WCET Analysis of HLS-Generated Processors Using Abstract Interpretation

Thomas Feuilletin
Univ Rennes, Inria, CNRS, IRISA, France

Dylan Leothaud
Univ Rennes, Inria, CNRS, IRISA, France

Simon Rokicki
Univ Rennes, Inria, CNRS, IRISA, France

Steven Derrien
UBO, Lab-STICC, Brest, France

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA, France

Abstract

Deriving sound and precise timing models remains one of the main obstacles to static Worst-Case Execution Time (WCET) analysis. Modern processors exhibit diverse and evolving microarchitectures, making manual construction of timing models labor-intensive, error-prone, and difficult to adapt across processor variants. High-Level Synthesis (HLS) enables rapid customization of processor cores and architectural exploration, offering an opportunity to automate not only hardware generation but also the derivation of associated timing models.

This paper presents an automated WCET analysis for HLS-generated processors based on abstract interpretation. We exploit the internal Gated-SSA representation of the HLS flow to automatically extract an abstract timing model capturing speculation and stall mechanisms. WCET estimation at the basic block level is then formulated as an exploration of abstract microarchitectural states within a basic block.

The approach safely accounts for timing anomalies, while remaining scalable thanks to an efficient state-merging strategy. Integrated into the Heptane WCET tool and evaluated on Mälardalen benchmarks and a RISC-V, the method achieves the same tightness as a handcrafted timing model, while improving over a previously proposed automated approach.

2012 ACM Subject Classification Hardware → Static timing analysis

Keywords and phrases Static Analysis, Worst-Case Execution Time, High-Level Synthesis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.14

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.20713492`

Funding *Thomas Feuilletin*: ANR LOTR – `https://lotr.gitlabpages.inria.fr/website/.`

1 Introduction

Custom processor cores are widely used in embedded systems to improve performance, energy efficiency, and reliability. In time-critical domains, strict temporal guarantees must also be provided, typically through Worst-Case Execution Time (WCET) analysis [21].

WCET estimation requires accurate microarchitectural timing models. However, such models differ substantially across processor implementations, even when they share the same Instruction Set Architecture (ISA). This diversity is particularly pronounced in the RISC-V ecosystem, where numerous core variants coexist. In practice, timing models are often constructed manually: experts interpret documentation – or inspect RTL when available – and derive simplified abstractions. This process is time-consuming, error-prone, and difficult to adapt across processor variants [12]. Although extracting timing information directly

from RTL may seem attractive, the low abstraction level and structural complexity of Hardware Description Languages (HDLs) make this approach challenging and, so far, not fully automated.

At the same time, processor design methodologies are evolving. Traditional RTL-centric flows struggle to keep pace with the rapid evolution of processors such as RISC-V, characterized by an expanding set of ISA extensions and increasingly stringent requirements for security, predictability, and resilience. High-Level Synthesis (HLS) offers an alternative design paradigm. In HLS flows, processor behavior is described in high-level languages such as C or C++, from which hardware implementations are automatically generated. This enables rapid exploration of architectural trade-offs and facilitates the integration of custom microarchitectural features.

However, conventional commercial HLS tools are not well suited to synthesizing processor cores directly from behavioral Instruction-Set Simulator (ISS) descriptions, typically yielding inefficient implementations [9]. Recent research has addressed these limitations by introducing dynamic and speculative execution mechanisms into HLS frameworks [7, 14]. In particular, speculative loop pipelining [10] enables the synthesis of efficient pipelined processor cores from high-level descriptions by inferring an explicit microarchitectural model [9].

In this work, we leverage these inferred microarchitectural representations to automatically derive sound and tight WCET timing models for simple yet relevant pipelined cores. We show that timing-model construction can be integrated directly into an HLS-based processor design flow, enabling precise and scalable WCET analysis. We make the following contributions:

- A precise yet scalable timing model and associated WCET analysis for RISC-V cores. The timing model is automatically extracted from the internal representation of an HLS flow. Compared to previous work [8], the WCET analysis accounts for timing anomalies using an efficient state-exploration algorithm.
- Its implementation and experimental validation for a RISC-V core within the Heptane WCET analysis tool [13], demonstrating substantial improvements in tightness compared to the analysis in [8].

The remainder of this paper is organized as follows. Section 2 provides background information on WCET analysis techniques and HLS-based processor synthesis approaches. Section 3 presents the proposed timing model and its extraction from an HLS framework. Section 4 describes the corresponding timing analysis technique. Section 5 presents and discusses the experimental results. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Background

2.1 Worst Case Execution Time analysis

Worst-Case Execution Time (WCET) estimation aims at deriving safe upper bounds on program execution time [21]. For the scope of this paper, we focus on static WCET estimation techniques, which compute safe execution-time bounds without executing the program on the target hardware.

Static WCET analysis typically involves two main steps. First, *low-level analysis* estimates the execution time of small code fragments, known as *basic blocks* (sequences of instructions with no internal branching except possibly at the last instruction), using an abstract timing model of the processor. Such timing models capture relevant architectural effects, including pipeline behavior, cache effects, and branch prediction, while remaining conservative with

respect to actual hardware timing. Second, *high-level path analysis* identifies the longest feasible execution path in the program's Control-Flow Graph (CFG), which represents possible control-flow transitions between basic blocks. A commonly used method is the Implicit Path Enumeration Technique (IPET [15]), which formulates WCET computation as an integer linear programming (ILP) optimization problem.

