some $n \in \mathbb{N}^+$.
- There is no cycle in the graph consisting of vertices representing Boolean circuits only.

Finally, there are no other types of edges other than the ones described above.

Due to the above restriction to at most one inbound edge for a single connection type, one can use a simpler notation to uniquely describe the edges. In particular, an edge $e \in E$ that satisfies $t(e) = (v, \mathtt{c})$, $v \in V$, $\mathtt{c} \in \mathbb{T}$, can be encoded using the expression $v.\mathtt{c}$. Finally, the function $\omega\colon E \to \mathbb{N}^+$ represents a mapping that assigns some bit-width to all edges of the PSG. The mapping can be naturally expanded to be defined over registers too—namely, we let $\omega(v_r) = \omega(v_r.\mathtt{d})$ for all $v_r \in V_r$. Additionally, it must also hold that $\omega(e_{out}) = \omega(v_r.\mathtt{d})$ for any $(v_r, e_{out}) \in V_r \times \{e \in E \mid s(e) = (v_r, \mathtt{q})\}$.

Since we propose the notion of PSGs to be as simple as possible, it does not take into account *memories* and *memory ports*. Instead, it contains architectural registers, which can be used to represent particular memory cells. In the paper, we assume that a memory is modelled using a finite number of architectural registers representing the cells of the memory. Memory ports are then modelled using additional logic circuits that select the appropriate memory cell using its address. In particular, for a memory with n addressable units, there are architectural registers $m_0, \ldots, m_{n-1} \in V_a$. A read memory port of such a memory is modelled using a single multiplexer circuit $v_{read} \in V_{mx}$ connected to each of the registers representing memory units—for each $m_i, 0 \leq i < n$, there is an edge $e = v_{read}.\mathtt{c}_i$ connect-

ing a multiplexer case with the corresponding memory unit $s(e) = (m_i, \mathtt{q})$. The selector edge $v_{read}.\mathtt{sel}$ then represents a memory address and $v_{read}.\mathtt{q}$ represents the data-out connection of the memory port. A write memory port is modelled by n circuits used to enable writing to a given memory-cell $m_i, 0 \leq i < n$. Each of these circuits implements a Boolean function $(sel = i) \wedge en$, $0 \leq i < n$, where sel represents a memory port address and en enables writing to the memory. A schematic of a write and a read memory port is depicted in Fig. 1.

3.2 The Transition System Induced by a PSG

Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values, and let $\mathbb{B}^n$ denote the set of bit-vectors of size $n \geq 1$. A PSG $G = (V, E, s, t, \omega)$ induces a (finite) *transition system* $(C, \hookrightarrow)$ whose set of states $C = \bigotimes_{v \in V_r} \mathbb{B}^{\omega(v)}$ is the set of configurations of the PSG G and whose transition relation $\hookrightarrow \subseteq C \times C$ is defined later in this section.[1] We use $c[v_r]$ to denote the bit-vector value of the register $v_r \in V_r$ in a configuration $c \in C$. We abuse the notation and write $c[e]$ to denote the value transferred over an edge $e \in E$ in the configuration c as well. Given an edge $e \in E$ such that $s(e) = (v_f, \mathtt{q})$ where $v_f \in V_f$ is a circuit computing a function $fn(a_0, \ldots, a_{n-1})$, $n \in \mathbb{N}$, the value of $c[e]$ can be recursively expressed as $c[e] = fn(c[e_{a_0}], \ldots, c[e_{a_{n-1}}])$ where $e_{a_i} \in E$, $0 \leq i < n$, corresponds to the edge of the i-th parameter of the function fn. In the case that an edge $e \in E$ is an outbound edge of a register $v_r \in V_r$, i.e., $s(e) = (v_r, \mathtt{q})$, we let $c[e] = c[v_r]$.

For each register $v_r \in V_r$ of a bit-width m, $m \geq 1$, we assume the standard *next-state function* $f^{next}_{v_r} \colon \mathbb{B}^{(2 \cdot m + 2)} \to \mathbb{B}^m$ where the register v_r is written a value transferred over the $v_r.\mathtt{d}$ edge iff the $v_r.\mathtt{rst}$ edge transfers "0" and $v_r.\mathtt{en}$ transfers "1" in the given configuration. Next, the value of the register v_r is nullified if the $v_r.\mathtt{rst}$ edge transfers "1". In the following, we will refer to such a transition as register *clearing*. Finally, the register v_r keeps the same value if both $v_r.\mathtt{en}$ and $v_r.\mathtt{rst}$ transfer the value of "0". This will be referred as register *stalling* in the following explanation. When put together, the next state function $f^{next}_{v_r}$ can be formally defined as follows:

$$f^{next}_{v_r}(curr, new, en, rst) := \begin{cases} curr & en = 0 \wedge rst = 0, \\ new & en = 1 \wedge rst = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then, the relation $\hookrightarrow$ contains a transition $c \hookrightarrow c'$ iff $c'[v_r] = f^{next}_{v_r}(c[v_r], c[v_r.\mathtt{d}], c[v_r.\mathtt{en}], c[v_r.\mathtt{rst}])$ for all $v_r \in V_r$.

[1] Note that we do not introduce any notion of initial states of the transition system. This is because we will use the notion of transition systems to help us explore which sequences of transitions of the system $(C, \hookrightarrow)$ are possible while restricting the exploration to relevant states externally to the notion of transition systems (as we will later see in Section 7.2).

Since our approach builds on analysing conditions that hold in certain stages of the execution of a given instruction, we now introduce a notion of edge conditions. An *edge condition* is a pair (e, b), denoted $e \rightsquigarrow b$, meaning that the edge $e \in E$ transfers some value $b \in \mathbb{B}^{\omega(e)}$. By $\mathbb{E}$, we denote the set of all such edge conditions. Further, we define a mapping $\gamma \colon \mathbb{E} \to 2^C$ that assigns each edge condition $(e \rightsquigarrow b) \in \mathbb{E}$ the set of configurations from C in which the edge e transfers the value b, i.e., $\gamma(e \rightsquigarrow b) := \{c \in C \mid c[e] = b\}$. Given a set $K \subseteq \mathbb{E}$, we also use the point-wise extension $\gamma(K) := \bigcap_{k \in K} \gamma(k)$ of γ.

3.3 Parametric Systems

When checking presence of hazards in a processor represented by a PSG, our approach derives as an intermediate model a parametric network of processes operating on a linear topology, which we denote as a parametric system in what follows. In particular, we use a common notion of parametric systems where processes (which, in our case, represent instructions being executed) may perform local transitions as well as universally or existentially guarded global transitions [13,22,2]. Formally, a *parametric system* (PS) is a pair $P = (Q, \Delta)$ where Q is a finite set of states of a process and Δ is a set of transition rules over Q. A *global transition rule* is of the form $\mathbb{Q}_\circ \colon G \models q \to q'$ where $\mathbb{Q} \in \{\forall, \exists\}$, $\circ \in \{\leftarrow$ "left", $\rightarrow$ "right", $\leftrightarrow$ "left or right"$\}$, $G \subseteq Q$, and $q, q' \in Q$. A *local transition rule* is then of the form $q \to q'$ where $q, q' \in Q$.

A PS induces an infinite transition system whose configurations are finite non-empty words over Q, i.e., elements of the set $Q^+ := \{q_1 \ldots q_n \mid 1 \leq i \leq n \wedge q_i \in Q\}$. For any index $1 \leq i \leq n$, the i-th process can change a configuration $q_1 \ldots q_i \ldots q_n$ to a configuration $q_1 \ldots q'_i \ldots q_n$ when it goes from its state q_i to q'_i using some of the transition rules. The rule can be applied only if its guard is satisfied. For example, the meaning of the guard $\exists_\leftrightarrow \colon G$ is "for each state q from the set G, there should be at least one process in the linear topology including the current one so that the process is in the state q". Formally, the guard $\exists_\leftrightarrow \colon G$ is satisfied in the configuration $q_1 \ldots q_i \ldots q_n$ by the i-th process iff $\forall q \in G \; \exists 1 \leq j \leq n : \; q_j = q$. Similarly, the meaning of the guard $\exists_\leftarrow \colon G$ is "for each state q from the set G, there should be at least one process to the left of the current one so that the process is in the state q". Formally, the guard $\exists_\leftarrow \colon G$ is satisfied in the configuration $q_1 \ldots q_i \ldots q_n$ by the i-th process iff $\forall q \in G \; \exists 1 \leq j < i : \; q_j = q$. The meaning of the other guards is defined analogically.

In the following, we assume working with PSs equipped with a *fairness assumption* requiring that every process must perform a step during a transition between two configurations. Intuitively, this corresponds to the fact that all the

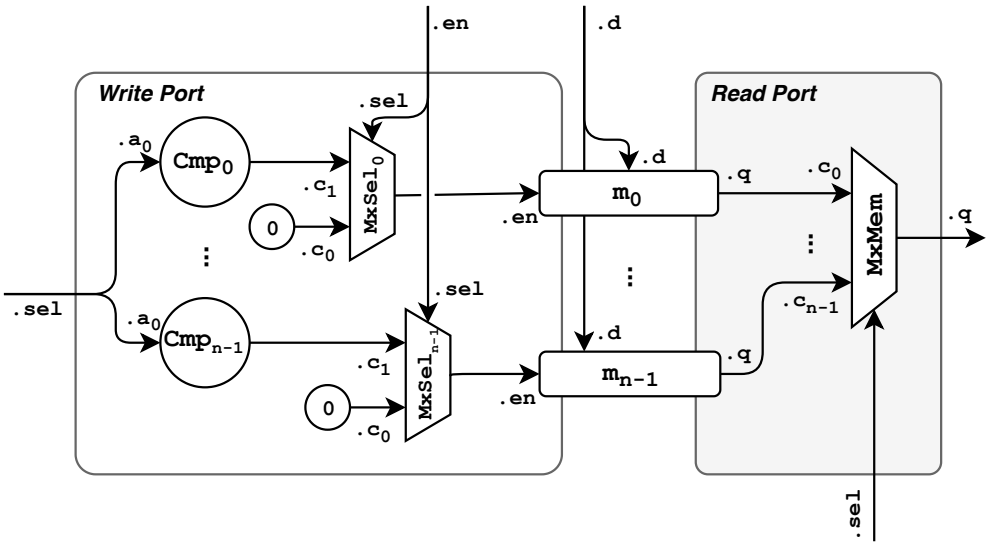


Fig. 1: A schematic of a write and a read memory port.

instructions represented by processes must make a (possibly idling) step in their execution. Moreover, we expect that global transition rules have a *higher priority* than the local ones.

We will reduce the problem of checking existence of a hazard in a given PSG to a *reachability problem* given by a PS P, a regular set $I \subseteq Q^+$ of initial configurations, and a regular set $Bad \subseteq Q^+$ of bad configurations. In particular, we will define Bad as the upward closure of a finite set $B \subseteq Q^+$ of minimal bad configurations where some of the instructions being executed got to an undesirable state. This is, $Bad = \{c \in Q^+ \mid \exists b \in B\colon b \sqsubseteq c\}$ where $\sqsubseteq$ is the usual sub-word relation (i.e., $u \sqsubseteq s_1 ... s_n \Leftrightarrow u = s_{i_1} ... s_{i_k}$ for some $1 \leq i_1 \leq ... \leq i_k \leq n$, $1 \leq k \leq n$). Now, let $R \subseteq Q^+$ denote the set of all *reachable configurations* of the given PS P, i.e., the set of all those configurations that can be reached from some initial configuration from the set I by applying a finite sequence of the transition rules such that the fairness assumptions are respected. We say that the PS P is safe wrt I and Bad iff no bad configuration is reachable, i.e., $R \cap Bad = \emptyset$.

3.4 Data and Control Hazards

Hazards in the instruction pipeline of central processing units (CPUs) are problems caused by inadequate synchronisation of earlier and later instructions running concurrently through the pipeline that may cause potential corruption of the data used by the instructions, with some result of the computation that referred to such data eventually propagated to a programmer-visible register [24]. Three common types of hazards are data hazards, control hazards, and structural hazards. In this article, we will further focus on the first two types of the hazards and on CPU designs that do not use out-of-order execution. We will now give informal definitions of each of the considered hazard types, which we will later formalize in Section 6.

Definition 1 A *read-after-write* (RAW) data hazard is a scenario in which a later-started instruction uses data supposed to be produced by an earlier-started instruction, but the earlier instruction has not yet managed to proceed far enough in the pipeline to write the data into the register used by the later instruction. The later instruction then stores a potentially wrong result of its execution, obtained by dealing with obsolete data, into some programmer-visible register.

Definition 2 A *write-after-read* (WAR) data hazard is a scenario in which some data that should be used by an earlier-started instruction are overwritten by a later-started instruction before the earlier instruction manages to read the data. The earlier instruction then stores a potentially wrong result of its execution, obtained by dealing with data seemingly coming from the future, into some programmer-visible register.

Definition 3 A *write-after-write* (WAW) data hazard is a scenario in which an earlier-started instruction overwrites the result of a later-started instruction that is stored in some programmer-visible register, which then ends up containing obsolete data.

Definition 4 A *control* (CTL) hazard is a scenario where an earlier-started control-flow instruction changes the flow of the control, but some later, speculatively-started instruction manages to store some data into a programmer-visible register.

In commonly used in-order execution designs, the above specified hazards are eliminated by *pipeline stalling* and/or *operand forwarding*. For pipeline stalling, it is necessary for a processor to be equipped with a control logic that determines whether a hazard could/will occur. If such a situation is detected, the control logic inserts no-operation (`NOP`) instruction, sometimes called *bubble*, into the pipeline. Therefore, before the later instruction from the pair of instructions which would cause the hazard executes, the earlier one will have sufficient time to proceed far enough in the pipeline so that the hazard does not happen.

In the case of operand forwarding, additional (redundant) data-paths are introduced into the processor design. These data-paths are aimed to provide an option to propagate partially computed data[2] from an earlier instruction to a later one in order to minimize the number of `NOP` instructions that would otherwise have to be inserted using the above mentioned stalling technique.

4 The Proposed Approach to Hazard Detection

Our approach for verifying that the pipeline logic prevents hazards consists of the following steps: (i) a simple data-flow analysis intended to distinguish particular stages of the pipeline, (ii) a consistency check to make sure that the flow logic guarantees an in-order execution of instructions through the identified pipeline stages, (iii) a static analysis deriving constraints over data-paths of instructions that can potentially cause a pipeline hazard, (iv) generation of a parametric system modelling mutual interactions between potentially conflicting instructions allowed by the derived constraints, and (v) an analysis of the constructed parametric system to see whether the identified interactions may lead to a hazard.

We assume the processor under verification to be represented using a PSG, which can be easily obtained from a description of the processor on the register transfer level (RTL) written in common hardware description languages, such as VHDL or Verilog.

Example 1. Throughout the following sections, we will be illustrating the different steps of our approach on a running example depicted in Fig. 2. The figure shows a PSG describing a part of a simple microprocessor with an accumulator architecture with the following architectural registers: X (a memory index register), A (an accumulator), PC (the program counter), $Prog_i$ (program memory cells), and Mem_j (data memory cells) where $0 \leq i \leq \ell$, $0 \leq j \leq k$ and k, resp. ℓ, are the sizes of the memories[3]. The depicted part of the CPU is used when executing arithmetic and load/store instructions. In order to keep the PSG easily readable, types of connections are shown for architectural registers and case-`c` edges of multiplexers only. Also, since enable (i.e., "`en`") and clear (i.e., "`rst`") connections for pipeline registers[4] are common for each stage, they are left out up to the ones that are required in the further explanation.

In the CPU, the computation starts in Stage 1 by using the content of the program counter PC to address the ith cell of the program memory $Prog_i$. An instruction fetched from the program memory cell is stored into the register $IdIr$ that represents the so-called fetch register. The fetched instruction word in $IdIr$ is then decoded by an instruction decoder in Stage 2. Boolean circuits that belong to the decoder are shown in yellow. Next, an address stored in the index register is used to fetch data from the jth cell of the data memory Mem_j in Stage 3. Optionally, the index register can be *auto-incremented*. The auto-incrementation logic is a feature allowing for an early incrementation of the value of a register for memory addressing just before or right after it is read. We then speak about the so-called pre-/post-increment, respectively. The auto-incrementation feature usually brings a more efficient execution of sequences of instructions that access the processor's memory (for instance, when computing over long arrays or other juxtaposed data). This speed-up results from removing the need of otherwise required pipeline stalls. However, the feature also introduces potential WAW and WAR hazards that must be handled properly. Finally, in Stage 4, the decoded opcode part of the instruction is used to determine the type of an ALU operation (with the ALU itself colored in purple) and to select destination registers by setting their enable connection "`en`" to logical "1".

The Boolean circuit $Flow$ in Fig. 2 represents the flow logic of the second pipeline stage. This logic is responsible for dealing with WAR hazards on the index register X. The flow logic implements the function

$$Flow(IncX, OfWrMem) := \neg IncX \vee \neg OfWrMem.$$

In case a later instruction wants to perform an auto-increment of the index register X while an earlier instruction is going to use the content of X for a memory write, the flow logic uses the enable "`en`" and clear "`rst`" signals of pipeline registers to insert a pipeline bubble between the instructions into Stage 3. ◁

5 Preprocessing a Processor Structure Graph

This section describes the first two steps of the proposed approach: namely, the data-flow analysis identifying pipeline

[2] The data that have not been written to its final register.

[3] We assume that the $Impl$, Or, and Not vertices of the PSG compute the standard implication $f_{impl}\colon \mathbb{B}^2 \to \mathbb{B}$, disjunction $f_{or}\colon \mathbb{B}^2 \to \mathbb{B}$, and negation $f_{not}\colon \mathbb{B} \to \mathbb{B}$ functions. That is, for instance, $f_{impl}(a_0, a_1) := a_0 \Rightarrow a_1$ for $a_0, a_1 \in \mathbb{B}$.

[4] For a full list of pipeline registers, see Table 1 in Section 5.1.