Designing accurate yet safe timing models for static WCET analysis raises several issues:

- **Availability of timing information**. An abstract timing model must remain consistent with actual hardware behavior, which requires identifying the architectural features relevant to timing analysis and ensuring that sufficient timing information is available. However, detailed timing characteristics of hardware components are not always publicly documented or easily accessible. Therefore, model construction must balance fidelity to hardware behavior with the availability of exploitable timing data.
- **State-space explosion.** Modern processors exhibit a large number of possible concrete microarchitectural states (pipeline, cache, branch predictor, etc.). Enumerating all concrete states is intractable. This motivates the use of abstraction techniques such as *abstract interpretation* to compactly represent state sets while preserving safety guarantees [1].
- **Composability of timing analyses**. Timing effects of different hardware components are often interdependent. Designing composable analyses that allow separate modeling of architectural features while preserving global safety is challenging, particularly in the presence of timing anomalies [16, 4], where locally worst-case microarchitectural events may not correspond to global worst-case execution times. When timing anomalies may occur, exploring hardware states is necessary to obtain sound WCET bounds.

This paper addresses these three challenges by automatically extracting abstract timing models for High-Level Synthesis (HLS)-designed processors. Our focus is on low-level analysis, while standard high-level WCET analysis techniques are assumed to be available at the program level.

2.2 Generating processors with High-Level Synthesis (HLS)

High-Level Synthesis (HLS) is a design methodology that analyzes a high-level behavioral description of an algorithm and generates a hardware component that implements this behavior for a given target (e.g., an FPGA or an ASIC with a given technology). The primary objective of HLS tools is to transform an untimed description into a scheduled implementation, assigning each operation to a precise clock cycle. Several scheduling techniques exist, among which loop pipelining is one of the most effective. Loop pipelining overlaps successive iterations of a loop by initiating a new iteration before the previous one has completed. The *Initiation Interval* (II) of a pipelined schedule (ideally 1) is the number of cycles to wait before starting a new iteration. As loop pipelining is a static scheduling technique, all possible dependencies must be considered when scheduling operations. Once generated, the resulting cycle duration depends on the target. However, the schedule, which takes the target specifications into account, remains the one determined by the HLS tool.

The synthesis of a pipelined processor using HLS takes as input an Instruction Set Simulator (ISS) and generates the processor hardware. The ISS, illustrated in the left part of Figure 1, is an infinite loop operating on the processor's registers (including the Program Counter – PC). Using commercial HLS tools to synthesize a pipelined processor yields poor performance because the exact value of the next program counter (PC) must be determined before executing the next instruction. Consequently, II cannot be equal to the ideal value of 1, because one must wait for the outcome of a conditional branch before knowing the next value of PC.

Recent advances in HLS have introduced *Speculative Loop Pipelining* (SLP) [7]. This technique speculates on the values produced by an iteration in order to start subsequent iterations as early as possible. SLP is built upon the Gated-SSA [19, 20] representation, a refined variation of Single Static Assignment (SSA) Intermediate Representation (IR) used in compiler frameworks. In SSA, each variable is assigned exactly once, simplifying data-flow analysis; when multiple control-flow paths merge, SSA introduces $\phi - nodes$ to select the appropriate variable value depending on the execution path taken. In Gated-SSA, $\phi - nodes$ are represented as multiplexer-like nodes called $\mu - node$ and $\gamma - node$. The former represents loop headers (i.e., the choice between the variable's value before the loop's start and the variable's value from the loop body), whereas the latter represents other reconvergence points in the control-flow. SLP consists of identifying $\gamma - nodes$ with a fast and a slow path, in order to speculate that the fast path is taken. The intermediate representation is then transformed by inserting penalty delays on slow paths (the number of cycles due to pipeline stalls), which increases the reuse distance and improves the achievable minimal II. To preserve the original program semantics, a Finite State Machine (FSM) is inserted to control the speculation logic. The FSM drives the $\gamma - nodes$, verifies the effective value of the control signal, and, depending on the validity of the speculation, triggers some *rollback* or *commit* signal.

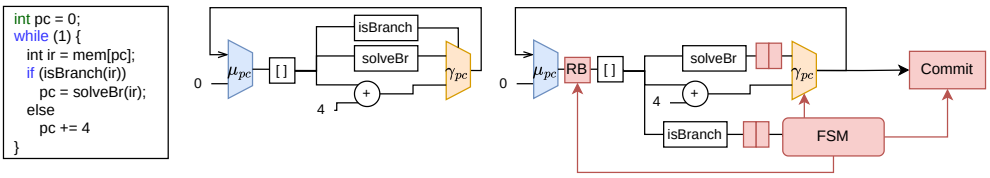

Figure 1 Example of a simple processor synthesized using speculative loop pipelining. The leftmost part represents the source code, the central part is the corresponding Gated-SSA IR, and the rightmost part is the transformed IR, with additional delays and control logic (in red).

Figure 1 illustrates a simple speculation scenario in which the next program counter is either PC+4 or the target of a branch instruction starting from an inital pc equal to 0. In this example, we assume that three clock cycles are required to determine if an instruction branches and, if so, at which target address. The Gated-SSA representation of this example is depicted in the central part of Figure 1. To achieve a pipeline with II=1, the SLP technique speculates that the next PC value is PC+4 and generates the right-most representation. We can see the FSM driving the different speculation mechanisms, and the delays inserted on slow paths (delays of 2 clock cycles compared to a perfect pipeline issuing one instruction per cycle).

Speculative loop pipelining is a key mechanism for generating pipelined processors using HLS. In a conventional pipelined processor, two fundamental speculations are performed: first, that the next instruction address is equal to PC+4 (see Figure 1), and second, that no read-after-write (RAW) dependency exists between consecutive instructions. These assumptions correspond directly to the forms of speculation enabled by speculative loop pipelining.

A notable advantage of using HLS to generate pipelined processors is the ability to explicitly model speculation and precisely identify the sources of pipeline penalty. In the example from Figure 1, the value sent to the FSM directly controls the number of pipeline penalty (delays) and can be used to estimate the WCET of sequences of instructions.

3 Timing Model: Description and Extraction

Our analysis, presented in Section 4, explores the set of microarchitectural states reachable during the execution of a Basic Block (BB) to identify the sequence of states with the longest execution time. This section defines what the *state* of the micro-architecture handled by the proposed analysis is and how it is extracted from the internal Gated-SSA representation of the HLS framework. As a running example, a simple two-instruction core is used to illustrate the main concepts. Readers who are not familiar with HLS flows and the associated data structures may focus on the model definition and skip the paragraphs dedicated to model extraction without losing essential information.

3.1 Running Example

Let us consider a simple core with two instructions *add* and *mul*, whose Gated-SSA representation is depicted in Figure 2. This example has two speculation points:

- The first speculation point (γ_{regs}-node) speculates on the absence of read-after-write (RAW) dependencies (an instruction reads a value written by a previous instruction) on the registers file. If a dependency (also known as an alias) exists, a penalty delay of one or two cycles (depicted in red) is added, as explained later.
- The second speculation point (γ_{res}-node) speculates that the result of the shortest instruction (addition) is used. In case of misspeculation, a two-cycle penalty is incurred, as detailed later.

The program counter *pc* fetches an instruction from external memory ([ ] operation after the μ_{pc}-node). An instruction is coded as a 5-bit value with: 1 bit for the opcode (0 for an addition, 1 for the multiplication), two bits for the source register `rs`, and two bits for the destination register `rd`. The `extract` block splits the instruction into its three parts.

The control part below the `extract` block checks for a dependency (alias) between a register write `rd` from the two previous cycles, and a register read `rs` in the current iteration. The two δ-nodes keep the values of `rd` of the two previous cycles. Their purpose is not to stall the pipeline but to carry over values from the previous cycle to the current one. At the beginning of an iteration, δ_1 saves the value of `rd` of the current iteration, and forwards the value saved in the previous cycle, where δ_2 saves the value forwarded by δ_1 and forwards its current value. The values of δ_1 and δ_2 are compared with the current `rs`.

The result of the comparison controls a set of multiplexers that controls the first speculation point (γ_{regs}) in our running example. If the alias distance is 0, (i.e., current iteration aliases with the last cycle), the core goes through 2 penalty cycles (represented by the two red rectangles in Figure 2) to wait for the write operation to complete; if the alias distance is 1, then the core only goes through one penalty cycle; otherwise, there are no penalties. Once the value is ready to be read, the [ ]-node reads the value of register `rs`.

In our running example, addition and multiplication cannot be performed simultaneously. Since the multiplication takes three cycles, a structural hazard occurs when a multiplication starts and must be handled. Thus, the second speculation point, at γ_{res}, controlled by the `opcode` value, causes two penalties when a multiplication occurs. Once the operation is completed, the result is written in the register file at the address `rd` (with the α-node).

Both the output of the alias detection and the `opcode` are fed to the FSM, which adds the appropriate number of penalties based on the signals' values during execution. P_{res} represents the number of pipelined penalties caused by γ_{res}, and P_{regs} those caused by γ_{regs}.

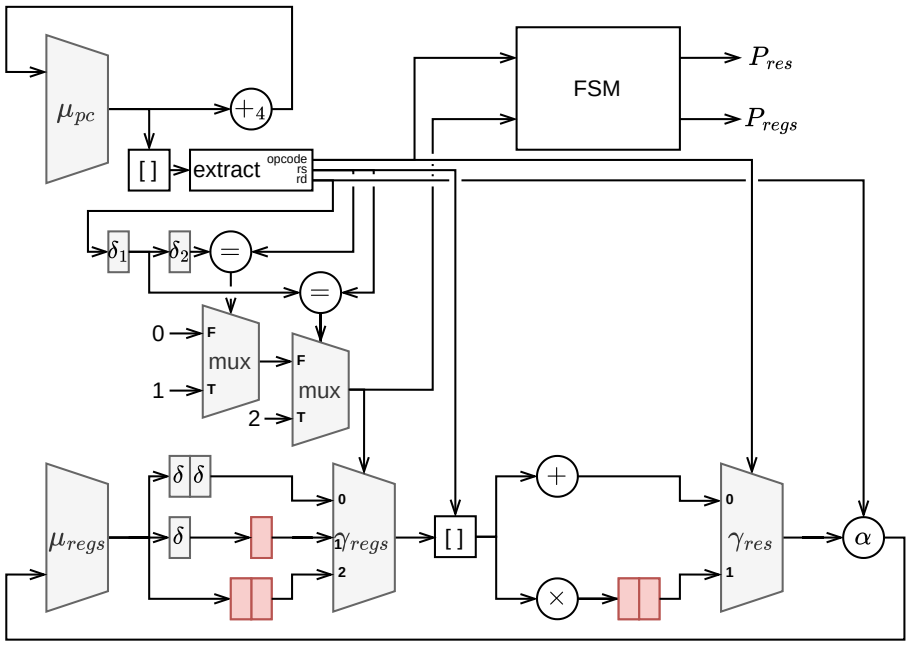

Figure 2 Gated-SSA representation of a simple core with two instructions *add* and *mul*, extracted from an HLS flow implementing Speculative Loop Pipelining. This simple core has two speculation points, γ_{regs} and γ_{res}, controlled by alias-detection logic and the *opcode*, respectively.

3.2 Model Description and Extraction

Unlike most analyses in the state of the art, we *do not* directly model the core pipeline; instead, we determine the number of penalties it incurs.

The key point of speculative loop pipelining is the ability to produce a pipeline with an II equal to 1 (i.e., a new iteration/instruction starts at every cycle). In such a design, where an instruction starts speculatively every cycle, in case of a misspeculation for an instruction I, the subsequent instructions that have started after I are *canceled*. Then, the execution of I is stalled for a given number of cycles to wait for the correct value before starting the next iteration in a correct state.