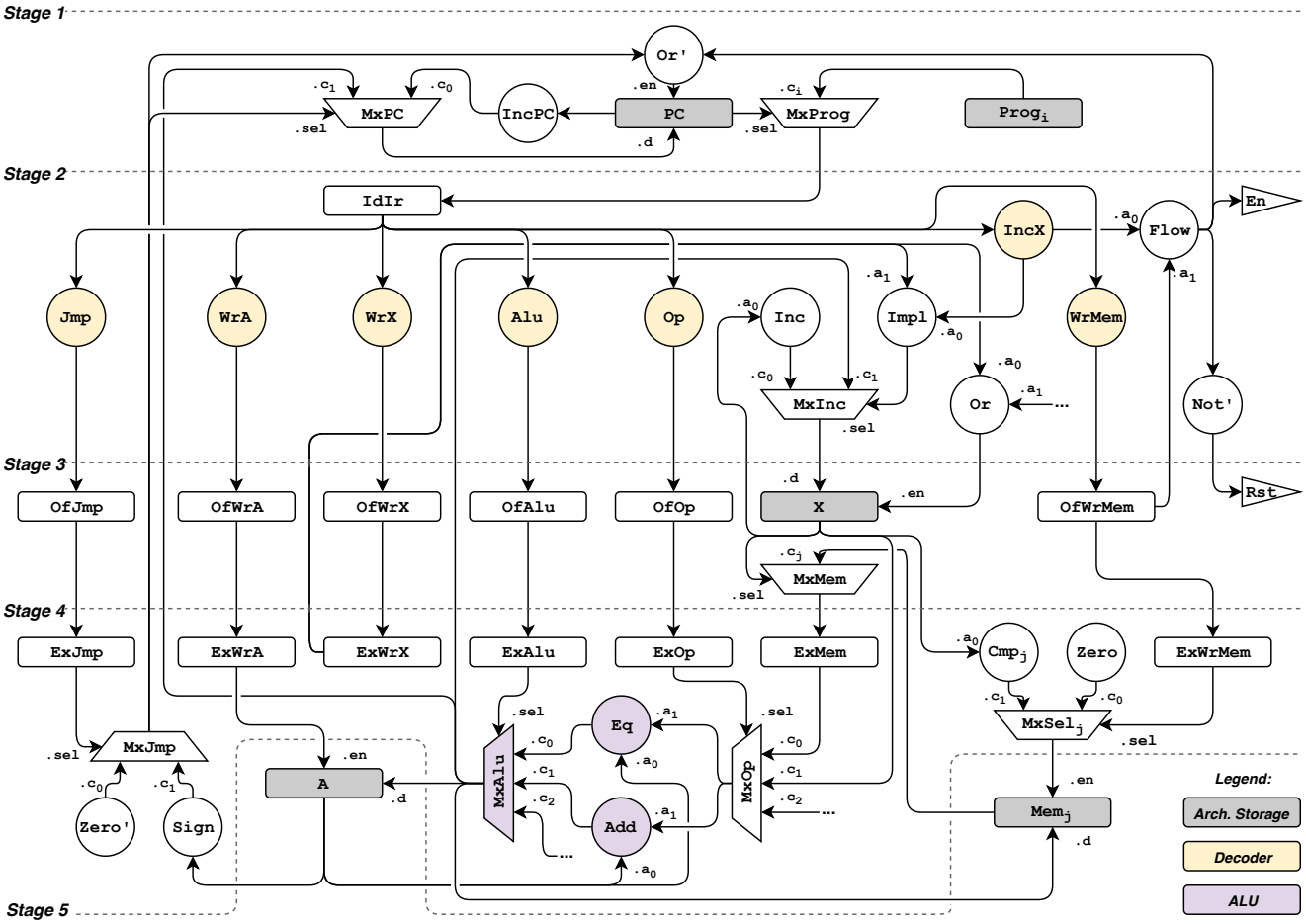


Fig. 2: A processor structure graph of a part of a CPU with an accumulator architecture.

stages and the pipeline consistency check ensuring a proper in-order execution of instructions within the pipeline.

5.1 Data-Flow Analysis Discovering Pipeline Stages

The input of the proposed verification method consists of a PSG and a list of its architectural registers, including the program counter v_{pc}. On this input, the method starts by a simple data flow analysis whose goal is to compute the number of pipeline stages. We then map registers, logic functions, and edges of the PSG into the pipeline stages. We define a *pipeline stage* as the sub-graph of the PSG that is responsible for executing a single-cycle step of an instruction. The pipeline stage that an edge or a vertex (representing a register or circuit) of a PSG belongs to is given by the minimum number of cycles needed to propagate data from the input of the program counter to the edge or the output of the given vertex, respectively. Hence, as a particular case, the program counter itself belongs to Stage 1.

The data-flow analysis that we use starts from the program counter v_{pc} and its Stage 1 and propagates the so-far computed stages forward through the PSG. If several stage values are propagated to a single vertex or edge, the minimum is taken. Whenever a propagated stage value passes a register, it is incremented by one. If there is a register with no path originating from the program counter, such as $Prog_i$ in Fig. 2, then its stage is derived from the lowest stage that reads from that register. For instance, $Prog_i$ is only read by $IdIr$ and so $Prog_i$ will be placed to the stage preceding the one where $IdIr$ is located. The analysis gives us a mapping $\varphi\colon V \cup E \to \mathbb{S}, \mathbb{S} = \{1, \ldots, n\}, n \geq 1$, which maps vertices and edges of the PSG to pipeline stages.

Subsequently, we derive the so-called *write stage* mapping $\varphi^{\text{wr}}\colon V \cup E \to 2^{\mathbb{S}}$ that maps each vertex or edge to the set of stages that directly influence its value. Namely, we include into $\varphi^{\text{wr}}(x)$ the stage of every pipeline register $v_p \in V_p$ from which there is a path to x that does not pass through any further register from V_p. Likewise, we derive the *read stage* mapping $\varphi^{\text{rd}}\colon V \cup E \to 2^{\mathbb{S}}$ for each vertex or edge that describes which stages are directly influenced by its value. In particular, we include into $\varphi^{\text{rd}}(x)$ the stage of

Table 1: Registers of the CPU from Fig. 2 and the corresponding pipeline stages.

Register	Stage φ	Write stages φ^{wr}	Read stages φ^{rd}	Pivot
PC	1	$\{1,2,3,4\}$	$\{1,2\}$	–
$Prog_i$	1	$\emptyset$	$\{2\}$	–
X	3	$\{2,3,4\}$	$\{3,4,5\}$	–
A	5	$\{4\}$	$\{1,2,3,4,5\}$	–
Mem_j	5	$\{4\}$	$\{4\}$	–
$IdIr$	2	$\{1,2,3,4\}$	$\{1,2,3\}$	✓
$OfJmp$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfWrA$	3	$\{2,3,4\}$	$\{4\}$	×
$OfWrX$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfAlu$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfOp$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$OfWrMem$	3	$\{2,3,4\}$	$\{1,2,3,4\}$	✓
$ExJmp$	4	$\{3,4\}$	$\{1,2,3,4\}$	✓
$ExWrA$	4	$\{3,4\}$	$\{5\}$	×
$ExWrX$	4	$\{3,4\}$	$\{1,2,3\}$	✓
$ExAlu$	4	$\{3,4\}$	$\{1,2,3,5\}$	✓
$ExOp$	4	$\{3,4\}$	$\{1,2,3,5\}$	✓
$ExWrMem$	4	$\{3,4\}$	$\{1,2,3,5\}$	✓
$ExMem$	4	$\{3,4\}$	$\{3,5\}$	✓

every pipeline register $v_p \in V_p$ to which there is a path from x that does not pass through any other register from V_p.

Pipeline stages of the registers from the PSG of Fig. 2 and the corresponding read and write stages, computed as described above, are shown in Table 1. (The notion of pivots will be introduced later on.)

5.2 Pipeline Consistency Checking

The second step of our approach is consistency checking which checks whether the flow logic assures a correct in-order execution of all instructions through all the identified pipeline stages. This means that all instructions which are fetched from the program memory should flow from the first stage to the last stage while maintaining their execution order with no loss or duplication of an instruction. To check the above, we verify whether the flow logic obeys a set of rules which expresses how the control connections (`en`, `rst`) of registers in adjacent pipeline stages should be set. In particular, we use a strengthened variant of the rules proposed in [20]. The rules have been strengthened since (as we will see later on) our approach builds on an assumption that, if some pipeline stage is stalled, then all predecessor stages have to be stalled as well. This means that our approach rules out some extreme ways of pipeline implementation allowed by the original rules. An example of such a situation is an optimization of the execution during stage stalling when an instruction preceded by a series of `NOP` instructions is allowed to proceed to the next stage in order to increase the throughput.

For the following, assume a transition system $(C, \hookrightarrow)$ induced by the PSG being verified. We introduce mappings $st, rst \colon V_p \to 2^C$ defined as

$$st(v_p) := \gamma(\{v_p.\texttt{en} \rightsquigarrow 0, v_p.\texttt{rst} \rightsquigarrow 0\}),$$
$$rst(v_p) := \gamma(v_p.\texttt{rst} \rightsquigarrow 1).$$

Intuitively, for any register $v_p \in V_p$, $st(v_p)$ and $rst(v_p)$ are the sets of configurations in which v_p is stalled or cleared, respectively. The pipeline consistency rules that we check are then the following:

- *Rule 1*: If some pipeline register of a stage $s \in \mathbb{S}$ is stalled, then all pipeline registers of the Stage s have to be stalled, i.e., for all $v_p, v'_p \in V_p$:

  $$\varphi(v_p) = \varphi(v'_p) \Rightarrow st(v_p) \subseteq st(v'_p).$$

  The rule follows the idea that an instruction carried by a pipeline stage cannot be fragmented. The rule also reflects one of the fundamental assumptions about pipelined execution from [20]: namely, at any given time, an instruction is always in a single pipeline stage only. As a corollary, by simply swapping v_p and v'_p, one can derive a stronger statement $\varphi(v_p) = \varphi(v'_p) \Rightarrow st(v_p) = st(v'_p)$.
- *Rule 2*: If some pipeline register in a Stage $s \in \mathbb{S} \setminus \{\max(\mathbb{S})\}$ is stalled, then all pipeline registers of the Stage $s+1$ have to be stalled or cleared, i.e., for all $v_p, v'_p \in V_p$:

  $$\varphi(v_p) = \varphi(v'_p) - 1 \Rightarrow st(v_p) \subseteq st(v'_p) \cup rst(v'_p).$$

  This rule is a rephrased version of Equation (15) from [20] and prevents duplication of an instruction.
- *Rule 3*: If some pipeline register in a Stage $s \in \mathbb{S} \setminus \{1\}$ is stalled, then all pipeline registers of the Stage $s-1$ have to be stalled, i.e., for all $v_p, v'_p \in V_p$:

  $$\varphi(v_p) = \varphi(v'_p) + 1 \Rightarrow st(v_p) \subseteq st(v'_p).$$

  This rule is a rephrased version of Equation 16 from [20] and prevents an instruction to be lost.
- *Rule 4*: If some pipeline register in a Stage $s \in \mathbb{S}$ is cleared, then all pipeline registers of the Stage s have to be cleared, i.e., for all $v_p, v'_p \in V_p$:

  $$\varphi(v_p) = \varphi(v'_p) \Rightarrow rst(v_p) \subseteq rst(v'_p).$$

  Similarly to Rule 1, this rule prevents fragmentation of an instruction and it is a part of the basic assumptions about pipelined execution mentioned in [20].

We check the above rules using an SMT solver [5,21, 3] for the bit-vector logic. To convert the rules into the bit-vector logic, we first define an operator * that maps edges of a PSG to variables of the bit-vector logic (BVL) such that $e_1^\star = e_2^\star \Leftrightarrow s(e_1) = s(e_2)$ for each $e_1, e_2 \in E$. Intuitively, edges with the same source must have the same value. Then, for any $e \in E$, we define a BVL formula $\psi(e)$ that encodes how the value transmitted over e is computed from values stored in registers. The formula $\psi(e)$ is recursively defined as

$$\psi(e) := \begin{cases} e^\star = g(e_1^\star, ..., e_m^\star) \wedge \bigwedge_{i=1}^{m} \psi(e_i) & s(e) = (v, \mathtt{q}) \wedge \\ & v \in V_f, \\ true & \text{otherwise} \end{cases}$$

where g denotes the Boolean function computed by the circuit $v \in V_f$.

Now, the inclusion test $st(v_p) \subseteq st(v_p')$ from Rule 1 can be reduced to checking validity of the following formula:

$$\begin{aligned} \Phi(v_p) := (&\psi(v_p.\mathtt{en}) \wedge \psi(v_p.\mathtt{rst}) \wedge \psi(v_p'.\mathtt{en}) \wedge \\ &\psi(v_p'.\mathtt{rst}) \wedge v_p.\mathtt{en}^\star = 0 \wedge v_p.\mathtt{rst}^\star = 0) \Rightarrow \\ &(v_p'.\mathtt{en}^\star = 0 \wedge v_p'.\mathtt{rst}^\star = 0). \end{aligned}$$

Intuitively, $\Phi(v_p)$ says that if the values of $v_p.\mathtt{en}$, $v_p.\mathtt{rst}$, $v_p'.\mathtt{en}$, and $v_p'.\mathtt{rst}$ are computed according to the given flow logic and v_p is stalled, then v_p' is stalled too. Instead of checking validity of $\Phi(v_p)$, one can check unsatisfiability of the negation of the formula, i.e., $\neg sat(\neg\Phi(v_p))$. Moreover, as $\neg\Phi(v_p) = \psi(v_p.\mathtt{en}) \wedge \psi(v_p.\mathtt{rst}) \wedge \psi(v_p'.\mathtt{en}) \wedge \psi(v_p'.\mathtt{rst}) \wedge v_p.\mathtt{en}^\star = 0 \wedge v_p.\mathtt{rst}^\star = 0 \wedge (v_p'.\mathtt{en}^\star = 1 \vee v_p'.\mathtt{rst}^\star = 1)$, the check $\neg sat(\neg\Phi(v_p))$ can be replaced by the following two simpler checks:[5]

$$\neg sat \begin{pmatrix} \psi(v_p.\mathtt{en}) & \wedge\, v_p.\mathtt{en}^\star & = 0 \wedge \\ \psi(v_p.\mathtt{rst}) & \wedge\, v_p.\mathtt{rst}^\star & = 0 \wedge \\ \psi(v_p'.\mathtt{en}) & \wedge\, v_p'.\mathtt{en}^\star & = 1 \end{pmatrix} \quad (1)$$

$$\neg sat \begin{pmatrix} \psi(v_p.\mathtt{en}) & \wedge\, v_p.\mathtt{en}^\star & = 0 \wedge \\ \psi(v_p.\mathtt{rst}) & \wedge\, v_p.\mathtt{rst}^\star & = 0 \wedge \\ \psi(v_p'.\mathtt{rst}) & \wedge\, v_p'.\mathtt{rst}^\star & = 1 \end{pmatrix} \quad (2)$$

Hence, Rule 1 can be checked by applying the checks from Equations 1 and 2 to all $v_p, v_p' \in V_p$ such that $\varphi(v_p) = \varphi(v_p')$.

Rules 2–4 can be checked in a very similar way as Rule 1.

[5] Note that, in Equation 1, we may remove the $\psi(v_p'.\mathtt{rst})$ conjunct since the constraint $v_p'.\mathtt{rst}^\star = 1$ is not present, and likewise with $\psi(v_p'.\mathtt{en})$ in Equation 2.

6 Static Detection of Potential Pipeline Hazards

According to Definitions 1–4, a pipeline hazard (of any of the discussed kinds) occurs when two instructions access the same architectural register and at least one of the accesses is a write. We will further use the term *spoiler* whenever referring to the writing instruction causing the hazard. The other involved instruction will then be called a *victim* instruction. Finally, we will speak about a *hazard case* when referring to the pair formed by a spoiler and a victim instruction.

In this section, we will first focus on identifying a finite set of hazard cases potentially causing hazards in a given processor. For that, we will use a static hazard analysis examining the PSG and pipeline stage mappings φ, φ^{wr}, φ^{rd} determined by the data-flow analysis from Section 5.1. In order to be able to describe a spoiler-victim pair forming a hazard case, we will introduce several auxiliary notions, among which the so-called *forward execution*, *minimal transfer execution*, and *maximal store execution* are the most important.

We begin by introducing a notion representing a generic concept of a data transfer between two vertices within a given PSG. Naturally, each such transfer must conform to the φ^{wr} and φ^{rd} mappings. We first formalize the notion of data transfers in a broader form in Definition 5, which is narrowed later on in Definition 6. In particular, Definition 5 is broader in the sense that it may describe data transfers that can only be achieved when multiple instructions are involved and some of the instructions pass the data back to lower stages of the pipeline where they are processed by instruction(s) that entered the pipeline later. This would mean that a spoiler itself (and likewise a victim) could consist of multiple instructions. Dealing with such situations is, of course, relevant, but we will restrict ourselves to the case of the spoiler and victim being single instructions each, generating the so-called forward executions (Definition 6). Nevertheless, this does not mean that we ignore multi-instruction hazards completely. Instead, thanks to the concepts introduced in Definitions 7, 8, and 10, our approach is capable of generating alarms in situations where the potentially erroneous value is not made visible but ends up in a pipeline register from which another instruction can make its effect visible. Such alarms may, however, be false, and our approach will not be able to see that.

Definition 5 Let $G = (V, E, s, t)$ be a PSG and π an alternating sequence $\langle v_1, e_1, \ldots, e_{k-1}, v_k \rangle$, $k > 1$, of vertices interconnected by edges, that is, $v_1, \ldots, v_k \in V$, $e_1, \ldots, e_{k-1} \in E$, $s(e_i) = (v_i, \mathtt{c_i})$, and $t(e_i) = (v_{i+1}, \mathtt{c_{i+1}})$ for each $1 \leq i < k$ and $\mathtt{c_1}, \ldots, \mathtt{c_k} \in \mathbb{T}$. Moreover, assume that, in π, no vertex appears twice, i.e., $i \neq j \Rightarrow v_i \neq v_j$ for $1 \leq i, j \leq k$. We say that a pair (π, τ) is an *execution* if there exists a valuation $\tau\colon \{1, \ldots, 2k-1\} \to \mathbb{S}$ s.t. $v_i \in V_r \Rightarrow \tau(j) - 1 \in \varphi^{\text{wr}}(v_i)$ for all $1 < i \leq k$ and

$j = 2i - 1$. We denote π and τ as an *execution path* and an *execution plan*, respectively. We will further use $\mathbb{X}$ to denote the set of all executions. Moreover, we use π^{fst} and π^{lst} to denote the first v_1, resp., the last v_k of the path π. Analogically, we will use shortcuts τ^{fst} and τ^{lst} in order to refer to the valuation of the first $\tau(1)$ and last element $\tau(2k-1)$ of the execution path π, respectively, i.e., $\tau^{\text{fst}} = \tau(1)$ and $\tau^{\text{lst}} = \tau(2k-1)$.

Intuitively, given an execution, its execution plan represents a sequence of stages in which particular vertices are written during a data transfer. Hence, taking into account the unit delay of writing, the value written to a vertex v_i is obtained from a value computed in the stage $\tau(j) - 1$, $j = 2i - 1$ (with the first element of the path being special and excluded from this requirement).

Example 2. Consider the PSG G depicted in Fig. 2. A pair (π_1, τ_1) s.t. $\pi_1 = \langle X, MxMem.\texttt{sel}, MxMem, ExMem.\texttt{d}, ExMem, MxOp.\texttt{c}_0, MxOp, Eq.\texttt{a}_1, Eq, MxAlu.\texttt{c}_0, MxAlu, A.\texttt{d}, A\rangle$ and $\tau_1 = \{1^{X} \mapsto 3, 2^{MxMem.\texttt{sel}} \mapsto 3, 3^{MxMem} \mapsto 3, 4^{ExMem.\texttt{d}} \mapsto 3, 5^{ExMem} \mapsto 4, 6^{MxOp.\texttt{c}_0} \mapsto 4, 7^{MxOp} \mapsto 4, 8^{Eq.\texttt{a}_1} \mapsto 4, 9^{Eq} \mapsto 4, 10^{MxAlu.\texttt{c}_0} \mapsto 4, 11^{MxAlu} \mapsto 4, 12^{A.\texttt{d}} \mapsto 4, 13^{A} \mapsto 5\}$ is an execution in G describing one of the possible data transfers from the register X to the register A. Note that we indexed the left-hand sides of the mappings by the corresponding registers to make the mappings more readable and to allow the reader to check that the execution indeed obeys the rules from Definition 5 when considering the φ^{wr} mapping from Table 1.

Another example of an execution is a pair (π_2, τ_2) where $\pi_2 = \langle ExJmp, MxJmp.\texttt{sel}, MxJmp, MxPC.\texttt{sel}, MxPC, PC.\texttt{d}, PC, MxProg.\texttt{sel}, MxProg, IdIr.\texttt{d}, IdIr\rangle$ and $\tau_2 = \{1^{ExJmp} \mapsto 4, 2^{MxJmp.\texttt{sel}} \mapsto 4, 3^{MxJmp} \mapsto 4, 4^{MxPC.\texttt{sel}} \mapsto 4, 5^{MxPC} \mapsto 4, 6^{PC.\texttt{d}} \mapsto 4, 7^{PC} \mapsto 5, 8^{MxProg.\texttt{sel}} \mapsto 1, 9^{MxProg} \mapsto 1, 10^{IdIr.\texttt{d}} \mapsto 1, 11^{IdIr} \mapsto 2\}$. ◁

To narrow our selection only to executions that are feasible by a single instruction, one needs to only think of executions tied with execution plans where stages form a non-decreasing sequence. Intuitively, a single instruction in the pipeline can only move forward or stay in the same stage. This leads us to the definition given next.