In a processor core generated by such flows, speculation depends on both the core's internal state and its inputs (e.g., the instruction fetched). Thus, we need to model all parts of the processor core that are involved in speculation. To do so, for a given core, we construct a timing model represented by the 4-tuple (Q, I, F, T). Each component of the tuple will be defined more precisely in the following sections.

- Q is the state-space of possible core *states*. Q is a set of n-tuples, each n-tuple storing the values of the variables; Q is an *abstract* state of the processor, that selects variables relevant to timing analysis only, and allows for unknown variables;
- I is the set of possible *inputs*, each being defined by an m-tuple containing the values of input variables;
- F is a *penalty estimator* function $F : Q \times I \longrightarrow P(\mathbb{N})$, with $P(\mathbb{N})$ the power set of $\mathbb{N}$ that gives for a given state $q \in Q$ the set of possible penalties throws by this state;
- And T is the transition function from a state q with an input i and p penalties.

From a timing estimation perspective, the central part of the model is the F function. While T lets us traverse the states the processor passes through during the execution of a basic block, F captures the possible timing penalties incurred during execution.

State and Input Modeling

We define V_{core} as the set of *core variables* used in computations (e.g., core registers such as `pc` and other registers). These variables can be of any type, from single-bit to n-bit integers to arrays of values. Among these variables, only a subset, $V_{state} \subseteq V_{variable}$, impacts speculations and therefore timing. We call the elements of V_{state} the *state variables*. A state $q \in Q$ is an assignment of values to each $v \in V_{state}$ representing a possible processor state.

In a state q, each variable $v \in V_{state}$ can take:

- either a value within its domain (e.g., 0 to $2^{32} - 1$ for a 32-bit variable),
- or the special value *unknown*, which represents any possible value that the variable may take.

For instance, at the beginning of the analysis of a basic block, we can define an initial state q_{init} such that $\forall v \in V_{state}, v = unknown$. This represents a state in which no information about the prior execution is available.

Similarly to Q, we define the set I_{core} of *core inputs* as the set of variables originating from outside the processor (e.g., data fetched from memory, such as *instructions*). The set $I_{state} \subseteq I_{core}$ is the subset of *core inputs* that impact speculations, and an input $in \in I$ is an assignment of values to each $i_{state} \in I_{state}$. In the simplest form, a state $i_{state} \in I_{state}$ has size one, and represents the instruction to be executed. As further detailed in Section 4, the timing analysis will set the input state at each analysis step.

Extraction of State and Input

The set of inputs I_{state} and variables V_{state} can be obtained in a single pass from the HLS intermediate representation. Inputs can be readily identified in the Gated-SSA representation, as accesses to values external to the core are performed through read operations ([ ]-nodes). However, not all [ ]-nodes are accesses to external values. For instance, in Figure 2, the [ ]-node after γ_{regs} is an access to a register, whereas the [ ]-node after μ_{pc} is an access to external memory. Access to external inputs can be identified through information available in the HLS flow. There are two types of variables in V_{core} that together compose the processor state: variables from the μ-nodes, that change at the instruction level, and variables from the δ-nodes, that change at the cycle level.

Algorithm 1 describes how to construct V_{state} and I_{state}. It traverses the Gated-SSA representation recursively, starting from nodes that affect speculation and therefore timing (γ-nodes). It begins by initializing a worklist with the control signals of all γ-nodes. It then moves backward in the Gated-SSA representation until it finds μ-nodes or [ ]-nodes. For each encountered operation, if it is a δ-node, the algorithm adds the operation to the set V_{state} and its operand to the worklist; otherwise, it adds each operand of the operation to the worklist. The process repeats until a μ-node or a [ ]-node is reached. For a μ-node, since it represents a loop backedge in the Gated-SSA representation, it is added to V_{state}, but its operands are not added to the worklist. For a [ ]-node, if it represents an access to an external memory, it is added to I_{state}.

Applying Algorithm 1 to the running example in Figure 2 yields one input variable and two state variables:

- The input variable is the instruction, extracted from the [ ]-node after the μ_{pc}.
- The two state variables are the two δ-nodes used in the alias detection logic.

Algorithm 1 Algorithm to build V_{state} and I_{state}.

Require: Γ ▷ The set of speculated γ-nodes
Ensure: V_{state} ▷ The set of state variables
Ensure: I_{state} ▷ The set of state inputs
1: $WorkingList \leftarrow \emptyset$ ▷ List of operations
2: $V_{state} \leftarrow \emptyset$
3: $I_{state} \leftarrow \emptyset$
4: **for** $\gamma \in \Gamma$ **do**
5: $WorkingList$.push(γ.control)
6: **end for**
7: **while not** $WorkingList$.empty() **do**
8: $operation \leftarrow WorkingList$.pop()
9: **if** $operation = \delta$ **then**
10: V_{state}.add($operation$)
11: $WorkingList$.push($operation$.operands)
12: **else if** $operation = \mu$ **then**
13: V_{state}.add($operation$)
14: **else if** $operation = [\,] \wedge [\,]$.isExternal **then**
15: I_{state}.add($operation$)
16: **else**
17: $WorkingList$.push($operation$.operands)
18: **end if**
19: **end while**

The three δ-nodes before the γ_{regs} are not part of V_{state}, because they are not in the datapath of any control signal.

Penalty Estimator Construction

The penalty estimator function F takes as input a pair ($q \in Q$, $in \in I$) and computes the set of possible penalties that may occur when executing an instruction in state q with input i. Since variables in q may take the value *unknown*, and because the number of penalties depends on the values of the variables in q, multiple possible penalties may be obtained. For instance, in our running example, if the destination address `rd` from two cycles ago is different from the source address `rs`, but the destination address of the last cycle is *unknown*, then the number of penalties can be either two or zero.

Function F is built from the Finite State Machine (FSM) of the extracted Gated-SSA representation, and the logic that computes the control signals entering the FSM.

Knowing the mapping between a control value and the corresponding number of penalties is not sufficient. We also need a means to compute this control value from the current state and the inputs. This is achieved by leveraging the *constants propagation* optimization pass of the HLS flow. We can exploit this pass by replacing, in the Gated-SSA representation, each δ-node and μ-node used to construct V_{state}, as well as $[\,]$ used to construct I_{state}, with their constant values. We then apply the constant propagation pass and retrieve the resulting values at the FSM inputs. If one of the inputs of an operation is a value *unknown*, constant propagation fails, and all possible penalties are taken.

In our running example, the FSM has two inputs, one for the γ_{regs} and the other for γ_{res}. The value of γ_{regs} is $\{0 \to 0, 1 \to 1, 2 \to 2\}$, the value assigned to γ_{res} is $\{0 \to 0, 1 \to 2\}$. Let us consider that the current iteration has for input 11110 and the current state is $q = (\delta_1 = 01, \delta_2 = 10)$. Thus, for the γ_{res} control signal, the input constant 11110 will be split in three part: *opcode* 1 (multiplication), `rd` 11, and `rs` 10. Since the control signal of γ_{res} is `opcode` and the latter is equal to 1, then the corresponding number of penalties due to γ_{res} is 2. For the γ_{res}, `rs` value is compared with $\delta_1 = 01$ and $\delta_2 = 10$. The former comparison returns $false$, whereas the latter returns $true$. Thus, the control signals for both multiplexers are constant, so the multiplexer can be simplified using constant propagation. Finally, the constant 1 arrives at the FSM input associated with γ_{regs}, and the corresponding penalty number is 1. In this example, for the given state and input, the corresponding number of penalties is 3: 1 to wait for the result of a previous instruction and 2 due to the multiplication.

Transition Function

As mentioned above, our analysis operates from the starting state of one instruction to the starting state of the next. In an ideal execution on a processor with an *initiation interval* (II) of one, there is one cycle between two iterations (i.e., instructions). However, in the presence of mispredictions, additional penalty cycles may be introduced between the start of one instruction and the start of the next.

Having now defined the state space Q and described how the temporal behavior of a state can be obtained through the function F, we now describe the transition function T used in our analysis to move from one state to the next. Function $T : (Q \times I \times \mathbb{N} \longrightarrow Q)$ takes a 3-tuple $q \in Q$, $i \in I$, and $p \in F(q, i)$ and computes the initial state q' of the next iteration.

Before defining T formally, let us illustrate T on our running example, in the situation where δ_1 is *unknown*, δ_2 has a concrete value, and F gives us two possible values for the number of penalties: 0 or 2. In the first case (no penalty), the next value of δ_1 is the destination address of the current instruction, thus dependent directly on the input vector, and the next value of δ_2 is the current value of δ_1 (i.e., *unknown*). For the second case (2 penalties), the computation of the next state is more complicated. The δ-nodes are cycle-wise variables; their values change at each cycle. Since there are two invalid cycles before the start of the next iterations, we need to handle the value carried by these cycles. In the HLS flow, to prevent aliasing between the current iteration and cancelled iterations, the alias-check values (i.e., source addresses) are paired with a 1-bit `valid` value. This bit is set to 0 for a cancelled value. For simplicity, we call values with their `valid` bit equal to 0 *invalid*. Thus, in our example, after the first stalled cycle, δ_1 takes the value *invalid* and δ_2 the current value of δ_1 (i.e., *unknown*). After the second stalled cycle, δ_1 gets another *invalid* value and δ_2 the first invalid value of δ_1. Finally, at the beginning of the next iteration, δ_1 takes the destination address of the previous iteration, and δ_2 the *invalid* value of δ_1. δ_1 takes the destination address of the previous instruction because this instruction is stalled for two cycles and then continues its execution where it was stopped.

Constant propagation is used to compute the transition function from a state q with an input i to the state q'. As seen in our running example, the two types of state variables evolve at different rates: instruction-level for μ-nodes and cycle-level for δ-nodes. While the evolution of variables encoded in μ-nodes is straightforward with the constant propagation, the ones encoded in δ-nodes are more complex and depend on the result of F.

For the δ-node variables, there are two cases. Either their value came from some computation or from another δ-node. The former case and μ-nodes variables are computed as F. The analysis replaces each μ-node, δ-node, and [ ]-node with their value for the current cycle. Then, it applies the constant propagation pass, and the value at the input of each μ-node and δ-node is their value for the next iteration. For the latter case, we need to keep track of the sequence of δ-nodes' predecessors of each δ-node, and we need to use the number p of penalties. We can distinguish three cases:

1. if $p = 0$ then there is no penalty, and thus, the δ take the value of its direct predecessor.
2. else, if the number of predecessors of the δ-node is less than or equal to p then the next value of δ coming from its direct predecessor is an *invalid* value.
3. otherwise, it takes the value of its p-th predecessor.

For instance, if we have a sequence $\delta_0, \delta_1, \delta_2$ and $p = 1$, then δ_0 takes for the next iteration the value obtained through constant propagation; δ_1 takes an *invalid* value since its number of predecessors is equal to p; and finally, δ_2 takes the value of δ_0 since δ_1 takes the value δ_0 during the stalled cycle (and δ_0 an *invalid* value), then δ_1 forwards its value to δ_2 for the beginning of the correct iterations after the stall. δ_0 forwards its *invalid* value to δ_1 and δ_0 receives the correct value computed after the stall.

4 Timing Analysis

The objective of the timing analysis is to estimate the WCET of a basic block (BB) for a given task on a synthesized microarchitecture. The results of our analysis are then used in a state-of-the-art IPET analysis technique to estimate the WCET of the entire task. Our analysis iterates over the BB's instruction sequence from first to last.