Definition 6 A *forward execution* is a special type of execution $(\langle v_1, e_1, \ldots, v_k\rangle, \tau) \in \mathbb{X}$, $k > 1$, where the following restrictions hold for all $1 < i \leq k$ and all $1 \leq j < k$: (i) $v_i \in V_r \Rightarrow \tau(2i-1) = \tau(2i-2)+1$, (ii) $v_i \in V_f \Rightarrow \tau(2i-1) = \tau(2i-2)$, and (iii) $e_j \in E \Rightarrow \tau(2j) = \tau(2j-1)$.

As can be seen, the constraints of Definition 6 require that the stage associated with a register should increase (Condition (i)) and that the stage associated with a Boolean circuit or an edge should remain the same (Conditions (ii) and (iii)). Clearly, if any of the conditions is not met, there could not be any single instruction capable of a data transfer described by the execution.

Example 3. Consider the executions from Example 2. The execution (π_1, τ_1) is a forward execution while (π_2, τ_2) is not since $\tau_2(8^{MxProg.\texttt{sel}}) \neq \tau_2(7^{PC})$. ◁

For further explanation, it is important to be able to identify a register from which the transferred data can be passed to another (later) instruction. Such an action occurs only if there exists a path leading from a register in a higher stage to a register that belongs to a lower one. This is formalized in the next definition.

Definition 7 A pipeline register $v \in V_p$ is a *pivot* if there exist a stage $s_r \in \varphi^{\text{rd}}(v)$ s.t. $s_r \leq \varphi(v)$.

We also need to establish a notion of a stage that can be cleared without the previous stage being stalled. Such a stage can be used to nullify the state of a partially executed instruction.

Definition 8 A stage $s \in \mathbb{S}$ is *independently clearable* if there exist pipeline registers $v_p, v'_p \in V_p$ s.t. $\varphi(v_p) = s = \varphi(v'_p)+1$ and $rst(v_p) \cap \overline{st(v'_p)} \neq \emptyset$ where st and rst are the mappings defined in Section 5.2.

We decide whether a stage satisfies the above given constrains for being independently clearable in a similar way to Rules 1–4. More precisely, an SMT solver performs the following check in this case:

$$sat\left(\begin{array}{lll} \psi(v_p.\texttt{rst}) & \wedge\, \psi(v'_p.\texttt{en}) & \wedge\, \psi(v'_p.\texttt{rst}) \\ v_p.\texttt{rst}^\star = 1 & \wedge\, (v'_p.\texttt{en}^\star = 1 & \vee\, v'_p.\texttt{rst}^\star = 1) \end{array}\right) \quad (3)$$

The above check can be further decomposed into two simpler checks while it suffices that at least one is satisfiable:

$$sat\left(\begin{array}{ll} \psi(v_p.\texttt{rst}) & \wedge\, v_p.\texttt{rst}^\star = 1 \\ \psi(v'_p.\texttt{en}) & \wedge\, v'_p.\texttt{en}^\star = 1 \end{array}\right) \quad (4)$$

$$sat\left(\begin{array}{ll} \psi(v_p.\texttt{rst}) & \wedge\, v_p.\texttt{rst}^\star = 1 \\ \psi(v'_p.\texttt{rst}) & \wedge\, v'_p.\texttt{rst}^\star = 1 \end{array}\right) \quad (5)$$

In the next step, we define an execution that can be performed by a single instruction and which *may* influence the value stored in some register.

Definition 9 A *store execution* is a forward execution $(\langle v_1, e_1, \ldots, e_{k-1}, v_k\rangle, \tau)$ for some $k > 0$, $v_1, v_k \in V_r$, so that $v_2, \ldots, v_{k-1} \notin V_r$. We also define a *maximal store execution* as a store execution that is not a suffix of any other store execution.

As a final step, we define an execution that can be performed by a single instruction and which *may* influence the data stored in an architectural register $v_a \in V_a$ by reading some data from a (potentially different) register $v \in V_r$ and transferring them to the register v_a.

Definition 10 A *transfer execution* is a forward execution $(\langle v_1, e_1, \ldots, e_{k-1}, v_k\rangle, \tau)$ for some $k > 0$, $v_k \in V_r$ that satisfies the following two properties: (i) The register v_k satisfies one of the following: (a) it is an architectural register $v_k \in V_a$, (b) it is a pipeline register $v_k \in V_p$ s.t. $t(e_{k-1}) = (v_k, \texttt{rst})$ and $\varphi(v_k)$ is an independently clearable stage, or (c) the register $v_k \in V_p$ is a pivot s.t. $t(e_{k-1}) = (v_k, \texttt{d})$. (ii) Moreover, $t(e_i) \notin V_p \times \{\texttt{en}, \texttt{rst}\}$ for all $1 \leq i < k$. We also define a *minimal transfer execution* as a transfer execution that does not contain any prefix that is a transfer execution.

Condition (i-a) is straightforward as the execution affects the architectural register directly in this case. Clearing the target pipeline register $v_k \in V_p$ in an independently clearable stage as described in Condition (i-b) causes cancellation of any partially executed instruction in Stage $\varphi(v_k)$. Such an event may indirectly influence any architectural register $v_a \in V_a$ that belongs to a stage $s \geq \varphi(v_k)$. Similarly, concerning Condition (i-c), if the target pipeline register $v_k \in V_p$ is a pivot, the value read from it—by a later instruction—may also indirectly influence any architectural register that the later instruction writes to. Next, as described by Condition (ii), the transfer execution must not traverse through enable connections of pipeline registers. Such executions cannot influence the value of any architectural register. Their only impact can be that they stall a stage. This also holds for reset connections of pipeline registers in a stage that is not independently clearable—in this case, an instruction cannot be lost since the previous stage is always stalled. In such a case, the pipeline consistency given by Rules 1–4 from Section 5.2 assures correct preservation of all partially executed instructions.

An incorrectly handled pipeline hazard manifests upon the first write of improper data into some architectural register of the design. Therefore, it suffices to further deal with the minimal transfer executions only. We can now formalize the notion of hazard cases in a unified way for all the different kinds of hazards (restricted to the case when the spoiler and victim consist of single instructions) as follows. In particular, we represent a hazard case as a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ where χ_{sp} and χ_{vi} are spoiler and victim executions appropriate for the concerned kind of hazard. A more rigorous description of each considered type of hazard cases is given in the following definitions.

Definition 11 A *RAW hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \ldots, e_{k-1} = v_{k-1}^{sp}.\texttt{d}, v_k^{sp} = v\rangle, \tau_{sp})$ of a spoiler instruction and a minimal transfer execution $\chi_{vi} = (\langle v_1^{vi} = v, e_1^{vi}, \ldots, v_\ell^{vi}\rangle, \tau_{vi})$ of a victim instruction where $v \in V_a \setminus \{v_{pc}\}$, $k, \ell > 1$, and data in the architectural register v can be read by the victim instruction before they are written by the spoiler, i.e., $\tau_{vi}^{\text{fst}} < \tau_{sp}^{\text{lst}}$.

Definition 12 A *WAR hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \ldots, e_{k-1} = v_{k-1}^{sp}.\texttt{d}, v_k^{sp} = v\rangle, \tau_{sp})$ of a spoiler instruction and a minimal transfer execution $\chi_{vi} = (\langle v_1^{vi} = v, e_1^{vi}, \ldots, v_\ell^{vi}\rangle, \tau_{vi})$ of a victim instruction where $v \in V_a \setminus \{v_{pc}\}$, $k, \ell > 1$, and data in the architectural register v can be written by the spoiler before they are read by the victim, i.e., $\tau_{sp}^{\text{lst}} < \tau_{vi}^{\text{fst}}$.

Definition 13 A *WAW hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \ldots, e_{k-1}^{sp} = v.\texttt{d}, v_k^{sp} = v\rangle, \tau_{sp})$ of a spoiler instruction and a maximal store execution $\chi_{vi} = (\langle v_1^{vi}, e_1^{vi}, \ldots, e_{\ell-1}^{vi} = v.\texttt{d}, v_\ell^{vi} = v\rangle, \tau_{vi})$ of a victim instruction where $v \in V_a \setminus \{v_{pc}\}$, $k, \ell > 1$, and data into the architectural register v can be written from two different stages. In the following, without a loss of generality (since the conflicting instructions can always be swapped), we will assume the spoiler to perform a write operation in an earlier stage, i.e., $\tau_{sp}^{\text{lst}} < \tau_{vi}^{\text{lst}}$.

One can observe that there is no need to include any minimal transfer execution in the case of WAW hazard since an error that is caused by the hazard is manifested instantly by writing an incorrect value to the register v.

Definition 14 A *CTL hazard case* is a tuple $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$ consisting of a maximal store execution $\chi_{sp} = (\langle v_1^{sp}, e_1^{sp}, \ldots, e_{k-1} = v_{k-1}^{sp}.\texttt{d}, v_k^{sp} = v_{pc}\rangle, \tau_{sp})$ of a spoiler instruction and a minimal transfer execution $\chi_{vi} = (\langle v_1^{vi} = v_{pc}, e_1^{vi}, \ldots, v_\ell^{vi}\rangle, \tau_{vi})$ of a victim instruction where $k, \ell > 1$, $v_{pc} \neq v_\ell^{vi}$. Moreover, the v_{pc} register must be written with data originating from a source other than the program counter's auto-increment logic, which we consider to appear in Stage 1. Therefore, the spoiler must always write from a stage other than the first one, i.e., $\tau_{sp}^{\text{lst}} > 2$.

Note that, since the definition of a particular hazard case speaks about registers, their access stages, and the path along which the problematic data are transferred, it is not defined for a single concrete instruction only but for an entire class of instructions that conform to the criteria given by the hazard case. Further, note that the cases when $\tau_{sp}^{\text{lst}} = \tau_{vi}^{\text{fst}}$ for RAW, WAR, and CTL hazards as well as the cases when $\tau_{sp}^{\text{lst}} = \tau_{vi}^{\text{lst}}$ for WAW hazards are not covered by the above definitions. This is because our approach assumes correct execution of isolated instructions, which rules such cases out. Such correctness can be checked separately using, e.g., methods described in [7,9]. Finally, albeit CTL hazards are quite similar to RAW hazards (as CTL hazards could be considered a special kind of RAW hazards on the v_{pc} register), we prefer to keep them separate due to CTL hazards do still come with some special constraints (such as the treatment of the program counter's auto-increment logic) and due to the different nature of errors that may typically arise from them.

In order to generate the set $\mathbb{H}$ of hazard cases, we proceed as follows. First, using results of the data-flow analysis from Section 5.1, we find all registers $v_a \in V_a$ for which there is a risk that some hazard situation may be initiated between stages $s_1, s_2 \in \mathbb{S}$. The conditions that must hold for s_1, s_2 differ for different hazard cases. For instance, for RAW hazards, we need the following conditions to hold: $s_1 - 1 \in \varphi^{\text{wr}}(v_a)$, $s_2 + 1 \in \varphi^{\text{rd}}(v_a)$, and $s_2 < s_1$. The condition $s_2 < s_1$ reflects the fact that the needed data are read from v_a before they are written into v_a. The rest of the condition reflects that it must be possible to write to v_a in stage s_1 and read in stage s_2, i.e., it must have a predecessor register in stage s_1-1 and a successor register in stage s_2+1. The subtraction/addition of 1 is applied due to the unit write delay that happens between the data are read from the previous register and written to v_a and then between reading the data from v_a and writing them to the successor register. For other kinds of hazards, the conditions are derived from the kind of hazard analogously as for RAW hazards as shown later on. Second, we find all maximal store executions that terminate in the register v_a. Finally, we generate all minimal transfer executions originating from the v_a vertex of the given PSG G.[6]

The procedure for generating the set $\mathbb{H}$ is shown in Alg. 1. The procedure first constructs auxiliary sets A_{RAW}, A_{WAR}, and A_{CTL} strictly following the constraints given by RAW, WAR, and CTL hazard cases (see Definitions 11, 12, and 14). The sets A_{RAW}, A_{WAR}, and A_{CTL} consist of quintuples characterising suspected hazards. They include the architectural register v_a on which the hazard happens, the target register v_t through which the hazard manifests, and three stages: namely, stages s_1 and s_2 in which the conflicting read/write operations on v_a happen, and stage s_3 in which the hazard gets manifested. For WAW hazards, the procedure later on proceeds similarly, but there is no v_t and s_3 needed since the hazard manifests immediately upon the second write operation (Definition 13). The auxiliary sets are then used for finding maximal store and minimal transfer executions in the PSG. A standard breadth-first search algorithm during which constraints from Definitions 5–10 are checked on-the-fly can be used to obtain the minimal transfer executions in G for the suspected hazards. Similarly, the procedure may deploy the depth-first search algorithm while checking constraints from Definitions 5, 6, and 9 in order to find the maximal store executions.

Example 4. Consider the PSG from Fig. 2 and the mappings shown in Table 1. One can see that there is a potential WAR hazard on the index register $X \in V_a$ because, for example, it can be written in Stage 3 ($\varphi^{\text{wr}}(X) = \{2, 3, 4\}$) and read by Stage 5 ($\varphi^{\text{rd}}(X) = \{3, 4, 5\}$). By Definition 12, to form a WAR hazard, the PSG must contain (i) a maximal store execution of a spoiler instruction $(\pi_{sp}, \tau_{sp}) \in \mathbb{X}$ ending in X and (ii) a minimal transfer execution $(\pi_{vi}, \tau_{vi}) \in \mathbb{X}$ leading from X to some target register. There are multiple executions of spoiler and victim instructions that satisfy the above criteria. Each of them must be considered in order to verify that the design is free of WAR hazards. For instance, one may consider a spoiler execution (π_{sp}, τ_{sp}) with $\pi_{sp} = \langle X,\ Inc.\mathtt{a}_0,\ Inc,\ MxInc.\mathtt{c}_0,\ MxInc,\ X.\mathtt{d},\ X\rangle$ and $\tau_{sp} = \{1^X \mapsto 2,\ 2^{Inc.\mathtt{a}_0} \mapsto 2,\ 3^{Inc} \mapsto 2,\ 4^{MxInc.\mathtt{c}_0} \mapsto 2,\ 4^{MxInc} \mapsto 2,\ 5^{X.\mathtt{d}} \mapsto 2,\ 6^X \mapsto 3\}$. Further, we can consider a victim execution (π_{vi}, τ_{vi}) with the target memory cell Mem_j written in Stage 5 where $\pi_{vi} = \langle X,\ Cmp_j.\mathtt{a}_0,\ Cmp_j,\ MxSel_j.\mathtt{c}_1,\ MxSel_j,\ Mem_j.\mathtt{en},\ Mem_j\rangle$. An instance of an execution plan τ_{vi} for the path π_{vi} is $\{1^X \mapsto 4,\ 2^{Cmp_j.\mathtt{a}_0} \mapsto 4,\ 3^{Cmp_j} \mapsto 4,\ 4^{MxSel_j.\mathtt{c}_1} \mapsto 4,\ 5^{MxSel_j} \mapsto 4,\ 6^{Mem_j.\mathtt{en}} \mapsto 4,\ 7^{Mem_j} \mapsto 5\}$. The given pair of a spoiler and victim

[6] For WAW hazards, in the final step, we generate maximal store executions instead. In this case, the error caused by the hazard is immediately visible from the programmer's point of view, and there is no need of its propagation to another architectural register.

Algorithm 1 Procedure computing a set of hazard cases $\mathbb{H}$.

Require: A PSG $G = (V, E, s, t, \omega)$, a set $V_a \subseteq V$ of architectural registers, a program counter $v_{pc} \in V_a$, a set $V_p \subseteq V$ of pipeline registers, $V_a \cap V_p = \emptyset$, a set $V_{pivot} \subseteq V_p$ of pivots, and a set $S_{ic} \subseteq \mathbb{S}$ of independently clearable stages.
Ensure: A set $\mathbb{H} \subseteq \mathbb{X} \times \mathbb{X}$ of hazard cases in the CPU encoded by G.
1: $V_t := V_a \cup V_{pivot} \cup \{v \in V_p \mid \varphi(v) \in S_{ic}\}$
2: Let $\mathbb{A}$ denote $V_a \times \mathbb{N} \times \mathbb{N} \times V_t \times \mathbb{N}$
3: $A_{RAW} := \{(v_a, s_1, s_2, v_t, s_3) \in \mathbb{A} \mid s_1 - 1 \in \varphi^{\text{wr}}(v_a) \wedge s_2 + 1 \in \varphi^{\text{rd}}(v_a) \wedge \mathbf{s_2} < \mathbf{s_1} \wedge s_3 - 1 \in \varphi^{\text{wr}}(v_t) \wedge s_2 \leq s_3\}$
4: $A_{WAR} := \{(v_a, s_1, s_2, v_t, s_3) \in \mathbb{A} \mid s_1 - 1 \in \varphi^{\text{wr}}(v_a) \wedge s_2 + 1 \in \varphi^{\text{rd}}(v_a) \wedge \mathbf{s_1} < \mathbf{s_2} \wedge s_3 - 1 \in \varphi^{\text{wr}}(v_t) \wedge s_2 \leq s_3\}$
5: $A_{CTL} := \{(v_{pc}, s_1, 1, v_t, s_3) \in \mathbb{A} \mid s_1 - 1 \in \varphi^{\text{wr}}(v_a) \wedge 2 \in \varphi^{\text{rd}}(v_{pc}) \wedge \mathbf{s_1} > \mathbf{2} \wedge s_3 - 1 \in \varphi^{\text{wr}}(v_t) \wedge v_{pc} \neq v_t \wedge s_3 > 1\}$
6: $A := A_{RAW} \cup A_{WAR} \cup A_{CTL}$
7: $\mathbb{H} := \emptyset$
8: **for** $(v_a, s_1, s_2, v_t, s_3) \in A$ **do**
9: $\quad \chi_{sp} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k\rangle \wedge (\{v_k\} \times \mathbb{N} \times \mathbb{N} \times V_t \times \mathbb{N}) \cap A \neq \emptyset \wedge t(e_{k-1}) = (v_k, \mathtt{d}) \wedge v_2, \ldots, v_{k-1} \notin (V_a \cup V_p) \wedge \tau^{\text{lst}} = s_1 \wedge (\pi, \tau)$ is a maximal store execution $\}$
10: $\quad \chi_{vi} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k\rangle \wedge (\{v_1\} \times \mathbb{N} \times \mathbb{N} \times \{v_k\} \times \mathbb{N}) \cap A \neq \emptyset \wedge \tau^{\text{fst}} = s_2 \wedge \tau^{\text{lst}} = s_3 \wedge (\pi, \tau)$ is a minimal transfer execution $\}$
11: $\quad \mathbb{H} := \mathbb{H} \cup (\chi_{sp} \times \chi_{vi})$
12: **end for**
13: Let $\mathbb{A}'$ denote $V_a \times \mathbb{N} \times \mathbb{N}$
14: $A_{WAW} := \{(v_a, s_1, s_2) \in \mathbb{A}' \mid s_1 - 1, s_2 - 1 \in \varphi^{\text{wr}}(v_a) \wedge \mathbf{s_1} < \mathbf{s_2}\}$
15: **for** $(v_a, s_1, s_2) \in A_{WAW}$ **do**
16: $\quad \chi_{sp} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k\rangle \wedge (\{v_k\} \times \mathbb{N} \times \mathbb{N}) \cap A_{WAW} \neq \emptyset \wedge t(e_{k-1}) = (v_k, \mathtt{d}) \wedge v_2, \ldots, v_{k-1} \notin (V_a \cup V_p) \wedge \tau^{\text{lst}} = s_1 \wedge (\pi, \tau)$ is a maximal store execution $\}$
17: $\quad \chi_{vi} := \{\ (\pi, \tau) \in \mathbb{X} \mid \pi = \langle v_1, e_1, \ldots, e_{k-1}, v_k\rangle \wedge (\{v_k\} \times \mathbb{N} \times \mathbb{N}) \cap A_{WAW} \neq \emptyset \wedge t(e_{k-1}) = (v_k, \mathtt{d}) \wedge v_2, \ldots, v_{k-1} \notin (V_a \cup V_p) \wedge \tau^{\text{lst}} = s_2 \wedge (\pi, \tau)$ is a maximal store execution $\}$
18: $\quad \mathbb{H} := \mathbb{H} \cup (\chi_{sp} \times \chi_{vi})$
19: **end for**
20: **return** $\mathbb{H}$

is clearly a candidate for a WAR hazard since the needed data are overwritten before they are read (unless some control logic over the involved executions prevents the hazard, which will be the subject of further checking). ◁

Example 5. Further, as an example of a control hazard, one can consider a spoiler execution (π_{sp}, τ_{sp}) with $\pi_{sp} = \langle ExAlu,\ MxAlu.\texttt{sel},\ MxAlu,\ MxPC.\texttt{c}_1,\ MxPC,\ PC.\texttt{d},\ PC\rangle$ and $\tau_{sp} = \{1^{ExAlu} \mapsto 4,\ 2^{MxAlu.\texttt{sel}} \mapsto 4,\ 3^{MxAlu} \mapsto 4,\ 4^{MxPC.\texttt{c}_1} \mapsto 4,\ 5^{MxPC} \mapsto 4,\ 6^{PC.\texttt{d}} \mapsto 4,\ 7^{PC} \mapsto 5\}$. As an instance of a victim execution (π_{vi}, τ_{vi}), we can consider an execution path $\pi_{vi} = \langle PC,\ MxProg.\texttt{sel},\ MxProg,\ IdIr.\texttt{d},\ IdIr\rangle$ with an execution plan $\tau_{vi} = \{1^{PC} \mapsto 1,\ 2^{MxProg.\texttt{sel}} \mapsto 1,\ 3^{MxProg} \mapsto 1,\ 4^{IdIr.\texttt{d}} \mapsto 1,\ 5^{IdIr} \mapsto 2\}$. Note that, in this case, $IdIr \notin V_a$, but we know from Table 1 that the pipeline register $IdIr$ is a pivot, and so it is still a valid terminating element for a transfer execution. ◁

7 Parametric Systems for Potential Hazards

We will now describe how the potentially hazardous behaviour of a spoiler and a victim instruction described by a hazard case can be modelled and checked for feasibility using a parametric system P. Namely, the parametric system P will be constructed such that if the behaviour is found infeasible by analysing P, the hazard case does not describe a real hazard (the suspected hazard gets prevented by the pipeline flow logic). Intuitively, in the system P, we map $n \geq 2$ instructions in the pipeline to n concurrently running processes in a linear array (with the earliest instruction on the left).

Note that the notion of parametric systems does not limit the value of n from above, and verification methods designed for them must cope with that. Exploiting this feature of parametric verification is not necessary in our case since the relevant values of n are limited by the length of the pipeline. However, the use of parametric systems is still beneficial since while there is a single spoiler and victim and the number of "padding" instructions in between of them is bounded, we still do not know how many "padding" instructions should be considered for the hazard to manifest. Hence, instead of exploring all possible values of n, we use parametric verification to perform the verification for any value of n, building on that the efficiency of the parametric verification approach we use is quite sufficient for us. Moreover, should our approach be extended to handle in a more precise way even multi-instruction hazards, the value of n could become unbounded, and the power of parametric verification could become truly needed.

In the parametric systems we construct, all instructions are initially in a state saying that their execution has not started. Then, they proceed through individual stages of the pipeline during which they may interact with each other by means of the pipeline flow logic, e.g., an earlier instruction may force a later instruction to be stalled or cleared. Finally, the instructions end up in a state denoting that they left the pipeline.

In the following explanation, we start by constructing the set of states of the system P. Then, we proceed to capturing the above mentioned influence of the pipeline flow logic and reflect it in the transition relation of the system P. Finally, we define the set of minimal bad configurations of the system P that describes the prohibited interleavings of instructions causing the hazard.

7.1 States and Edge Conditions of the Parametric System

Given a hazard case of the form $(\chi_{sp}, \chi_{vi}) \in \mathbb{X}^2$, $\chi_{sp} = (\pi_{sp}, \tau_{sp})$, $\chi_{vi} = (\pi_{vi}, \tau_{vi})$, the parametric system P will model interactions among four classes of processes $\mathbb{K} := \{sp$ "spoiler", vi "victim", sf "stall-flow", nf "normal-flow"$\}$ (where each process represents an executing instruction of some class). This follows the fact that each type of the considered pipeline hazard is caused by some pair of instructions. The sp class represents the spoiler part of the hazard case, i.e., an instruction that writes to a register $v \in V_a$ in a stage $\tau_{sp}(v)$. The vi class then represents an instruction corresponding to the victim part of the hazard case, reading or writing from/to v in a stage $\tau_{vi}(v)$. Further, the sf and nf classes both denote any other instructions than the spoiler and victim—we just differentiate two operating modes of these instructions. As we will discuss later in Section 7.2, the difference between the stall- and normal-flow operation modes is that an sf-class instruction in a stage $s_0 \in \mathbb{S}$ causes that all pipeline stages $s \in \mathbb{S}$ s.t. $s < s_0$ get stalled. Both the sf and nf classes serve as a pipeline filler and a sink for cleared (flushed) instructions.

To facilitate the construction of a parametric system allowing us to verify whether a given hazard case corresponds to a real hazard or not, we need to introduce an *extended* set of stages. Let $\bar{\mathbb{S}} := \mathbb{S} \cup \{\bot, \top\}$ be the set of stages extended with auxiliary initial "$\bot$" and final "$\top$" stages. Intuitively, the initial stage "$\bot$" will represent instructions that have not entered the pipeline yet. Likewise, the final stage "$\top$" will denote finished instructions that have already left the pipeline.

We will then represent the behaviour of instructions given by a hazard case $h = (\chi_{sp}, \chi_{vi})$ in the form of a labelled parametric system, called a *hazard system* (HS), $P^h = (Q^h, \Delta^h, \alpha^h)$ where $Q^h := \mathbb{K} \times \bar{\mathbb{S}}$, Δ^h will be introduced in Section 7.2, and $\alpha^h \colon Q^h \to 2^{\mathbb{E}}$ is a state labelling function. The labelling function α^h associates each state with a set of edge conditions that should hold in this state for the hazard to be executable. We will show the construction of the labelling below. Note that each state $q \in Q^h$ represents a unique instruction class and a stage in which an instruction of this

class is supposed to be. Finally, for a proper understanding of the rest of the section, we once again stress that particular states in Q^h are states of *individual instructions*, not of the entire system. A configuration of the system P^h is a sequence of such states.

Next, we define the mapping α^h describing which edge conditions must hold in a state $q = \langle \kappa, s \rangle \in Q^h$, which is a state of an instruction of the class $\kappa \in \mathbb{K}$ in the stage $s \in \bar{\mathbb{S}}$, for that instruction to execute in accordance with the hazard case h. First, for instructions of the classes $\kappa = sf$ and $\kappa = nf$, we define $\alpha^h(\langle \kappa, s \rangle) := \emptyset$ for every $s \in \bar{\mathbb{S}}$ since we do not expect any special behaviour from instructions of these classes, and, on every realistic processor, we can always find instructions that do not interfere with the spoiler and victim instructions and may serve as the needed pipeline filler. Likewise, we define $\alpha^h(\langle \kappa, s \rangle) := \emptyset$ for any $\kappa \in \mathbb{K}$ and $s \in \{\bot, \top\}$, i.e., for instructions that have not yet started or that have already ended.

For the spoiler and victim instructions, the idea is to extract the edge conditions by looking for the necessary settings of selector, enable, and clear edges so that the data involved in the potential hazard are carried over the paths π_κ for $\kappa \in \{sp, vi\}$ that are a part of the concerned spoiler and victim executions $\chi_\kappa = (\pi_\kappa, \tau_\kappa)$. The mapping α^h can be constructed from three auxiliary mappings $\alpha^h_{\texttt{sel}}$, $\alpha^h_{\texttt{en}}$, and $\alpha^h_{\texttt{rst}} \colon \mathbb{X} \to 2^{\mathbb{E} \times \mathbb{S}}$ where $\alpha^h_{\texttt{sel}}$ will be examining all edges but the last one (hence covering all edges that route the data through multiplexers) and the last edge will be covered by exactly one of the two remaining mappings (related to enabling a write of the data to the target register or clearing the register).

In order to define $\alpha^h_{\texttt{sel}}$, we will use an auxiliary mapping $\sigma \colon V_{mx} \times E \to \mathbb{E}$. Given a multiplexer $v_{mx} \in V_{mx}$, the mapping σ captures the edge condition that must hold over the multiplexer's selector edge $v_{mx}.\texttt{sel}$ for the data on the i-th inbound-case edge $v_{mx}.\texttt{c}_i$ to be propagated to the multiplexer's outbound edge $v_{mx}.\texttt{q}$. Hence,

$$\sigma(v_{mx}, v_{mx}.\texttt{c}_i) := v_{mx}.\texttt{sel} \rightsquigarrow \mathit{bin}_{\omega(v_{mx}.\texttt{sel})}(i)$$

where $\mathit{bin}_n \colon \mathbb{Z} \to \mathbb{B}^n$ is the standard two's complement encoding of a decimal value on n bits.

Now, the $\alpha^h_{\texttt{sel}}$ mapping is defined as

$$\begin{aligned}\alpha^h_{\texttt{sel}}(\chi) := \{(\sigma(v_i, e_{i-1}), \tau(2i-1)) \mid 1 < i < k \wedge \\ v_i \in V_{mx} \wedge \chi = (\langle v_1, e_1, \ldots, e_{i-1}, v_i, \ldots, v_k \rangle, \tau)\}.\end{aligned} \tag{6}$$

Intuitively, the $\alpha^h_{\texttt{sel}}$ mapping produces a set of pairs consisting of a condition $\sigma(v_i, e_{i-1}) \in \mathbb{E}$ over selector edges that is required by the multiplexer $v_i \in V_{mx}$ to propagate the data along the execution path π and the stage $\tau(v_i)$ in which the particular condition must be satisfied. Similarly, the $\alpha^h_{\texttt{en}}$ and $\alpha^h_{\texttt{rst}}$ mappings establish the necessary condition for the final edge of the execution's target register, making sure that either writing of the data into the register is enabled or the register is cleared:

$$\begin{aligned}\alpha^h_{\texttt{en}}(\chi) := \{(v_k.\texttt{en} \rightsquigarrow 1, \tau(2k-1)) \mid \\ \chi = (\langle v_1, e_1, \ldots, e_{k-1} = v_k.\texttt{d}, v_k \rangle, \tau)\},\end{aligned} \tag{7}$$

$$\begin{aligned}\alpha^h_{\texttt{rst}}(\chi) := \{(v_k.\texttt{rst} \rightsquigarrow 1, \tau(2k-1) \mid \\ \chi = (\langle v_1, e_1, \ldots, e_{k-1} = v_k.\texttt{rst}, v_k \rangle, \tau)\}.\end{aligned} \tag{8}$$

In particular, $\alpha^h_{\texttt{en}}$ ensures that the data transferred along the path described by the execution χ are indeed written to its destination register v_k at the end of the execution. Therefore, $\alpha^h_{\texttt{en}}$ produces a singleton containing a pair consisting of the condition $v_k.\texttt{en} \rightsquigarrow 1$ and the stage $\tau(2k-1)$, which is the stage where the data reside just prior to the write. Similarly, $\alpha^h_{\texttt{rst}}$ produces a singleton containing a pair consisting from the condition $v_k.\texttt{rst} \rightsquigarrow 1$ and the stage $\tau(2k-1)$ so that the target register is indeed cleared. Using the above mappings, we can define α^h for the given hazard case $h = (\chi_{sp}, \chi_{vi})$ such that the following holds for any state $\langle \kappa, s \rangle \in \{sp, vi\} \times \bar{\mathbb{S}}$:[7]

$$\begin{aligned}\alpha^h(\langle \kappa, s \rangle) := \{c \in \mathbb{E} \mid (c, s) \in \alpha^h_{\texttt{sel}}(\chi_\kappa) \cup \alpha^h_{\texttt{en}}(\chi_\kappa) \cup \\ \alpha^h_{\texttt{rst}}(\chi_\kappa)\}.\end{aligned} \tag{9}$$

Example 6. Assume the hazard case (χ_{sp}, χ_{vi}) shown in Example 4 for the microprocessor from Example 1. First, we focus on the spoiler execution $\chi_{sp} = (\pi_{sp}, \tau_{sp})$. Since the microprocessor contains five pipeline stages, the spoiler gets associated with the set of states $Q^h_{sp} := \{sp\} \times \bar{\mathbb{S}}$ where $\bar{\mathbb{S}} = \{\bot, 1, \ldots, 5, \top\}$. We will now show how the α^h mapping is computed for the states of Q^h_{sp}. From the definition of α^h, it directly follows that

$$\alpha^h(\langle sp, \bot \rangle) = \alpha^h(\langle sp, \top \rangle) = \emptyset.$$

For the states $\langle sp, 1 \rangle$, ..., $\langle sp, 5 \rangle$, one has to first compute the auxiliary mappings $\alpha^h_{\texttt{sel}}$, $\alpha^h_{\texttt{en}}$, and $\alpha^h_{\texttt{rst}}$ from Equation 9. As the X register is written via its d connection, it immediately follows that

$$\alpha^h_{\texttt{rst}}(\chi_{sp}) = \emptyset.$$

Next, since the path π_{sp} of the spoiler store execution χ_{sp} passes through a single multiplexer, namely, $MxInc$, via the edge $MxInc.\texttt{c}_0$ with $\tau_{sp}(4^{MxInc.\texttt{c}_0}) = 2$, we get

$$\alpha^h_{\texttt{sel}}(\chi_{sp}) = \{(MxInc.\texttt{sel} \rightsquigarrow 0, 2)\}.$$

For $\alpha^h_{\texttt{en}}$, we only need to assure that the register X is written at the end of the execution. Since $\tau_{sp}(5^{X.\texttt{d}}) = 2$, we let

$$\alpha^h_{\texttt{en}}(\chi_{sp}) = \{(X.\texttt{en} \rightsquigarrow 1, 2)\}.$$

[7] Note that the executions can also end by an $v_k.\texttt{en}$ edge. However, in this case, no matter what the value of the enable signal is a hazard happens by enabling/not enabling a write of some data into an architectural register. Hence, no further condition is needed in this case.

Finally, by uniting the above computed auxiliary mappings, we get that

$$\alpha^h(\langle sp, 2\rangle) = \{MxInc.\texttt{sel} \rightsquigarrow 0, X.\texttt{en} \rightsquigarrow 1\}$$

and $\forall i \in \mathbb{S} \setminus \{2\}\colon \alpha^h(\langle sp, i\rangle) = \emptyset$. Analogically, for the victim execution $\chi_{vi} = (\pi_{vi}, \tau_{vi})$ of the analyzed hazard case, we would infer that

$$\alpha^h_{\texttt{sel}}(\chi_{vi}) = \{(MxSel_j.\texttt{sel} \rightsquigarrow 1, 4)\}$$

and $\alpha^h_{\texttt{rst}}(\chi_{vi}) = \alpha^h_{\texttt{en}}(\chi_{vi}) = \emptyset$. Therefore, we get that

$$\alpha^h(\langle vi, 4\rangle) = \{MxSel_j.\texttt{sel} \rightsquigarrow 1\}$$

and $\forall i \in \mathbb{S} \setminus \{4\}\colon \alpha^h(\langle vi, i\rangle) = \emptyset$. ◁

7.2 The Transition Relation of the Parametric System

For the construction of the transition relation Δ^h presented later on, we will first introduce three predicates that characterise mutual interactions of pairs of instructions whose execution has reached some states $q_1, q_2 \in Q^h$ of the verified HS P^h. We stress that q_1 and q_2 are states of the execution of two considered instructions, which are of course a part of a single configuration of the HS P^h. Before providing rigorous definitions of the predicates, which are given later in this section, we first provide some intuition behind them.

A pair of states $q_1, q_2 \in Q^h$ and a stage $s \in \mathbb{S}$ satisfy the ternary *stage stall* predicate $\overset{st}{\underset{h}{\longleftrightarrow}} \subseteq Q^h \times \mathbb{S} \times Q^h$ provided that the edge conditions associated with the states q_1 and q_2 ensure that the stage s is stalled, and thus the contents of all pipeline registers of s stays unchanged. We will further use the shorthand $q_1 \overset{st}{\underset{h,s}{\longleftrightarrow}} q_2$ for $(q_1, s, q_2) \in \overset{st}{\underset{h}{\longleftrightarrow}}$.

Further, a pair of states $q_1, q_2 \in Q^h$ and a stage $s \in \mathbb{S}$ satisfy the ternary *stage clear* predicate $\overset{cl}{\underset{h}{\longleftrightarrow}} \subseteq Q^h \times \mathbb{S} \times Q^h$ provided that the stage s is cleared, i.e., the contents of all pipeline registers of s is nullified. We will further use the shorthand $q_1 \overset{cl}{\underset{h,s}{\longleftrightarrow}} q_2$ for $(q_1, s, q_2) \in \overset{cl}{\underset{h}{\longleftrightarrow}}$.

Finally, a pair of states $q_1, q_2 \in Q^h$ satisfies a binary *state conflict* predicate $\overset{cf}{\underset{h}{\longleftrightarrow}} \subseteq Q^h \times Q^h$ provided that the given processor excludes a configuration where two instructions would appear in the states q_1, q_2 at the same time. We will further use the shorthand $q_1 \overset{cf}{\underset{h}{\longleftrightarrow}} q_2$ for $(q_1, q_2) \in \overset{cf}{\underset{h}{\longleftrightarrow}}$. For instance, one of the typical scenarios when two states $q_1, q_2 \in Q^h$ are in a state conflict occurs when there exists an edge $e \in E$ so that $e \rightsquigarrow b_1 \in \alpha^h(q_1) \wedge e \rightsquigarrow b_2 \in \alpha^h(q_2)$, $b_1, b_2 \in \mathbb{B}$, while $b_1 \neq b_2$.

In order to formally define the above described predicates, we first introduce two auxiliary notions: in particular, (i) a mapping $unwind_h\colon Q^h \to 2^C$ where C is the set of configurations of the TS $T^h = (C, \hookrightarrow)$ induced by the PSG and (ii) a predicate $csat_h \subseteq 2^{\mathbb{E}} \times 2^{Q^h}$.

The purpose of the $unwind_h$ mapping is to compute all configurations of the TS T^h in which T^h (and hence the processor it represents) can be when the processor contains an instruction of a class κ in a stage s while executing within the given hazard case h. The considered configurations must be such that the processor can reach them by going through all preceding stages and such that the processor can finish the execution of the instruction by going through all its further stages, all the time executing within the hazard case h. In particular, let $m = \max(\mathbb{S})$ be the number of stages and let $\langle \kappa, s\rangle \in Q^h$ be an instruction state representing an instruction of a class κ in a stage s within a hazard case h. Then, $unwind_h(\langle \kappa, s\rangle)$ consists of exactly all those configurations $c_0 \in C$ such that there is a sequence of consecutive states $\langle c_{-s+1}, \ldots, c_0, \ldots, c_{m-s}\rangle$ in T^h that conforms to the following rules:

$$\forall -s < i < m - s\colon c_i \hookrightarrow c_{i+1}, \tag{10}$$

$$\forall -s < i \leq m - s\colon c_i \in \gamma(\alpha^h(\langle \kappa, s + i\rangle)). \tag{11}$$

Note that the negative/positive index in the above equations simply means moving forward to/away from the pipeline start when considering the origin given by the index of 0, respectively. The first constraint above ensures that we indeed consider a trace in the TS T^h. The second condition then ensures that the trace passes all stages of an instruction of the given class while the processor is executing within the given hazard case.