For each instruction, the analysis starts from a set of states inherited from previous instructions, or the initial state q_{init}. It then computes one or more successor states that will serve as initial states for the next instruction in the sequence, along with the number of penalties incurred to reach them.

The analysis constructs an acyclic graph, with nodes representing states and edges representing transitions between states. Transitions are labeled with the number of penalties between the source and destination nodes. The WCET of the BB can be obtained using a longest-path algorithm on the resulting graph. As detailed later, a node-merging process is applied to avoid state-space explosion.

The initial state of the analysis can be either the state in which all state values of the model are *unknown*, or the final states of the previous basic blocks, to augment analysis precision at the cost of higher analysis cost.

To illustrate our analysis, let us take as an example the timing model derived from our running example in Figure 2, which is shown in Figure 3. In this model, the only input is the 5-bit instruction, with 1-bit `opcode`, 2-bit destination register `rd`, and 2-bit source register `rs`. The state Q of the model is a tuple (δ_1, δ_2) where δ_1 is a 2-bit value containing the `rd` of the previous cycle and δ_2 is a 2-bit value containing the value of `rd` from two cycles in the past. Only the logic used to compute the `opcode` and to detect aliases was kept during model extraction.

Algorithm 2 shows how our analysis estimates the WCET of a basic block. It first adds all possible initial states to a worklist (variable $WorkList$). Then, for each input vector i, in the program order, and for each state q in $WorkList$, it computes all possible numbers of penalties P with the penalty estimator function F. Next, for each $p \in P$, the analysis computes, using the transition function T, the next state q' and adds the edges $q \xrightarrow{p} q'$ to the graph, and adds the state q' to the worklist.

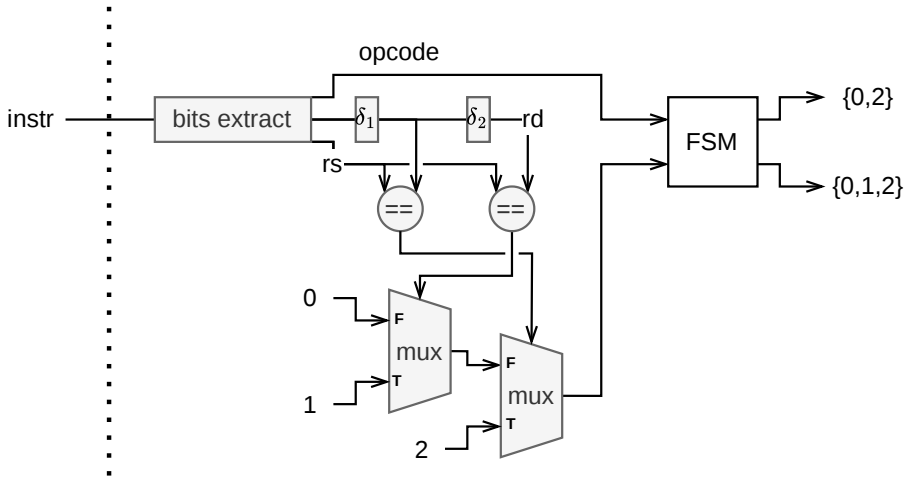

Figure 3 Gated-SSA subset of Figure 2 representing the extracted model. Inputs are at the left of the dashed line, δ_1 and δ_2 are the two elements of Q, and the remaining is the logic used to compute the control signals of the two speculated γ-nodes, which are used to compute T and F.

Once all input vectors have been computed, the analysis adds, for all final states in $WorkList$, edges to a *sink* node. This node is used as the destination node in longest-path computation. Similarly, the analysis adds an edge from *src* to each $q \in Q_{init}$ to obtain a single-source point.

Figure 4 shows the graph obtained by our analysis, with the *src* and *sink* nodes, for the basic block shown in the bottom-left part of the Figure, executed on our example model. The initial state, the left-most one in the graph, has a state $q = (\text{unknown}, \text{unknown})$ and an input vector $i = (00110)$. Since the `opcode` depends solely on the known input (i.e., 0), this signal throws zero penalties. However, since the alias detection signal depends on both δ_1 and δ_2 that have the value *unknown*, this signal can throw three possible numbers of penalties: 0, 1, or 2 penalties. Thus, we have three possible next states: one with no penalty, one with 1 penalty, and one with 2 penalties. The next state with zero penalty cycles is straightforward to compute: δ_1 takes the value of *rd*, which is 01, and δ_2 takes the old value of δ_1, which is *unknown*. For the two other states, the calculation is less straightforward, as the analysis needs to account for penalties to compute the next values of both δ. For instance, the state that starts after one penalty (the middle one in Figure 4) starts after two cycles instead of one. In the first cycle, the penalty, δ_1, takes an *invalid* value, and δ_2 takes the old value of δ_1. Then, in the second cycle, δ_1 takes the correct 01 value, and δ_2 takes the old value of δ_1, which is *invalid*. The state that starts after two penalties, the top one, is computed similarly.

4.1 State Explosions Issue

We observe in Figure 4 that after the first iteration, our analysis has three possible states from which the next iteration can start. Our analysis repeats the previous steps for each possible state to obtain the next initial states of the third instruction, and again to obtain the final states (in green in Figure 4), which are connected to the *sink* node. The rapidly increasing number of states to explore raises the issue of the applicability of our analysis to a real sequence of instructions on a real processor.

Algorithm 2 WCET Estimation of a Basic Block.