The above described computation of the $unwind_h$ mapping can be implemented symbolically using a BVL formula $unwind^\star_h(q)$ for any $q \in Q^h$. To describe the computation, we introduce the notation $\hookrightarrow^\star_{(i,i+1)}$ to denote the result of a (straightforward) conversion of the relation $\hookrightarrow$ to a BVL formula where all variables representing the current state of the TS T^h are indexed with i and those representing the future state are indexed with $i + 1$. Moreover, as in Section 5.2, we use $e^\star_i$ to denote the conversion of an edge $e \in E$ indexed with the trace index i to a BVL variable. Then, given $q = \langle \kappa, s\rangle \in Q^h \setminus \mathbb{K} \times \{\bot, \top\}$, the BVL formula $unwind^\star_h(q)$ is obtained as follows:

$$\begin{aligned}
F_1 &:= \bigwedge_{i=-s+1}^{m-s-1} \hookrightarrow^\star_{(i,i+1)},\\
F_2(q) &:= \bigwedge_{i=-s+1}^{m-s} \bigwedge_{e \rightsquigarrow b \in \alpha^h(\langle \kappa, s+i\rangle)} e^\star_i = b,\\
F_3 &:= \bigwedge_{e \in E} e^\star = e^\star_0,\\
unwind^\star_h(q) &:= \exists \overline{E} : F_1 \wedge F_2(q) \wedge F_3.
\end{aligned} \tag{12}$$

Above, the existential quantification ranges over the set $\overline{E} = \{e_i^\star \mid e \in E \wedge -s < i \leq m - s\}$. Its reason is to get rid of the concrete past and future values of the variables that appear in the execution, keeping only their impact on the current values of the variables.[8] Finally, in order to extend the definition of $unwind_h$ for initial and final states $q' \in \mathbb{K} \times \{\bot, \top\}$, we define $unwind_h^\star(q') := true$.

Further, we proceed to the second auxiliary predicate: $csat_h$. The $csat_h$ predicate determines satisfiability of a set of edge conditions $I \subseteq \mathbb{E}$ in a situation when the pipeline contains instructions in states from a set $S \subseteq Q^h$. Formally, it is defined as follows:

$$csat_h(I, S) \Leftrightarrow \bigcap_{c \in I} \gamma(c) \cap \bigcap_{q \in S} unwind_h(q) \neq \emptyset. \tag{13}$$

The evaluation of $csat_h(I, S)$ can be naturally reduced to checking the satisfiability of a BVL formula as follows:

$$csat_h(I, S) \Leftrightarrow sat\Big(\bigwedge_{e \rightsquigarrow b \in I} e^\star = b \wedge \bigwedge_{q \in S} unwind_h^\star(q) \Big). \tag{14}$$

Now, the predicate $csat_h$ can be used to precisely define the needed predicates $\overset{st}{\underset{h}{\longleftrightarrow}}$, $\overset{cl}{\underset{h}{\longleftrightarrow}}$, and $\overset{cf}{\underset{h}{\longleftrightarrow}}$ as follows.

Definition 15 For any instruction states $q_1, q_2 \in Q^h$ and any stage $s \in \mathbb{S}$, the *stage stall* predicate $q_1 \overset{st}{\underset{h,s}{\longleftrightarrow}} q_2$ is defined as follows:

$$\begin{aligned} q_1 \overset{st}{\underset{h,s}{\longleftrightarrow}} q_2 \Longleftrightarrow \exists\, v_p \in V_p : \varphi(v_p) = s \wedge \\ \neg csat_h(\{v_p.\texttt{en} \rightsquigarrow 1\}, \{q_1, q_2\}) \wedge \\ \neg csat_h(\{v_p.\texttt{rst} \rightsquigarrow 1\}, \{q_1, q_2\}). \end{aligned} \tag{15}$$

Intuitively, the definition requires that the presence of some instructions in states q_1 and q_2 in the pipeline ensures that there is a pipeline register v_p in stage s, which we denote as a *representative register* below, such that the value of v_p can neither be updated nor cleared, i.e., v_p keeps its value. Note that the already established validity of the consistency Rules 1 and 4 implies that the setting of any control edge (`en`, `rst`) is the same for all pipeline registers across the given pipeline stage, and so the fact that some representative register is stalled means that all registers of the given stage are stalled (and the instruction that is now in stage s stays in it).

In a similar fashion, we define the $\overset{cl}{\underset{h}{\longleftrightarrow}}$ predicate.

Definition 16 For any instruction states $q_1, q_2 \in Q^h$ and any stage $s \in \mathbb{S}$, the *stage clear* predicate $q_1 \overset{cl}{\underset{h,s}{\longleftrightarrow}} q_2$ is defined as follows:

$$\begin{aligned} q_1 \overset{cl}{\underset{h,s}{\longleftrightarrow}} q_2 \Longleftrightarrow \exists\, v_p \in V_p : \varphi(v_p) = s \wedge \\ \neg csat_h(\{v_p.\texttt{rst} \rightsquigarrow 0\}, \{q_1, q_2\}). \end{aligned} \tag{16}$$

Note that the definition requires that the representative register *must* be cleared (since the formula cannot be satisfied with the v_p.`rst` edge being zero). The consistency rules then assure that the same holds for all registers of the given stage.

To define the $\overset{cf}{\underset{h}{\longleftrightarrow}}$ predicate, we only need to be able to determine whether two given instruction states are prohibited from occurring together in a single pipeline configuration by the control logic of the considered processor. This is, however, easy thanks to the $csat_h$ predicate as shown below.

Definition 17 For any instruction states $q_1, q_2 \in Q^h$, the *state conflict* predicate $q_1 \overset{cf}{\underset{h}{\longleftrightarrow}} q_2$ is defined as follows:

$$q_1 \overset{cf}{\underset{h}{\longleftrightarrow}} q_2 \Longleftrightarrow \neg csat_h(\emptyset, \{q_1, q_2\}). \tag{17}$$

Intuitively, the expression $csat_h(\emptyset, \{q_1, q_2\})$ does not put any constraints on edge conditions, but it still checks whether some concurrently executing instructions can simultaneously get into states q_1 and q_2. Hence, its negation says that this is excluded in the given processor, allowing us to define the $\overset{cf}{\underset{h}{\longleftrightarrow}}$ predicate.

Example 7. In this example, we will demonstrate how the predicate $\overset{st}{\underset{h}{\longleftrightarrow}}$ can be evaluated for a given pair of states and a given stage. Let us consider states $\langle sp, 2\rangle$, $\langle vi, 3\rangle$, Stage 2, and the hazard case $h = (\chi_{sp}, \chi_{vi})$ from Example 4. Here, the spoiler instruction in state $\langle sp, 2\rangle$ writes into the register X the (auto-incremented) value previously read from the same register. The victim instruction in state $\langle vi, 4\rangle$ then reads the value j from the register X and uses it as an index to access the memory cell Mem_j.

From Definition 15, we know that, in order to determine the value of $\langle sp, 2\rangle \overset{st}{\underset{h,2}{\longleftrightarrow}} \langle vi, 3\rangle$, one has to (i) pick a representative pipeline register $v_p \in \{v \in V_p \mid \varphi(v) = 2\}$, (ii) evaluate $\Phi_1 := \neg csat(\{v_p.\texttt{en} \rightsquigarrow 1\}, \{\langle sp, 2\rangle, \langle vi, 3\rangle\})$, and (iii) evaluate $\Phi_2 := \neg csat(\{v_p.\texttt{rst} \rightsquigarrow 1\}, \{\langle sp, 2\rangle, \langle vi, 3\rangle\})$. The representative register v_p for Stage 2 can be picked randomly—indeed, as we have already said, the consistency Rules 1 and 4 (from Section 5.2) imply that the setting of any control edges (`en`, `rst`) is the same for all pipeline registers in a particular pipeline stage.

As for Step (i) above, it suffices to look in Table 1 and choose, for instance, $IdIr$ as the representative register. Moreover, in Example 1, we have pointed out that the value of the

[8] In our implementation of the approach, we replace the existential quantification by simply pruning away all variables unrelated with any $e^\star$ for any $e \in E$ and by renaming the remaining variables in a unique way such that no conflicts arise when constructing more complex formulae on top $unwind_h^\star(q)$.

enable edge on the $IdIr$ register is determined by the following expression in BVL:

$$IdIr.\mathtt{en}^\star = \neg IncX.\mathtt{q}^\star \vee \neg OfWrMem.\mathtt{q}^\star. \tag{18}$$

Now, to address Step (ii), we know that, according to Equation 14, Φ_1 expands to

$$\begin{aligned}&\neg sat(unwind_h^\star(\langle sp, 2\rangle) \wedge unwind_h^\star(\langle vi, 3\rangle) \wedge \\ &\quad IdIr.\mathtt{en}^\star = 1).\end{aligned} \tag{19}$$

We further concetrate on the expansion of $unwind_h^\star(\langle sp, 2\rangle)$. According to Equation 12, we need to construct formulae F_1, $F_2(\langle sp, 2\rangle)$, and F_3. First, the transition relation described by Formula F_1 contains the following conjuncts[9]:

$$\begin{aligned}Impl.\mathtt{q}_0^\star &= (IncX.\mathtt{q}_0^\star \Rightarrow ExWrX.\mathtt{q}_0^\star) \wedge \\ MxInc.\mathtt{sel}_0^\star &= Impl.\mathtt{q}_0^\star.\end{aligned} \tag{20}$$

To see that the above holds, it suffices to check how the value of $MxInc.\mathtt{sel}$ is computed from its predecesors in the PSG shown in Fig. 2. The formula $F_2(\langle sp, 2\rangle)$ then gives

$$MxInc.\mathtt{sel}_0^\star = 0 \wedge X.\mathtt{en}_0^\star = 1, \tag{21}$$

which is a direct consequence of the result that we have obtained in Example 6 where we have shown

$$\alpha(\langle sp, 2\rangle) = \{MxInc.\mathtt{sel} \rightsquigarrow 0, X.\mathtt{en} \rightsquigarrow 1\}.$$

Finally, Formula F_3 simply asserts equality between zero-indexed and non-indexed variables. We can then apply the existential quantification from Equation 12, which allows us to get rid of the indexed variables, leading to that the below equality must hold:

$$IncX.\mathtt{q}^\star = 1. \tag{22}$$

Now, we will apply a similar approach to expand the formula $unwind_h^\star(\langle vi, 3\rangle)$. In this case, the following conjuncts of Formula F_1 turn out to be relevant:

$$\begin{aligned}ExWrMem.\mathtt{d}_0^\star &= OfWrMem.\mathtt{q}_0^\star &\wedge \\ ExWrMem.\mathtt{q}_1^\star &= f_{ExWrMem}^{next}(ExWrMem.\mathtt{q}_0^\star, \\ &\qquad ExWrMem.\mathtt{d}_0^\star, ExWrMem.\mathtt{en}_0^\star, \\ &\qquad ExWrMem.\mathtt{rst}_0^\star) &\wedge \\ MxSel_j.\mathtt{sel}_1^\star &= ExWrMem.\mathtt{q}_1^\star.\end{aligned} \tag{23}$$

Above, $f_{ExWrMem}^{next}$ is the next-state function that was defined in Section 3.2 and that propages the value on the data-in edge `d` to the data-out edge `q` iff the enable edge `en` is set and the reset edge `rst` is unset. Moreover, if `rst` is set, then the data-out `q` is nullified. Otherwise, when both `en` and `rst` are unset, the data-out edge `q` keeps the value from the previous cycle. Further, in Example 6, we have seen that

$$\alpha(\langle vi, 4\rangle) = \{MxSel_j.\mathtt{sel} \rightsquigarrow 1\},$$

which imples that the formula $F_2(\langle vi, 3\rangle)$ must ensure

$$MxSel.\mathtt{sel}_1^\star = 1. \tag{24}$$

By combining the observations from Formulae 23 and 24, and by adding Formula F_3 and the existential quantification of Equation 12, we obtain the following statement:

$$\begin{aligned}&\big((ExWrMem.\mathtt{en}^\star = 1) \Rightarrow (OfWrMem.\mathtt{q}^\star = 1) \big) \wedge \\ &\qquad ExWrMem.\mathtt{rst}^\star = 0.\end{aligned} \tag{25}$$

Here, the $ExWrMem.\mathtt{rst}^\star = 0$ conjuct comes from the fact that the data-out edge must not be zero because of the constraint in Formula 24.

Next, according to the consistency Rule 3 from Section 5.2, which holds globally at any pipeline cycle, any pipeline stage that directly precedes the currently stalled one must also be stalled. By induction, the rule can be generalized to any preceding stage. Therefore, the following expression must hold:

$$\begin{aligned}&(ExWrMem.\mathtt{en}^\star = 0 \wedge ExWrMem.\mathtt{rst}^\star = 0) \Rightarrow \\ &\qquad (IdIr.\mathtt{en}^\star = 0 \wedge IdIr.\mathtt{rst}^\star = 0).\end{aligned} \tag{26}$$

In particular, the above comes from the fact that $\varphi(IdIr) < \varphi(ExWrMem)$.

By applying the modus tollens rule on Formula 26, we get

$$\begin{aligned}&(IdIr.\mathtt{en}^\star = 1 \vee IdIr.\mathtt{rst}^\star = 1) &\Rightarrow \\ &(ExWrMem.\mathtt{en}^\star = 1 \vee ExWrMem.\mathtt{rst}^\star = 1).\end{aligned} \tag{27}$$

Finally, if we put together our observations made in Formulae 18, 22, 25, and 27, we can conclude that the expression

$$unwind_h^\star(\langle sp, 2\rangle) \wedge unwind_h^\star(\langle vi, 3\rangle) \wedge IdIr.\mathtt{en}^\star = 1$$

is not satisfiable. Thus, the expression Φ_1 evaluates to $true$.

Analogically, for Step (iii), we would also derive that Φ_2 is $true$, and therefore the predicate $\langle sp, 2\rangle \xleftrightarrow[h,2]{st} \langle vi, 3\rangle$ necessarily holds. In other words, this means that the `NOP` injection into Stage 3 takes place whenever there is a spoiler defined by χ_{sp} in Stage 2 and a victim described by χ_{vi} in Stage 3. ◁

[9] The entire formula is, of course, much bigger—indeed, it describes the entire transition relation. When the satisfiability checking is done automatically, the solver will consider the entire formula. However, we select its relevant parts only so that the example is readable.

We can now define transitions that the transition relation Δ^h of the HS P^h contains. First, for every instruction state $q = \langle \kappa, s \rangle \in Q^h$, Δ^h contains a transition $q \to q$ allowing the instruction that is in q to stay in q whenever the state q appears in a configuration of the pipeline of the given processor (i.e., a configuration of the transition system induced by P^h) that contains a combination of instruction states q_1, $q_2 \in Q^h$ which causes the instruction in the state q to be stalled. Formally, $\forall q = \langle \kappa, s \rangle, q_1, q_2 \in Q^h$:

$$(\exists_{\leftrightarrow} \colon \{q_1, q_2\} \models q \to q) \in \Delta^h \Leftrightarrow q_1 \xleftrightarrow[h,s]{st} q_2. \tag{28}$$

As we have already mentioned at the beginning of Section 7, we use the stall-flow sf and normal-flow nf instruction classes to model pipeline-filler instructions, i.e., to model all other instructions than the spoiler and victim. The difference between the stall- and normal-flow operation modes is that an sf-class instruction in a stage $s' \in \mathbb{S}$ causes all pipeline stages $s \in \mathbb{S}$ s.t. $s < s'$ to be stalled. In other words, an instruction stays in a state $q = \langle \kappa, s \rangle \in Q^h$ whenever q appears in a configuration of the pipeline containing an earlier instruction in the stall-flow operation mode. Formally, $\forall q = \langle \kappa, s \rangle$, $q' = \langle sf, s' \rangle \in Q^h$:

$$(\exists_{\leftarrow} \colon \{q'\} \models q \to q) \in \Delta^h \Leftrightarrow s < s'. \tag{29}$$

Including stalls caused by stall-flow instructions is necessary as they may introduce otherwise unreachable configurations of the verified HS P^h. Moreover, since a pipeline stall caused by some filler instruction may occur at any processor cycle, we will always allow random transitions between stall- and normal-flow operation modes of filler instructions in the upcoming explanation.

Next, an instruction in a state $q = \langle \kappa, s \rangle \in \widehat{Q^h}$, $\widehat{Q^h} = \mathbb{K} \times \widehat{\mathbb{S}}$, $\widehat{\mathbb{S}} = \mathbb{S} \setminus \{\max(\mathbb{S})\}$, is cancelled, i.e., yields a transition $q \to \langle \kappa', s+1 \rangle$, $\kappa' \in \{nf, sf\}$, provided that q appears in a configuration of the pipeline in which there exist instructions in states q_1 and q_2 that cause the stage $s+1$ to be cleared. More formally, $\forall q = \langle \kappa, s \rangle \in \widehat{Q^h}$, $\forall q_1, q_2 \in Q^h$, $\forall \kappa' \in \{nf, sf\}$:

$$\begin{aligned}(\exists_{\leftrightarrow} \colon \{q_1, q_2\} \models q \to \langle \kappa', s+1 \rangle) \in \Delta^h \Leftrightarrow \\ q_1 \xleftrightarrow[h,s+1]{cl} q_2 \wedge \neg\Big(q_1 \xleftrightarrow[h,s]{st} q_2\Big).\end{aligned} \tag{30}$$

Note that for a successful clearing of an instruction in the stage s, it is also required that s is not stalled at the same time.

For the case when our over-approximating abstraction allows two states q and q' that are conflicting to be reached in a single configuration of the transition system induced by the HS P^h, we introduce the following solution to reduce the number of possible false alarms. Namely, we kill the instruction that entered the pipeline later assuming that this instruction is in the state $q = \langle \kappa, s \rangle$, i.e., we introduce the transition $q \to \langle \kappa', s+1 \rangle$, $\kappa' \in \{nf, sf\}$, into Δ^h. Formally, $\forall q = \langle \kappa, s \rangle \in \widehat{Q^h}$, $\forall q' \in Q^h$, $\forall \kappa' \in \{nf, sf\}$:

$$(\exists_{\leftarrow} \colon \{q'\} \models q \to \langle \kappa', s+1 \rangle) \in \Delta^h \Leftrightarrow \langle \kappa, s \rangle \xleftrightarrow[h]{cf} q'. \tag{31}$$

As for the possibility of new instructions entering the pipeline, only the left-most instruction in a given configuration that has so far not entered the pipeline is allowed to enter it. Moreover, new instructions cannot enter the first stage if it is stalled. More precisely, $\forall q = \langle \kappa, \bot \rangle$, $q' = \langle \kappa', \bot \rangle$, $q_1, q_2 \in Q^h$:

$$(\exists_{\leftarrow} \colon \{q'\} \models q \to q) \in \Delta^h, \tag{32}$$

$$(\exists_{\leftrightarrow} \colon \{q_1, q_2\} \models q \to q \in \Delta^h) \Leftrightarrow q_1 \xleftrightarrow[h,1]{st} q_2. \tag{33}$$

Next, an instruction can proceed to the next stage iff none of the above rules is applicable. To model this fact, we use local transitions, building on that we define all global transitions (used above) to be of a higher probability than the local ones. Further, we add transitions reflecting that once finalized instructions stay in their final state forever. More rigorously, $\forall \langle \kappa, s \rangle \in \widehat{Q^h}$:

$$(\langle \kappa, s \rangle \to \langle \kappa, s+1 \rangle) \in \Delta^h, \tag{34}$$

$$(\langle \kappa, \bot \rangle \to \langle \kappa, 1 \rangle) \in \Delta^h, \tag{35}$$

$$(\langle \kappa, \max(\mathbb{S}) \rangle \to \langle \kappa, \top \rangle) \in \Delta^h, \tag{36}$$

$$(\langle \kappa, \top \rangle \to \langle \kappa, \top \rangle) \in \Delta^h. \tag{37}$$

To ensure a possibility of the pipeline being stalled by some filler instruction, we allow switching between stall- and normal-flow operation modes. More formally, $\forall \langle sf, s \rangle$, $\langle nf, s \rangle \in \widehat{Q^h}$:

$$(\langle nf, s \rangle \to \langle sf, s+1 \rangle) \in \Delta^h, \tag{38}$$

$$(\langle sf, s \rangle \to \langle nf, s+1 \rangle) \in \Delta^h. \tag{39}$$

Finally, we recall that global (i.e., guarded) transitions have a higher priority than local (i.e., unguarded) ones. That is, only if no global transition can be applied (such as a stall), a local one may be applied (e.g., proceeding to the next stage). Additionally, the transition relation Δ^h is constructed under the assumption that, in each step of the transition system induced by the HS P^h, each instruction whose state is a part of the given configuration of P^h must make a step. This is, if we take, e.g., a configuration $q_1 q_2 q_3$ consisting of three states of three instructions, all of the three instructions must synchronously fire some of the above described transitions such that we get the successor configuration $q_1' q_2' q_3'$.

Table 2: Roles of e-/ℓ-class instructions in hazards cases.

Hazard	e-class	Role	ℓ-class	Role
RAW	writes	spoiler (too slow)	reads	victim
WAR	reads	victim	writes	spoiler (too fast)
WAW	writes	victim	writes	spoiler (too fast)
CTL	writes	spoiler (too slow)	jumps	victim

7.3 Construction of the Minimal Bad Set

In the previous section, we have constructed a hazard system $P^h = (Q^h, \Delta^h, \alpha^h)$ that models possible interactions of a spoiler and a victim instruction, forming a hazard case $h = (\chi_{sp}, \chi_{vi}) \in \mathbb{H} \subseteq \mathbb{X}^2$, surrounded by other instructions during a pipelined execution. We now need to be able to check whether some kind of data or control hazard occurs.

To facilitate detection of possible hazards from the constructed HS, we will construct a set $B^h_\top$ of *minimal bad configurations* describing minimal illegal configurations whose reachability (within possibly larger configurations) will mean that the given hazard case h does indeed lead to a hazard. We define the set $B^h_\top$ wrt an extended hazard system $P^h_\top$ (defined later in this section), which is obtained by applying four transformations, described also later in the section, on the input system P^h. Since the ordering of instructions within a hazard case $h \in \mathbb{H}$ is an important factor in the following explanation, we will be speaking about pairs of instruction classes consisting of an e ("earlier") instruction class and an ℓ ("later") instruction class such that either $e = sp \wedge \ell = vi$ or $e = vi \wedge \ell = sp$, meaning that an earlier instruction always enters the pipeline sooner than the later one. For the e- and ℓ-class instructions, one of the following statements always holds: (a) For RAW and CTL hazards, the e-class instruction is a spoiler that enters the pipeline first and should write data to be read by the later instruction, but it is too slow and the later victim instruction uses obsolete data. (b) For WAR and WAW hazards, the spoiler sp is an ℓ-class instruction that enters the pipeline later, but it is too fast and it either destroys data to be read by the earlier victim instruction (WAR), or it stores its result too early and the result is overwritten by the obsolete result of the earlier instruction (WAW). To formalize the above for later use in this section, we define functions $\mathcal{E}, \mathcal{L} \colon \mathbb{H} \to \mathbb{K}$ so that $\mathcal{E}(h)$ gives the meaning of the e class in a hazard case $h \in \mathbb{H}$ while $\mathcal{L}(h)$ gives the meaning of the ℓ class. All these scenarios are summarized in Table 2.

We are going to build the set $B^h_\top$ such that it will contain so-called *hazard pairs* $q^1_e q^1_\ell, \ldots, q^n_e q^n_\ell$ of states of the earlier and later instruction such that a hazard described by the hazard case h may occur iff there exists a configuration of the system $P^h_\top$ that contains as a subword some hazard pair from the set $B^h_\top$ and that is reachable from the set of initial configurations LC: $I^h_\top$. Note, however, that the control states of the earlier/later instructions that signify that something relevant for the hazard has happened (some critical value has been written or read) do not necessarily occur at the same time. On the other hand, hazard pairs consist of pairs of states that should be reached at the same time. To resolve this discrepancy, we will pass information that the critical control state of an instruction has been reached to its successor states. For that, we will introduce several auxiliary notions, which will be introduced such that the detection of the different kinds of hazards may be described in an as uniform way as possible.

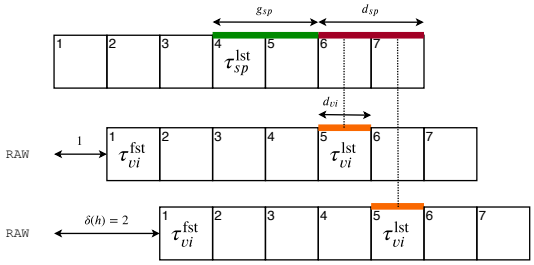
We first introduce the *hazard distance* δ that, intuitively, determines the maximum delay (measured in pipeline cycles) with which the later instruction can still cause a hazard. Intuitively, the basis of the distance is the difference in the number of the stages in which the colliding read/write operations happen within the concerned instructions. However, sometimes, this basic difference has to be decreased by one since one of the colliding operations must appear by at least one cycle earlier than the other, while in other cases a hazard appears even when they occur at the same time. More details on that are given below the definition, and an illustration is provided in Fig. 3.

Definition 18 The *hazard distance* $\delta \colon \mathbb{X} \times \mathbb{X} \to \mathbb{N}$ is defined as follows for all hazards $h = (\chi_{sp}, \chi_{vi}) \in \mathbb{X} \times \mathbb{X}$ where $\chi_k = (\pi_k, \tau_k)$ for $k \in \{sp, vi\}$:

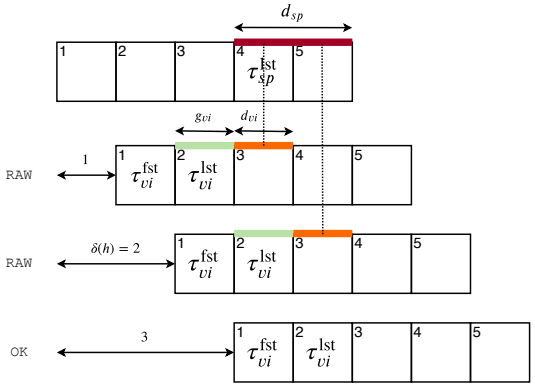
$$\delta(h) = \begin{cases} \tau^{\text{lst}}_{sp} - \tau^{\text{fst}}_{vi} - 1 & \text{if } h \text{ is a RAW or CTL hazard,} \\ \tau^{\text{fst}}_{vi} - \tau^{\text{lst}}_{sp} & \text{if } h \text{ is a WAR hazard, and} \\ \tau^{\text{lst}}_{vi} - \tau^{\text{lst}}_{sp} - 1 & \text{if } h \text{ is a WAW hazard.} \end{cases}$$

Notice that the hazard distance is indeed always non-negative as the definitions of RAW and CTL hazard cases (Definitions 11, 14) imply that $\tau^{\text{fst}}_{vi} < \tau^{\text{lst}}_{sp}$, and the definitions of WAR and WAW hazards (Definitions 12, 13) imply that $\tau^{\text{lst}}_{sp} < \tau^{\text{fst}}_{vi}$ (and, for the case of WAW hazards, one can add the fact that $\tau^{\text{fst}}_{vi} < \tau^{\text{lst}}_{vi}$). For RAW and CTL hazard cases, the distance is decremented by one because reading a value at a cycle when its writing was finished, which is what the corresponding value of τ records (recall that the writing starts one cycle earlier), is safe. On the other hand, in WAR hazards, overwriting the value that is read by the earlier instruction at the same time is an error. Finally, WAW hazards are special in that the conflict arises between two write operations where the most extreme case arises when the write operation in the spoiler appears one cycle before the write in the victim: that is why, we have the decrement by one in the formula of WAW hazards.

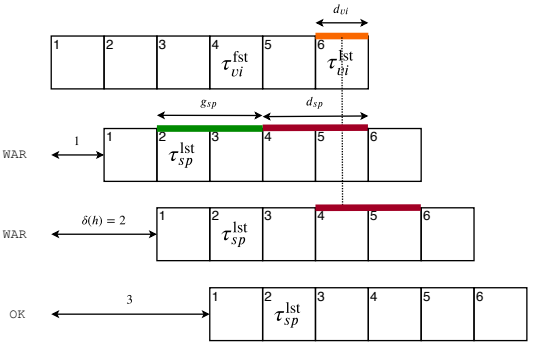
We will next introduce the so-called spoiler/victim gap and detection windows. Intuitively, the *gap window* g_{sp}/g_{vi} of a spoiler/victim instruction ι will tell us for how many cycles one has to wait within the execution of ι, starting from



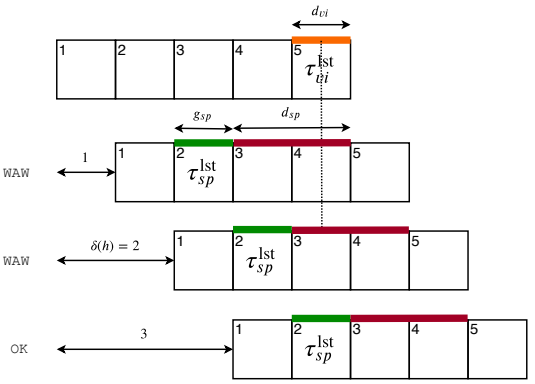
(a) Detection of a RAW hazard using a delay in the spoiler: $\delta(h) = \tau^{\text{lst}}_{sp} - \tau^{\text{fst}}_{vi} - 1 = 4 - 1 - 1 = 2$, $g_{sp} = \tau^{\text{lst}}_{vi} - \tau^{\text{lst}}_{sp} + 1 = 5 - 4 + 1 = 2$, $g_{vi} = 0$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.



(b) Detection of a RAW hazard using a delay in the victim: $\delta(h) = \tau^{\text{lst}}_{sp} - \tau^{\text{fst}}_{vi} - 1 = 4 - 1 - 1 = 2$, $g_{sp} = 0$, $g_{vi} = \tau^{\text{lst}}_{sp} - \tau^{\text{lst}}_{vi} - 1 = 4 - 2 - 1 = 1$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.



(c) Detection of a WAR hazard using a delay in the spoiler: $\delta(h) = \tau^{\text{fst}}_{vi} - \tau^{\text{lst}}_{sp} = 4 - 2 = 2$, $g_{sp} = \tau^{\text{lst}}_{vi} - \tau^{\text{fst}}_{vi} = 6 - 4 = 2$, $g_{vi} = 0$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.



(d) Detection of a WAW hazard using a delay in the spoiler: $\delta(h) = \tau^{\text{lst}}_{vi} - \tau^{\text{lst}}_{sp} - 1 = 5 - 2 - 1 = 2$, $g_{sp} = 1$, $g_{vi} = 0$, $d_{sp} = \delta(h) = 2$, and $d_{vi} = 1$.

Fig. 3: An illustration of hazard distances together with gap and detection windows used to construct minimal bad sets.

its critical write operation, until the detection of a possible hazard may start. In some cases, the gap will be zero while in some other cases it will be positive. The latter case will happen when the victim/spoiler instruction ι', possibly colliding with ι, has no chance to perform its write operation before the moment when the write operation of ι happens even if ι' starts right after ι. The *detection window* (of size at least one) will then tell us for how many cycles the detection of a possible hazard should be performed within a given instruction after the gap window passes.

In particular, we will define all the windows such that the detection window of victim instructions, denoted d_{vi}, will be fixed to one, i.e., $d_{vi} = 1$. Intuitively, the hazard detection will always be performed as soon as the victim instruction writes (and hence "publishes") the wrong data and the gap window of that instruction is over.

The detection window of a spoiler instruction will be possibly longer, in particular, it will correspond to the hazard distance, i.e., $d_{sp} = \delta(h)$ where $h = (\chi_{sp}, \chi_{vi})$ is the considered hazard case. The definition of the gap windows must then be done in such a way that any hazard may be detected with the detection windows defined as above, i.e., the detection within the particular instructions must be postponed such that the hazard can always be caught within the detection windows. This definition is more complex and is given below separately for different types of hazards.

Gap Windows for RAW and CTL Hazards

First, notice that $\tau^{\text{fst}} < \tau^{\text{lst}}$ holds for each forward execution $(\pi, \tau) \in \mathbb{X}$ where $\pi^{\text{fst}}, \pi^{\text{lst}} \in V_r$. Second, recall that the definitions of RAW and CTL hazard cases (Definitions 11, 14) imply that $\tau^{\text{fst}}_{vi} < \tau^{\text{lst}}_{sp}$. If put together, one can see that there are two possible orderings of τ^{fst}_{vi}, τ^{lst}_{vi}, and τ^{lst}_{sp}:

$$\tau^{\text{fst}}_{vi} < \tau^{\text{lst}}_{sp} \leq \tau^{\text{lst}}_{vi} \tag{40}$$

$$\tau^{\text{fst}}_{vi} < \tau^{\text{lst}}_{vi} < \tau^{\text{lst}}_{sp} \tag{41}$$

We start with the ordering (40), which is illustrated by the scenarios in Fig. 3(a). In this case, the spoiler finishes

its write operation earlier, and the RAW hazard occurs as soon as the victim performs its write operation. Hence, in order to be able to detect the hazard via states simultaneously reached in the spoiler and the victim, the detection needs to be put off in the spoiler. Provided that the we consider a victim that starts right after the spoiler, $\tau_{vi}^{\text{lst}} - \tau_{sp}^{\text{lst}} + 1$ cycles need to be skipped in the spoiler (including the cycle in which the write operation of the spoiler happens), and so $g_{sp} = \tau_{vi}^{\text{lst}} - \tau_{sp}^{\text{lst}} + 1$.[10] On the other hand, no cycles need to be skipped before the detection starts in the victim, and so $g_{vi} = 0$. Note that the detection of hazards with victims that start later than one cycle behind the spoiler is handled through the detection window d_{sp}.

Next, we consider the ordering (41), which is illustrated in Fig. 3(b). In this case, the victim performs the write operation first, and the hazard occurs as soon as the spoiler performs its write operation. Hence, this time, the detection needs to be put off in the victim. Using a similar reasoning as above, we define $g_{sp} = 0$ and $g_{vi} = \tau_{sp}^{\text{lst}} - \tau_{vi}^{\text{lst}} - 1$.[11]

Gap Windows for WAR Hazards

For an illustration of the gap and detection windows of WAR hazards, see Fig. 3(c). As above, we can use the fact that $\tau^{\text{fst}} < \tau^{\text{lst}}$ holds for each forward execution $(\pi, \tau) \in \mathbb{X}$ where $\pi^{\text{fst}}, \pi^{\text{lst}} \in V_r$. Moreover, the definition of WAR hazards (Definition 12) implies that $\tau_{sp}^{\text{lst}} < \tau_{vi}^{\text{fst}}$. Hence, for WAR hazards, τ_{vi}^{fst}, τ_{vi}^{lst}, and τ_{sp}^{lst} can be ordered as follows only:

$$\tau_{sp}^{\text{lst}} \quad < \quad \tau_{vi}^{\text{fst}} \quad < \quad \tau_{vi}^{\text{lst}} \tag{42}$$

Intuitively, after the spoiler instruction writes, the WAR hazard does not occur until the victim performs its write as well. Unlike for RAW/CTL hazards, we now consider as the base case not the situation when the later instruction starts right after the earlier, but the case when the later instruction starts as late as possible to be still able to cause a hazard, i.e., the case when the spoiler starts $\delta(h)$ cycles after the victim. Then, it is easy to see that the detection needs to be put off by $\tau_{vi}^{\text{lst}} - (\tau_{sp}^{\text{lst}} + \delta(h))$ cycles. Hence, we define $g_{sp} = \tau_{vi}^{\text{lst}} - (\tau_{sp}^{\text{lst}} + \delta(h)) = \tau_{vi}^{\text{lst}} - (\tau_{sp}^{\text{lst}} + \tau_{vi}^{\text{fst}} - \tau_{sp}^{\text{lst}}) = \tau_{vi}^{\text{lst}} - \tau_{vi}^{\text{fst}}$ while $g_{vi} = 0$. The cases of the spoiler that start sooner are then handled appropriately by using the detection window $d_{sp} = \delta(h)$ as also illustrated in Fig. 3(c).

Gap Windows in WAW Hazards

As with WAR hazards, for WAW hazards, the ordering between writes given in Equation 42 is the only possible. After the spoiler instruction writes, the WAW hazard does not occur until the victim performs its write as well. This cannot happen sooner than after passing through at least one pipeline stage. Therefore, we put the spoiler gap distance equal to one and the victim gap distance equal to zero, i.e., $g_{sp} = 1$ and $g_{vi} = 0$.

Tracking Passage through Gap and Detection Windows

To facilitate tracking whether a spoiler/victim instruction is inside a gap or detection window and, if so, how far inside the window it is, we will introduce a notion of *extended hazard systems* (EHS). In an EHS, each state of the execution of a spoiler/victim instruction will be labelled by a set of tags saying whether the write operation of the spoiler/victim has already happened and, if so, how many cycles have passed since then. The universe of tags $\mathcal{T}$ will therefore include all couples from the set $\{\texttt{win}_{sp}, \texttt{win}_{vi}\} \times \mathbb{N}$. The universe of tags is, however, not defined to be equal to the above set since we will need to add some more tags into it later on when we examine the effect of stalling of an instruction, which we will need to reflect in the tags as well. We defer the discussion of the stalling-related tags after we properly explain the basic spoiler/victim tags.

Below, we will introduce the EHSs step-wise by first adding tracking of spoiler windows, then victim windows, and then adding tracking of stalled instructions. This will lead to introduction of EHSs of various levels, with the zero level being the original hazard system, level one being the extension by tracking spoilers, etc.

More formally, for a hazard case $h = (\chi_{sp}, \chi_{vi})$ and the associated HS $P^h = (Q^h, \Delta^h, \alpha^h)$, the corresponding *extended hazard system (EHS) of level* $n \geq 0$ is a tuple $P_n^h = (Q_n^h, \Delta_n^h, \alpha_n^h, \beta_n^h)$ where:

1. Q_n^h is a finite subset of the set $Q^h \times (\mathbb{N} \cup \{\bot, \top\})^n$.[12] We let $Q_0^h = Q^h$, and we give the precise construction of the set Q_n^h for $n \geq 1$ below. Intuitively, the additional components of the states will allow us to track the passage of the spoiler/victim instructions through the gap and detection windows, for which some states of the original HS will need to be split to multiple occurrences to reflect whether an instruction in that state is in the window and, if so, how far. Moreover, some further splitting will be needed when some of the tracked instructions are stalled some number of times. The finiteness of Q_n^h will stem from that the tracked gap and detection windows are finite, that we are tracking a pair of instructions, and that the stalling can happen for finite time only.