Require: I ▷ Sequence of inputs
Require: Q_{init} ▷ The set of initial states
Ensure: $WCET$ ▷ WCET of a given BB represented by I
1: $Edges \leftarrow \emptyset$
2: **for** $q \in Q_{init}$ **do**
3: $Edges$.add$(src, 0, q)$
4: $WorkList$.push(q)
5: **end for**
6: **for** $i \in I$ **do** ▷ One input vector per instruction in the BB
7: $NextWL \leftarrow \emptyset$
8: **for** $q \in WorkList$ **do**
9: $P \leftarrow F(q, i)$
10: **for** $p \in P$ **do**
11: $q' \leftarrow T(q, i, p)$
12: $Edges$.add(q, p, q') ▷ $q \xrightarrow{p} q'$
13: $NextWL$.push(q')
14: **end for**
15: **end for**
16: $WorkList \leftarrow NextWL$
17: **end for**
18: **for** $q \in WorkList$ **do**
19: $Edges$.add$(q, 0, sink)$ ▷ All final states are connected to a *sink* node
20: **end for**
21: $WCET \leftarrow$ longPath$(Edges, src, sink)$ ▷ Longest path in $Edges$ from q_{init} to *sink*

Fortunately, because of the abstraction in our model, many possible initial states are equivalent within an iteration. For instance, in Figure 4, all initial states of the third iteration are equivalent. Our merging process merges nodes that are identical and further have an incoming edge with the same label (i.e., the number of penalties). In our example, the *blue* nodes of Figure 4 are merged into a single node. This merge strategy drastically reduces the number of states to explore.

4.2 Impossible Sequence of States Issue

If we look at the sequence of states depicted by the blue edges in Figure 4, we notice that this sequence cannot occur in any execution. Indeed, the four penalties introduced by the second iteration are decomposed as follows: two cycles for the multiplication penalty, two cycles for an alias between `rs` and `rd` from the previous instruction, and one in δ_1. And if we look at the content of δ_1 and the value of `rs`, we can see that the two are not equal and thus cannot alias.

The presence of this sequence of states stems from how both T and F work. To compute the next value of a state variable or the value of a control signal, our analysis extracts the logic from all input and state variables and routes it to the state-variable operator (μ-nodes or δ-nodes) or the FSM operator. Then, it assigns each input and state variable its current value and performs constant propagation through the extracted logic. In our example, the select signal of the right-most multiplexer in Figure 3 is the result of the comparison between `rs` and `rd` stores in δ_1. Since, at this state, these two values are constant (11 for `rs` and 01 for `rd`) and different, the select signal is *false*, and the multiplexer forwards the value of

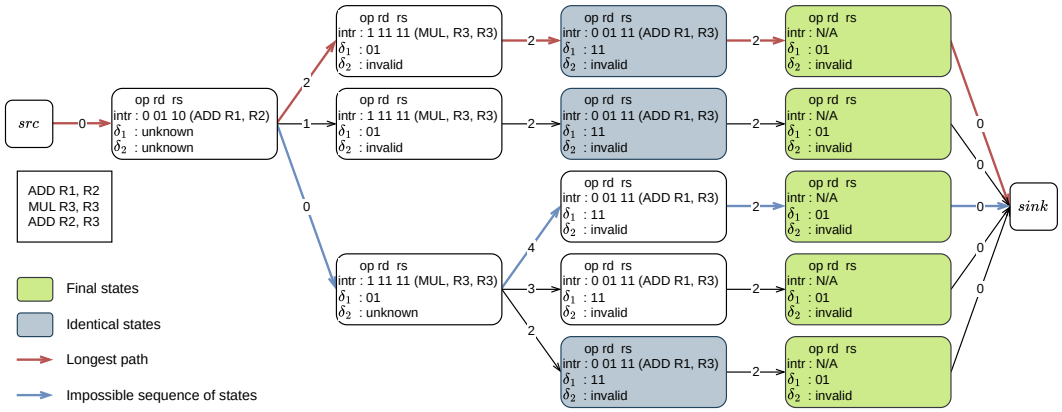

Figure 4 Directed acyclic graph obtained after the analysis of a simple BB of three instructions. The initial state q_{init}, at the left of the graph, is connected to the *src* node, and all final states, at the right of the graph, are connected to the *sink* node. The longest path in this result graph is displayed in red. The blue path is an infeasible sequence of states.

the left-most multiplexer. The select signal of the latter depends on an unknown value, the content of δ_2, and thus is itself unknown. Since the multiplexer can forward either 0 or 1, the signal sent to the rightmost multiplexer is *unknown*, and thus the control signal sent to the *FSM* is *unknown*.

Although we did not observe this phenomenon in the processor we synthesized (see Section 5), it could happen on other architectures that the longest path includes these impossible sequences of states, resulting in pessimistic WCET estimates. A different arrangement of the multiplexer-tree can prune these impossible state sequences, while creating others at other points of the analysis.

5 Experimental Results

5.1 Implementation and experimental setup

To our knowledge, only SpecHLS HLS flow [10] implements the speculative loop pipelining technique on which our technique is based. Thus, we decide to use SpecHLS, which heavily relies on the MLIR compilation framework[1] for analyses and code generation.

The Heptane WCET estimation tool [13] was used for WCET estimation at the task-level. We additionally implemented a handcrafted timing model as a baseline.

We performed the analysis on a micro-architecture model corresponding to the 5-stage rv32i RISC-V processor without forwarding. This model has two speculations. The first one speculated that the register currently read has no write operation in the two previous cycles, and the second one that the instruction is not a branch, which corresponds to the model used by Feuilletin et al. [8] to evaluate their analysis.

We use a subset of the Mälardalen benchmarks [11] for our experiments. This subset excludes benchmarks with floating-point operations, as our architecture does not support them and they must be emulated in software. Since loop bounds for emulated floating-point operations, required by Heptane, are difficult to derive due to the complexity of their implementation, we chose, for simplicity, to remove benchmarks that use floating-point operations.