[10] Intuitively, the addition of 1 is needed since the victim starts by one cycle later. Further, note that the gap is appropriately defined also for the case when $\tau_{sp}^{\text{lst}} = \tau_{vi}^{\text{lst}}$ when a gap window of size 1 is needed to compensate the fact that the victim starts by one cycle later.

[11] The subtraction of 1 comes from that the spoiler starts by one cycle earlier.

[12] For convenience, by a slight abuse of the notation, we let $(Q^h \times (\mathbb{N} \cup \{\bot, \top\})) \times (\mathbb{N} \cup \{\bot, \top\}) = Q^h \times (\mathbb{N} \cup \{\bot, \top\}) \times (\mathbb{N} \cup \{\bot, \top\})$ and $((q, i_1), i_2) = (q, i_1, i_2)$ for any $q \in Q^h$ and $i_1, i_2 \in \mathbb{N} \cup \{\bot, \top\}$, and likewise for higher values of n.

2. The transition relation Δ_n^h and the labelling function α_n^h lift the transition relation Δ^h and the labelling function α^h to the extended set of states. We have $\Delta_0^h = \Delta^h$ and $\alpha_0^h = \alpha^h$, and the construction of the relations for $n \geq 1$ is described in detail below.
3. Finally, $\beta_n^h \colon Q_n^h \to 2^{\mathcal{T}}$ is the new tag function. We let $\beta_0^h(q) = \emptyset$ for any $q \in Q_0^h$. For $n \geq 1$, the construction of the function will also be shown below.

For $n \geq 1$, the construction of the EHS P_n^h will be based on applying Alg. 2 and 3 several times on the EHS P_0^h. We start by presenting Alg. 2 that implements a procedure denoted as $window$. Given a set S of starting states, this procedure extends the input EHS such that it allows for tracking a spoiler/victim instruction, which performs some critical operation w (such as a write to its target register) in a state $q' \in S$, and then passes through the tracked gap and detection windows whose combined length is k. Here, note that we monitor the gap and detection windows joint into one window which is possible since the latter follows immediately after the former (and we can distinguish in which of the original windows we are by just looking at how deep into the combined window we are).

Intuitively, the algorithm extends all states of the input EHS by one more component that ranges over the set $I := \{\bot, \top, 0, \ldots, k-1\}$. When the additional component is $\bot$, the tracked instruction has not yet entered the gap/detection window. If the additional component i is from the set $\{0, \ldots, k-1\}$, the instruction is in the tracking window for $i+1$ cycles. If the additional component is $\top$, the instruction has already got out of the window.

The transition relation is updated straightforwardly such that the monitoring phase can be entered whenever an instruction is in some state from the given set S (and the monitoring has not yet started). If the monitoring is started, every executed transition increases the number of cycles spent in the window (recorded in the additional component of states) until the end of the window is reached. Note that, for transitions with guards, the states used in the guards must be lifted to the new set of states, which is done by allowing them to appear with any value of the additional component. Indeed, satisfaction of the guard is not subject to the cycle in which it is reached.

The α function does not depend on the additional component, and so it is lifted to the new set of states by ignoring the additional component. On the other hand, the β function is extended such that states that are inside the monitored window will be tagged by a couple (w, i), which says that the operation w is in the $(i+1)$-th cycle of its gap/detection window.

Now, before we can apply the $window$ transformations, we need to identify the set S of states where the critical write operations happen and the tracking of the passage of the gap/detection windows starts. Therefore, we introduce

Algorithm 2 The $window$ procedure transforming an EHS P_n^h to an EHS P_{n+1}^h to facilitate tracking of the execution of an instruction that performs a critical write operation w in a state from some given set S through a window of some given length k.

Require: An EHS $P_n^h = (Q_n^h, \Delta_n^h, \alpha_n^h, \beta_n^h)$ of any level $n \geq 0$, a set $S \subseteq Q_n^h$ of states to start the transformation from, a tag $w \in \{\texttt{win}_{sp}, \texttt{win}_{vi}\}$, and the length of the tracking window $k \in \{1, \ldots, \max(\mathbb{S})\}$.

Ensure: An EHS $P_{n+1}^h = (Q_{n+1}^h, \Delta_{n+1}^h, \alpha_{n+1}^h, \beta_{n+1}^h)$ where each state based on $q \in S$ together with its k reachable successors is tagged by a pair (w, i) where $0 \leq i < k$ denotes the distance of the successor from the original occurrence of q.

1: $I := \{\bot, \top, 0, \ldots, k-1\}$.

2: $Q_{n+1}^h := Q_n^h \times I$.

3: Δ_{n+1}^h is defined as the minimal relation such that the following two conditions hold:
(a) For every global transition $\mathbb{Q}_\circ \colon G \models q_1 \to q_2 \in \Delta_n^h$ and for every injection $\Gamma : Q_n^h \to I$, the following transitions are in Δ_{n+1}^h:
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, \bot) \to (q_2, \bot)$,
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, \bot) \to (q_2, 0)$ if $q_2 \in S$,
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, i+1)$ for all $0 \leq i < k-1$,
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, \top)$ for $i = k-1$,
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, \top) \to (q_2, \top)$

where $\widehat{\Gamma} \colon 2^{Q_n^h} \to 2^{Q_{n+1}^h}$ is defined such that $\forall Q' \subseteq Q_n^h \colon \widehat{\Gamma}(Q') := \{(q, \Gamma(q)) \mid q \in Q'\}$.

(b) For every local transition $q_1 \to q_2 \in \Delta_n^h$, the following transitions are in Δ_{n+1}^h:
- $(q_1, \bot) \to (q_2, \bot)$,
- $(q_1, \bot) \to (q_2, 0)$ if $q_2 \in S$,
- $(q_1, i) \to (q_2, i+1)$ for all $0 \leq i < k-1$,
- $(q_1, i) \to (q_2, \top)$ for $i = k-1$,
- $(q_1, \top) \to (q_2, \top)$.

4: $\forall (q, i) \in Q_n^h \times I : \alpha_{n+1}^h(q, i) = \alpha_n^h(q)$.

5: $\forall (q, i) \in Q_n^h \times \{\bot, \top\} : \beta_{n+1}^h(q, i) = \beta_n^h(q)$.

6: $\forall (q, i) \in Q_n^h \times \{0, \ldots, k-1\} : \beta_{n+1}^h((q, i)) = \beta_n^h(q) \cup \{(w, i)\}$.

the following auxiliary mapping. Given an EHS $P_n^h = (Q_n^h, \Delta_n^h, \alpha_n^h, \beta_n^h)$ of level n, we define $wr_{P_n^h} \colon \mathbb{K} \times \mathbb{X} \to 2^{Q_n^h}$ as the function that maps a class $\kappa \in \mathbb{K}$ and any execution $(\pi, \tau) \in \mathbb{X}$ to the set $\{q \in Q_n^h \mid q = \langle \kappa', s, i_1, \ldots, i_n \rangle \wedge \kappa' = \kappa \wedge s = \tau(\pi^{\text{lst}})\}$ of all the states of P_n^h where a κ-class instruction makes the write to its target register π^{lst} in the execution (π, τ). From the definition of the execution, we know that such a write occurs in the stage $\tau(\pi^{\text{lst}})$.

We can now proceed to the transformation of the original P_0^h to the EHS P_1^h that is extended to track the spoiler gap and detection windows. With the above notation and algorithm in hand, the EHS P_1^h can be obtained simply as:

$$P_1^h := window\Big(P_0^h, wr_{P_0^h}(sp, \chi_{sp}), \texttt{win}_{sp}, g_{sp} + d_{sp}\Big).$$

Indeed, the critical operation is writing in a spoiler, which we denote as $\texttt{win}_{sp}$. The write operation can happen in one of the states returned by $wr_{P_0^h}(sp, \chi_{sp})$. These states thus serve as the initial states for tracking the gap and detection windows. Their sizes are g_{sp} and d_{sp}, respectively, which gives the length $g_{sp} + d_{sp}$ of the combined window whose tracking is ensured in P_1^h by Alg. 2.

The EHS P_2^h extended to track the victim gap and detection windows can be obtained from P_1^h in a very similar way as follows:

$$P_2^h := window\Big(P_1^h, wr_{P_1^h}(vi, \chi_{vi}), \texttt{win}_{vi}, g_{vi} + d_{vi}\Big).$$

An example of a computation of the tracking window is demonstrated in Fig. 4.

Tracking Windows in Stalled Instructions

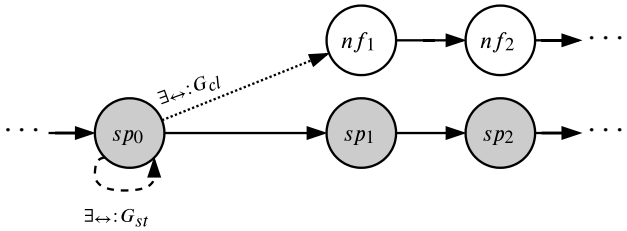
Since our approach builds on counting the exact number of cycles spent within the tracking windows, we also need to deal with any scenario when an ℓ-class instruction is stalled while the corresponding e-class instruction is not.[13] This scenario breaks the counting scheme introduced in the previous paragraphs as the later instruction can get delayed and the earlier instruction might get out of the detection window before the later one gets into its detection window. The goal of the following transformations is to compensate such misalignments by (1) using so-called *slack tags* to count how many times the later instruction gets stalled and (2) by expanding the detection window of the earlier instruction correspondingly.

The introduction of slack tags, which are drawn from the set $\{\texttt{sl}\} \times \mathbb{N}$, is implemented in Alg. 3, which takes us from the EHS P_2^h obtained by the previous transformations to EHS P_3^h as follows:

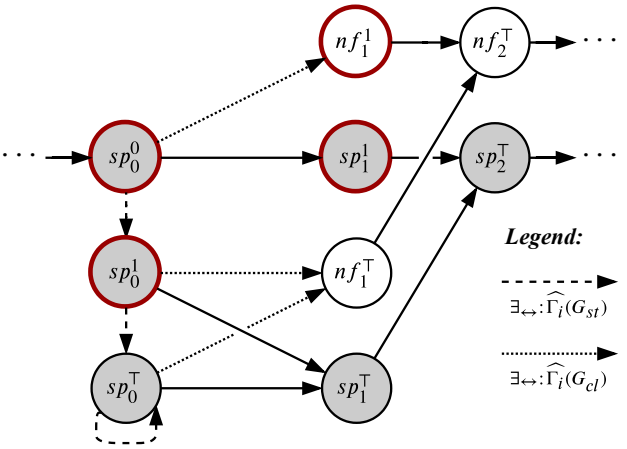
$$P_3^h := slack(P_2^h, \max(\mathbb{S})).$$

Intuitively, all states from the EHS P_2^h are considered to have the initial slack zero. Then, whenever a self-loop on any such state is possible, the self-loop is changed into a transition going to a new copy of the concerned state with the slack being one. More generally, a self-loop on a state with the slack being i is transformed into a transition to a new copy of that state with the slack being $i+1$ (unless the number of slack steps reaches the maximum number of pipeline stages—going to such a number and beyond is not necessary since such behaviours are ruled out by the initial sanity checks). The number of stalls (slack transitions) performed by an instruction is thus remembered in the structure of the states, and, in addition, we add it into the tags of the states at the end of Alg. 3 so that the slack information is easier to access. An example of an application of the *slack* mapping is demonstrated in Fig. 5.

[13] The converse cannot happen due to the basic consistency checks that we perform.



(a) A part of an EHS $P_n^h = (Q_n^h, \Delta_n^h, \alpha_n^h, \beta_n^h)$ modeling the behavior of a spoiler instruction before an application of the $window$ procedure. Note that the spoiler instruction in the state sp_0 might stall (if there are instructions from the set G_{st}), be cleared (if there are instructions from the set G_{cl}), or proceed to the next stage (represented by the state sp_1). The different styles of the lines representing global transitions are used to allow for better matching with the counterparts of the transitions in Fig. 4 (b).



(b) A part of the EHS $P_{n+1}^h = window(P_n^h, \{sp_0\}, \texttt{win}_{sp}, 2)$ that correponds to the same part of P_n^h depicted in Part (a). States sp_i^j, $0 \le i \le 2$, $0 \le j < 2$, for which $(\texttt{win}_{sp}, j) \in \beta_n^h(sp_i^j)$, are highlighted in red. Please note that each global transition from the orignal EHS P_n^h corresponds to a family of transitions given by all possible injections $\Gamma_1, \ldots, \Gamma_k \colon Q_n^h \to \{\bot, 0, 1, \top\}$ with the mappings $\widehat{\Gamma_i}$, $1 \le i \le k$, defined such that $\forall Q' \subseteq Q_n^h \colon \widehat{\Gamma_i}(Q') := \{(q, \Gamma_i(q)) \mid q \in Q'\}$. These families of transactions are denoted by the dashed and dotted lines in the figure. In particular, the dashed lines are used for the $\exists_{\leftarrow} : \widehat{\Gamma_i}(G_{st})$ family. The dotted lines then correspond with the $\exists_{\leftarrow} : \widehat{\Gamma_i}(G_{cl})$ family.

Fig. 4: An illustration of an application of the $window$ procedure on a fragment of an EHS P_n^h.

What remains to be done is to adjust the tracking window of the earlier instruction, which has to be done such that the extension corresponds to the number of the slack transitions taken by the later instruction. For that, we will again use the $window$ procedure from Alg. 2, but we will instruct it to add special tags of the form $\texttt{win}_{vi/sp}^{(i)}$ meaning that the tracking window of the earlier instruction is extended by i cycles. The definition of the bad configurations will then

Algorithm 3 A procedure for computing the *slack* mapping.

Require: An EHS $P^h_n = (Q^h_n, \Delta^h_n, \alpha^h_n, \beta^h_n)$ of any level $n \geq 0$ and the total number of pipeline stages $m \geq 1$.

Ensure: An EHS $P^h_{n+1} = (Q^h_{n+1}, \Delta^h_{n+1}, \alpha^h_{n+1}, \beta^h_{n+1})$ whose states P^h_{n+1} are tagged by pairs $(\mathtt{sl}, i)$ where $0 \leq i < m$ denotes the number of self-loop transitions taken by the later tracked instruction in the EHS P^h_n.

1: $I := \{\top, 0, \ldots, m-1\}$.

2: $Q^h_{n+1} := Q^h_n \times I$.

3: Δ^h_{n+1} is defined as the minimal relation such that the following two conditions hold:

(a) For every global transition $\mathbb{Q}_\circ \colon G \models q_1 \to q_2 \in \Delta^h_n$ and for every injection $\Gamma : Q^h_n \to I$, the following transitions are in Δ^h_{n+1}:

- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, i+1)$ if $q_1 = q_2$ for all $0 \leq i < m-1$,
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, i)$ if $q_1 \neq q_2$ for all $0 \leq i < m$,
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, i) \to (q_2, \top)$ if $q_1 = q_2$ and $i = m-1$,
- $\mathbb{Q}_\circ \colon \widehat{\Gamma}(G) \models (q_1, \top) \to (q_2, \top)$

where $\widehat{\Gamma} \colon 2^{Q^h_n} \to 2^{Q^h_{n+1}}$ is defined such that $\forall Q' \subseteq Q^h_n \colon \widehat{\Gamma}(Q') := \{(q, \Gamma(q)) \mid q \in Q'\}$.

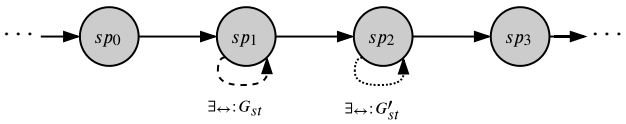
(b) For every local transition $q_1 \to q_2 \in \Delta^h_n$, the following transitions are in Δ^h_{n+1}:

- $(q_1, i) \to (q_2, i+1)$ if $q_1 = q_2$ for all $0 \leq i < m-1$,
- $(q_1, i) \to (q_2, i)$ if $q_1 \neq q_2$ for all $0 \leq i < m$,
- $(q_1, i) \to (q_2, \top)$ if $q_1 = q_2$ and $i = m-1$,
- $(q_1, \top) \to (q_2, \top)$.

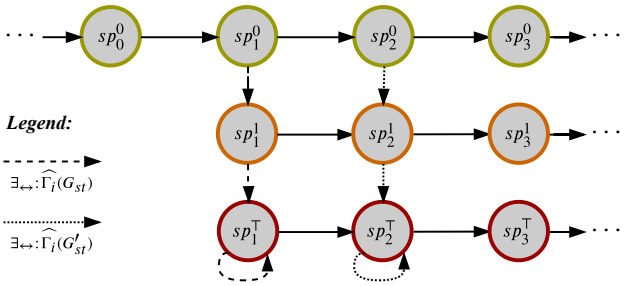
4: $\forall (q, i) \in Q^h_n \times I : \alpha^h_{n+1}(q, i) = \alpha^h_n(q)$.

5: $\forall (q, i) \in Q^h_n \times \{\top\} : \beta^h_{n+1}(q, i) = \beta^h_n(q)$.

6: $\forall (q, i) \in Q^h_n \times \{0, \ldots, m-1\} : \beta^h_{n+1}((q, i)) = \beta^h_n(q) \cup \{(\mathtt{sl}, i)\}$.



(a) A part of an EHS P^h_n modeling the behavior of a spoiler instruction before an application of the *slack* mapping. Note that the spoiler instruction in the states sp_1 and sp_2 might be stalled (if there are instructions from the set G_{st}, resp. G'_{st}). The different styles of the lines representing global transitions are used to allow for better matching with the counterparts of the transitions in Fig. 5 (b).



(b) A part of an EHS $P^h_{n+1} = slack(P^h_n, 2)$ that correponds to the same part of P^h_n depicted in Part (a). States sp^j_i, $0 \leq i \leq 3$, $0 \leq j < 2$, for which $(\mathtt{sl}, j) \in \beta^h_n(sp^j_i)$ with the same value of j, indicating that the instructions passed the same number of self-loops, share the same color. Please note that each global transition from the orignal EHS P^h_n corresponds to a family of transitions given by all possible injections $\Gamma_1, \ldots, \Gamma_k \colon Q^h_n \to \{\bot, 0, 1, \top\}$ with the mappings $\widehat{\Gamma}_i$, $1 \leq i \leq k$, defined such that $\forall Q' \subseteq Q^h_n \colon \widehat{\Gamma}_i(Q') := \{(q, \Gamma_i(q)) \mid q \in Q'\}$. Analogically to Fig. 4, these families of transactions are denoted by the dashed and dotted lines in the figure—the dashed lines are used for the $\exists_{\leftarrow} \colon \widehat{\Gamma_i}(G_{st})$ family while the dotted lines correspond with the $\exists_{\leftarrow} : \widehat{\Gamma_i}(G'_{st})$ family.

Fig. 5: An illustration of an application of the *slack* mapping on a fragment of an EHS P^h_n.

match states of the earlier instruction tagged by $\mathtt{win}^{(i)}_{vi/sp}$ with $\mathtt{win}_{sp/vi}$-tagged states of the later instruction that are at the same time tagged by such $\mathtt{sl}$ tags which show that the later instruction went through i slack transitions more than the earlier one.

With all the notation at hand, it is now easy to derive the EHSs P^h_{3+i} of levels $3+i$ for $1 \leq i \leq m$ with $m = \max(\mathbb{S})$ being the maximum number of pipeline stages that extend the tracking window of the earlier instruction by i cycles. For i iterating from 1 to m, we get

$$P^h_{3+i} := window\Big(P^h_{3+i-1}, wr_{P^h_{3+i-1}}(\kappa, \chi_\kappa), \mathtt{win}^{(i)}_\kappa, g_\kappa + d_\kappa + i\Big)$$

where $\kappa = \mathcal{E}(h)$. As the final step, we put $P^h_\top := P^h_{3+m}$.

Initial and Bad Configurations

Above, we have finished the construction of the EHS $P^h_\top$ designed to facilitate the construction of the set $B^h_\top$ of minimal bad configurations describing minimal illegal configurations whose reachability (within possibly larger configurations) will mean that the given hazard case h does indeed lead to a hazard. It now remains to define the set $B^h_\top$ along with the corresponding set of initial configurations between which reachability will have to be checked.

We first define the regular set $I^h_\top$ of initial configurations of $P^h_\top$ that consists solely of instructions in the state $\bot$, i.e., before entering the pipeline. An initial configuration may be of an arbitrary length, and it may contain exactly one spoiler sp and one victim instruction vi, interleaved by any other instructions in any order, modelled using the nf class. Formally, the set $I^h_\top$ of the initial states of EHS $P^h_\top$ is defined as follows

$$I^h_\top := I^h_1 \cup I^h_2$$

where

$$I^h_1 := \{\langle nf, \bot\rangle\}^*\{\langle vi, \bot\rangle\}\{\langle nf, \bot\rangle\}^*\{\langle sp, \bot\rangle\}\{\langle nf, \bot\rangle\}^*$$

and

$$I^h_2 := \{\langle nf, \bot\rangle\}^*\{\langle sp, \bot\rangle\}\{\langle nf, \bot\rangle\}^*\{\langle vi, \bot\rangle\}\{\langle nf, \bot\rangle\}^*.$$

Next, we define the set $B^h_\top$ of minimal bad configurations that describe hazardous configurations. The main challenge behind the construction of $B^h_\top$ is to correctly match detection states of the earlier and later instructions. For that, we will use the tracking mechanism that we have provided by the $\texttt{win}_{sp/vi}$ tags. Namely, we will construct $B^h_\top$ to include all configurations that contain pairs of states $q_e, q_\ell \in Q^h_\top$, consisting of a state q_e of an e-class instruction and a state q_ℓ of an ℓ-class instruction, for which the win tags correspond to the detection part of the tracking window. This is, we will consider the states $q_e = \langle \kappa^h_e, \ldots \rangle \in Q^h_\top$ obeying

$$\beta^h_\top(q_e) \in \{(\texttt{win}_{\kappa^h_e}, i) \mid g_{\kappa^h_e} \leq i < g_{\kappa^h_e} + d_{\kappa^h_e}\},$$

for $\kappa^h_e = \mathcal{E}(h)$, and, dually, the states $q_\ell = \langle \kappa^h_\ell, \ldots \rangle \in Q^h_\top$ satisfying

$$\beta^h_\top(q_\ell) \in \{(\texttt{win}_{\kappa^h_\ell}, i) \mid g_{\kappa^h_\ell} \leq i < g_{\kappa^h_\ell} + d_{\kappa^h_\ell}\},$$

for $\kappa^h_\ell = \mathcal{L}(h)$.

It now remains to deal with situations when some of the instructions are stalled. This is monitored using the sl tags. First, we can observe that we do not have to further elaborate cases when both (earlier and later) instructions are stalled together. Clearly, any hazard that would occur after these cases would also occur in the case when the instructions are not stalled. Second, the case when the earlier instruction is stalled while the later is not is excluded by the consistency of the pipeline. Therefore, it suffices to only consider those states of the earlier instruction q_e for which $(\texttt{sl}, 0) \in \beta(q_e)$. Next, let i be a counter that increases each time the later instruction is stalled while the earlier one is not. Since the consistency Rules 1–4 from Section 5.2 guarantee that each instruction leaves the pipeline in a final number of steps, the value of the counter i may only range from 0 to $\max(\mathbb{S})$. Every time the counter i is increased, the detection in the earlier instruction is postponed by a single pipeline cycle.

Taken all together, the set $B^h_\top$ of minimal bad configurations describing hazardous configurations is defined as

$$B^h_\top := \bigcup_{i=0}^{\max(\mathbb{S})} B^h_i \tag{43}$$

where

$$\begin{aligned} B^h_i := \{q_e q_\ell \mid\ & \{(\texttt{sl}, 0), (\texttt{win}^{(i)}_{\kappa^h_e}, i + j)\} \subseteq \beta^h_\top(q_e) \wedge \\ & \{(\texttt{sl}, i), (\texttt{win}_{\kappa^h_\ell}, k)\} \subseteq \beta^h_\top(q_\ell) \wedge \\ & g_{\kappa^h_e} \leq j < g_{\kappa^h_e} + d_{\kappa^h_e} \wedge \\ & g_{\kappa^h_\ell} \leq k < g_{\kappa^h_\ell} + d_{\kappa^h_\ell} \wedge \\ & q_e = \langle \kappa^h_e, \ldots \rangle \in Q^h_\top \wedge \kappa^h_e = \mathcal{E}(h) \wedge \\ & q_\ell = \langle \kappa^h_\ell, \ldots \rangle \in Q^h_\top \wedge \kappa^h_\ell = \mathcal{L}(h)\}. \end{aligned} \tag{44}$$

With the EHS $P^h_\top$ and the sets of initial $I^h_\top$ and minimal bad configurations $B^h_\top$ at hand, checking whether the hazard h is feasible reduces to checking whether there is some configuration in $B^h_\top$ that is reachable from some configuration in $I^h_\top$, for which one can use techniques described, e.g., in [2,4].

8 Experimental Evaluation

We have implemented the above described method in a prototype tool called Hades [10]. Hades is written in C++ combined with Python and consists of several components depicted in Fig. 6. The tool first reads an RTL description of the processor to be verified and converts it into its internal PSG representation. Currently, Hades supports the RTL format expressed in CodAL which is an architectural description language used in the processor design IDE [15]. For other RTL languages like VHDL and Verilog where architectural registers are not explicitly identified, a list of architectural registers with an explicit identification of the program counter must be provided.

The obtained PSG representation is then normalised and simplified. This step includes, for instance, a replacement of conditional branching by multiplexors, an application of value propagation, and a removal of redundant nodes and edges. The normalisation is done using an internal component of Hades called as the *RTL query engine* (RQE), which allows one to search for data-paths and substitute parts of the microprocessor RTL design described via a PSG. Subsequently, pipeline stages are identified by the data-flow analysis discussed in Section 5.1. Next, pipeline consistency is checked using Rules 1–4 from Section 5.2 by an SMT solver for bit-vector logic. Hades is compatible with all SMT solvers accepting the SMT2 formula format. In particular, for the below experiments, recent versions of Z3 (v4.8.8) [21] and CVC4 (v1.9) [3] were used. Further, after the PSG is annotated by pipeline stages identified by the data-flow analysis, Hades repeatedly utilizes the RQE and the SMT solver to extract potential hazard cases as described in Section 6 and to generate the appropriate hazard systems (HSs) for each hazard case as we have seen in Section 7. The generated HSs are then checked using the abstract regular model checker (ARMC) of [4]. The process of evaluation of the inputs and generation of the results by the above mentioned subsystems is orchestrated by the so-called "core" component of Hades.

We have tested the tool on the following processors[14]: *TinyCPU* is a small 8-bit processor with 3 pipeline stages that we use mainly for testing new verification methods.

[14] Most of the processors are available together with our tool. However, *SPP8* and *Codea2* were provided by the Codasip company [15] and are not publicly available at the time of writing this article—likewise for *SPP16*, which we derived from *SPP8*.

Table 3: Experimental results obtained using the Z3 SMT solver on processors whose names are given in the table.

Name (Stages) Variant	Simpl. Time [s]	Data Flow Analysis [s]	Consistency Checking [s]			Parametric System Generation and Verification [s]				Total Time [s]	Hazard Cases [#]	
			rqe	smt	core	rqe	smt	armc	core		pot.	real
TinyCPU (3)												
SCR	0.02	0.01	<0.01	0.54	0.30	0.01	1.11	12.23	2.29	16.51	6	0
SACRWV	0.03	0.01	<0.01	0.63	0.36	0.03	3.29	32.36	6.35	43.06	10	1
BCR	0.01	0.01	<0.01	0.50	0.26	0.01	1.09	10.20	2.54	14.62	6	0
BACRWV	0.01	0.01	<0.01	0.66	0.37	0.03	3.48	32.35	8.07	44.98	10	1
SFCR	0.02	0.01	<0.01	0.51	0.32	0.02	2.52	31.16	4.70	39.26	14	0
SFACRWV	0.02	0.01	<0.01	0.68	0.46	0.05	6.50	81.16	13.27	102.15	22	1
SPP (3)												
SCR	0.07	0.01	0.01	0.83	0.59	0.04	5.24	32.67	9.73	49.19	29	0
BCR	0.06	0.01	0.01	0.82	0.60	0.04	5.72	32.68	14.16	54.10	29	0
SPP16 (3)												
SCR	0.07	0.01	0.01	0.94	0.63	0.04	5.51	32.39	9.93	49.53	29	0
BCR	0.07	0.01	0.01	0.90	0.64	0.04	6.03	32.86	14.53	55.09	29	0
Codea2 (4)												
SFCR	0.19	0.04	0.01	1.19	0.67	0.33	74.96	247.17	128.48	453.04	243	4
CompAcc (4)												
SFACRWV	0.06	0.01	0.01	1.14	0.72	0.13	28.36	248.78	36.52	315.73	44	0
BFACRWV	0.07	0.02	0.01	1.14	0.67	0.16	34.97	249.45	49.99	336.48	59	0
DLX5 (5)												
SFCR	0.10	0.03	0.01	2.18	1.32	0.11	24.03	189.67	42.58	260.03	27	1
SFACRWV	0.10	0.03	0.01	2.36	1.42	0.20	46.82	292.63	81.04	424.61	62	2
BFCR	0.10	0.04	0.01	2.29	1.31	0.13	57.79	190.03	143.21	394.91	27	1
BFACRWV	0.12	0.02	0.01	2.34	1.43	0.23	91.20	292.95	361.14	749.44	62	2

S	Stalling Logic	B	Bypassing Logic	F	Flag Register(s)	A	Auto-increment Logic	
C	CTL Hazards	R	RAW Hazards	W	WAR Hazards	V	WAW Hazards	

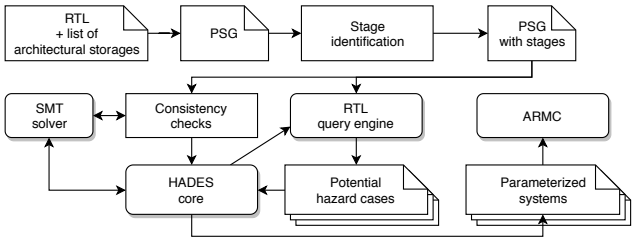


Fig. 6: A schematic of the Hades verification tool.

SPP8 is a 3-staged 8-bit ipcore with 16 general-purpose registers and a RISC instruction set consisting of 9 instructions. *SPP16* is a 16-bit variant of the previous processor with a more complex memory model (allowing, for instance, unaligned access). *Codea2* is a 16-bit processor with 4 pipeline stages dedicated for signal processing applications. The processor is equipped with 16 general-purpose registers, 15 special registers, a flag register, and an instruction set including 41 instructions where each may use up to 4 available addressing modes. *CompAcc* is an 8-bit processor with 4 stages that is based on an accumulator architecture with a very similar structure as the one shown in Fig. 2. Finally, *DLX5* is a 5-staged 32-bit processor able to execute a subset of the instruction set of the DLX architecture [24] (with no floating point instructions).

We consider multiple variants of the above introduced processors, which gives us 17 unique test cases in total. In particular, the variants of the particular processors differ in the following aspects: (i) the way how data hazards are avoided (pipeline stalling and clearing or data bypassing), (ii) the presence of flag/status registers, and (iii) utilization of the auto-increment logic.

We conducted a series of experiments on a PC with Intel Core i7-3770K @3.50GHz and 32 GB RAM whose results

are shown in Table 3. In these experiments, we used Hades backed by the Z3 solver. The first column gives the name of the verified processor and its variant. The specifics of each variant typically influence the types of potential hazards that may occur. Therefore, we have also encoded the hazard types into the variant's name (e.g., "C" for "contains potential control hazards"—we give a full list of the codes used below the table). The second and third columns give the time needed for the PSG simplification and its data flow analysis. The fourth and fifth columns (both split to a number of sub-columns) then give the duration of the consistency checking and the time spent by verification of the parametric systems that are created for each hazard case. In the sub-columns, the times given in the fourth and fifth columns are split to the times consumed by the different parts of our tool's architecture.

The sixth column provides the overall verification time, which remains in the order of minutes even for complex designs. Moreover, the tool also scales well with the growing width of the processor data-path as can be seen by comparing the times obtained for *SPP8* and *SPP16*.

From the obtained data, we further see that the verification time grows with the number of stages of the processor. Naturally, such an increase comes from the fact that, in CPUs with deeper pipelines, longer spoiler/victim executions are likely to occur. We have not analysed this growth from a complexity-theoretic point of view, but our experiments show that the average verification time required for verification of a single hazard case approximately doubles with each additional stage—it is 2.86 sec. for the considered processors with 3 pipeline stages, 4.91 sec. for the 4-staged ones, and 10.8 sec. for the 5-staged ones[15].

Since the current Hades implementation relies on exporting formulas in the `smt2` file format, the tool does not depend on any particular SMT solver. We took this opportunity and conducted the same set of experiments with the CVC4 solver too where we observed a similar pattern in which the average verification time per hazard case doubles with each added stage. We have also noticed that for certain tested designs (especially the less complex ones like *TinyCPU*) the CVC4 solver delivers results approximately 5-10 % faster than Z3. Therefore, utilization of a portfolio solver, which would run multiple SMT solvers in parallel and report the answer that is first returned by any one of them, should lead to even better verification times. However, we do not consider extensive benchmarking with SMT solvers a goal of this article.

[15] The considered processors differ from each other not only in the length of the pipeline but also in the overall structural complexity. However, the overall structural complexity is bigger for those processors that have a longer pipeline. Hence, the influence of solely the length of the pipeline can be even smaller than two-fold.

Finally, the two sub-columns of the last column give (i) the number of potential data and control hazard cases that had to be checked and (ii) the number of cases that were proven to be *real*, respectively. Note that each hazard case represents a separate task, and so the generation and verification of the parametric systems can be parallelized in the future. Even though the number of hazard cases is higher when compared to our previous work [12] (because of control hazards), the runtimes (in Table 3) have improved for most of the columns. A majority of this improvement is caused by an implementation of a mechanism that efficiently pre-computes and reuses the formulae F_1, F_2, and F_3 from Eq. 12. A drawback of this approach (compared to the one used in [12]) is a higher utilization of the SMT solver, which is, however, compensated by the lower verification time consumed by the Hades core component.

During the experiments, we identified a flaw in the RAW hazard resolution when accessing the data memory in a development version of the *SPP8* processor. Further, compared to our previous results from [12], we have extended our *TinyCPU* and *DLX5* processor models so that they can now handle a majority of control hazards. The different number of potential hazard cases is a side-effect of this change. All the remaining real hazard cases (shown in the last column) are control hazards.

Notice that each processor with the auto-increment logic, which typically resides in early pipeline stages, contains at least one instance of a control hazard. The auto-increment logic is a feature of addressing modes where a register used for computing a memory address is incremented once its value is obtained. Such an increment is not guarded even if the instruction itself will be discarded in later stages, e.g., due to an unsuccessful speculative execution. This is not considered a design error; instead, software compilers are instructed not to place instructions with an auto-increment near jumping instructions (they resolve the problem by explicitly generating a series of `NOP` instructions after a conditional branch). A similar situation occurs in the *DLX* processor where the fourth pipeline stage, designated for memory accesses, writes to memory whenever a store instruction is executed even if the branch condition of an earlier instruction is not yet evaluated (therefore the memory access cannot be prevented). This situation adds an extra occurrence of a control hazard and must also be taken into account when compiling a program. Finally, in *Codea2*, the logic that deals with control hazards is left out intentionally (by design), again relying on the compiler to take this into account.

9 Conclusion

We have presented an approach that harnesses methods for formal verification of parametric systems in order to discover incorrectly handled data and control pipeline hazards

in the RTL implementation of pipeline-based execution. The approach was developed with the aim to be highly automated, not requiring any additional efforts from the developers (apart from specifying the architectural registers). We have implemented the approach and successfully tested it on several non-trivial microprocessors where the approach was able to discover previously unknown flaws caused by unhandled hazards.

Acknowledgements This work was supported by the Czech Science Foundation under the project 20-07487S.

References

1. M. D. Aagaard. A hazards-based correctness statement for pipelined circuits. In *Proc. of Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 66–80. Springer, 2003.
2. P. A. Abdulla, F. Haziza., and L. Holík. All for the price of few (parameterized verification through view abstraction). In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, volume 7737 of *LNCS*, pages 476–495. Springer, 2013.
3. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
4. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 197–202. Springer, 2004.
5. R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.
6. Randal E. Bryant. Formal verification of pipelined Y86-64 microprocessors with UCLID5. Technical Report CMU-CS-18-122, 2018.
7. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
8. Cadence. *Tensilica Software Development Toolkit (SDK)*, 2014.
9. L. Charvát, A. Smrčka, and T. Vojnar. Automatic formal correspondence checking of ISA and RTL microprocessor description. In *Proc. of Microprocessor Test and Verification (MTV'12)*, pages 6–12. IEEE, 2012.
10. L. Charvát, A. Smrčka, and T. Vojnar. HADES Hades Hardware Verification Tool. `www.fit.vutbr.cz/research/groups/verifit/tools/hades/`, 2014.
11. L. Charvát, A. Smrčka, and T. Vojnar. Using formal verification of parameterized systems in RAW hazard analysis in microprocessors. In *Proc. of Microprocessor Test and Verification (MTV'14)*, pages 83–89. IEEE, 2014.
12. L. Charvát, A. Smrčka, and T. Vojnar. HADES: Microprocessor hazard analysis via formal verification of parameterized systems. In *Proc. of 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'16)*, 233, pages 87–93. EPTCS, 2016.
13. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 126–141. Springer, 2006.
14. CodAL architecture description language. `www.codasip.com/custom-processor`, 2019.
15. Codasip Studio for rapid processor development. `www.codasip.com`, 2019.
16. K. Hao, S. Ray, and F. Xie. Equivalence checking for function pipelining in behavioral synthesis. In *Proc. of Design, Automation and Test in Europe (DATE'14)*, pages 1–6. IEEE, 2014.
17. R. B. Jones, C. H. Seger, and D. L. Dill. Self-consistency checking. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 159–171. Springer, 1996.
18. A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to RTL equivalence checking. In *Proc. of Design, Automation and Test in Europe (DATE'09)*, pages 196–201. IEEE, 2009.
19. U. Kuhne, S. Beyer, J. Bormann, and J. Barstow. Automated formal verification of processors based on architectural models. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD'10)*, pages 129–136. IEEE, 2010.
20. P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proc. of Design, Automation and Test in Europe (DATE'02)*, pages 36–43. IEEE, 2002.
21. L. De Moura and N. Bjorner. Z3: An efficient SMT solver. In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
22. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, volume 4349 of *LNCS*, pages 299–313. Springer, 2007.
23. M. Ngyuen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz. Unbounded protocol compliance verification usign interval property checking with invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 27(11), 2008.
24. D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann, Boston, fourth edition, 2012.
25. J. Van Praet, D. Lanneer, W. Geurts, and G. Goossens. *nML: A Structural Processor Modeling Language for Retargetable Compilation and ASIP Design*, volume 1 of *Systems on Silicon*, pages 65–93. Morgan Kaufmann, Burlington, 2008.
26. Synopsys. *ASIP Designer: Design Tool for Application Specific Instruction-Set Processors, Designer Datasheet*, 2018.
27. M. N. Velev and P. Gao. Automatic formal verification of multithreaded pipelined microprocessors. In *Proc. of International Conference on Computer Aided Design (ICCAD'11)*, pages 679–686. IEEE, 2011.