[1] `https://mlir.llvm.org/`

We have compared our analysis to the PGA analysis previously published in [8]. PGA estimates the WCET of a basic block by unrolling the temporal model to create a graph and replacing fetch logics with the sequence of instructions. Then it performs constant propagation and dead-code elimination passes to specialize each part of the graph, and then uses a longest-path algorithm to determine the path in the graph that incurs the most penalty. Their timing model differs from ours: they do not perform any exploration of possible processor states, and further manage aliases between instructions in a safe yet pessimistic way. We implemented both analyses from a single run of the SpecHLS flow.

The baseline timing model implemented in Heptane was slightly modified to match the processor core we generated. This handcrafted model uses a reservation table to implement alias-detection logic and associated stall penalties, along with the branch instruction penalty. Since the microarchitecture does not implement advanced features such as cache or dynamic branch speculation, the handcrafted model is simple and small yet accurate.

5.2 Quality of WCET estimates

The first experiment compares the proposed approach with a handcrafted pipeline model and the PGA analysis previously published in [8]. The results are summarized in Table 1. The table displays the raw WCET estimates for the three techniques, as well as the ratio of the proposed approach to the existing approaches.

It is important to note that the experimental results for PGA are slightly different than those reported in [8]. Indeed, in their original work, Feuilletin et al. considered that the first instructions of a basic block could not alias with previous ones, which can be unsafe. In the numbers reported in Table 1, we consider for all three techniques that the first instruction may alias with previous ones, resulting in a pessimistic but safe WCET.

We observe that the proposed approach is less pessimistic than PGA, reducing the WCET estimate by an average of 3.6%. This difference is due to the fact that PGA, instead of exploring states, always considers the worst-case penalty at the instruction level, resulting in more pessimistic WCET estimates. The proposed approach is also as precise as the handcrafted model, always returning the same WCET estimate. This shows that our analysis, for this simple core, achieves to model the temporal behavior of the microarchitecture.

5.3 Analysis time

The second experiment evaluates the scalability of the proposed approach.

We first analyze the efficiency of the merging strategy proposed in Subsection 4.1. Figure 5 depicts the average number of states explored per instruction with and without the merging strategy, represented for different basic block sizes. For convenience, we only selected blocks with fewer than 30 instructions, which represent 95% of all basic blocks analysed. We observe that, with the merging strategy, the shorter the basic block, the higher the number of states per instruction. This tendency is due to the state of basic block analysis with value *unknown*, which leads to the creation of several states at the start of the basic block analysis. These states are quickly merged together when analyzing the next instructions. On average, across all basic blocks analyzed, the analysis used 1.6 states per instruction, since small basic blocks are the most common.

Without the merging strategy, the number of states per instruction does not decrease with the size of the basic blocks. As with the merging strategy, the unknown initial states yield a certain number of states, with more and more variables known. Thus, each resulting state does not split into many other states, leading to a constant number of states throughout the analysis. On average, across all basic blocks analyzed, the analysis used 2.62 states per instruction.

Table 1 Estimated WCET for the different benchmarks using the proposed approach and existing approaches (handcrafted model and PGA from [8]).

Benchmark	WCET estimate (cycles)			Comparison	
	Handcrafted	PGA	Proposed approach	vs. Handcrafted	vs. PGA
crc	397 422	411 199	397 422	0.00 %	-3.35 %
cover	15 135	15 890	15 135	0.00 %	-4.75 %
fibcall	1 695	1 816	1 695	0.00 %	-6.66 %
statemate	23 335	23 879	23 335	0.00 %	-2.28 %
ns	68 411	70 282	68 411	0.00 %	-2.66 %
bsort100	1 703 804	1 750 610	1 703 804	0.00 %	-2.67 %
adpcm	32 408 016	33 672 165	32 408 016	0.00 %	-3.75 %
jfdctint	287 047	300 108	287 047	0.00 %	-4.35 %
ndes	265 759	274 751	265 759	0.00 %	-3.27 %
edn	14 402 879	14 879 758	14 402 879	0.00 %	-3.20 %
matmult	20 535 407	21 229 581	20 535 407	0.00 %	-3.27 %
bs	499	520	499	0.00 %	-4.04 %
fir	413 950	428 261	413 950	0.00 %	-3.34 %
lcdnum	1 477	1 580	1 477	0.00 %	-6.52 %
fdct	396 466	409 180	396 466	0.00 %	-3.11 %
prime	4 471 975	4 671 601	4 471 975	0.00 %	-4.27 %
minmax	2 265	2 362	2 265	0.00 %	-4.11 %
expint	1 237 020	1 283 894	1 237 020	0.00 %	-3.65 %
nsichneu	42 750	44 006	42 750	0.00 %	-2.85 %
compress	866 312	893 302	866 312	0.00 %	-3.02 %
insertsort	12 420	12 442	12 420	0.00 %	-0.18 %
			Max	0.00 %	-0.18 %
			Min	0.00 %	-6.66 %
			Avg	0.00 %	-3.59 %

The analysis time using Heptane has also been measured. Across all benchmarks, the average analysis time using the proposed technique is 13 seconds (ranging from 0.47 seconds for `fibcall` to 87 seconds for `nsichneu`). If we do not use the merging strategy, the average analysis time is 28 seconds, ranging from 0.51 (`fibcall`) to 203 seconds (`nsichneu`). This is 20% longer than the PGA analysis we have reimplemented, and 94× longer than the handcrafted model.

5.4 Discussion

Our analysis is implemented within the MLIR framework, which introduces significant overhead, whereas the handcrafted model consists of small, self-contained C code. A previous analysis, PGA, is also implemented in MLIR and therefore suffers from the same overhead. However, PGA does not explore different execution paths and performs constant propagation and dead-code elimination only once per basic block. In contrast, our analysis applies these passes at each instruction to enable path exploration, which results in higher execution time compared to PGA. The execution time of our approach is largely tied to the current implementation, and there remains room for improvement, for example by generating C code rather than relying on the MLIR framework.

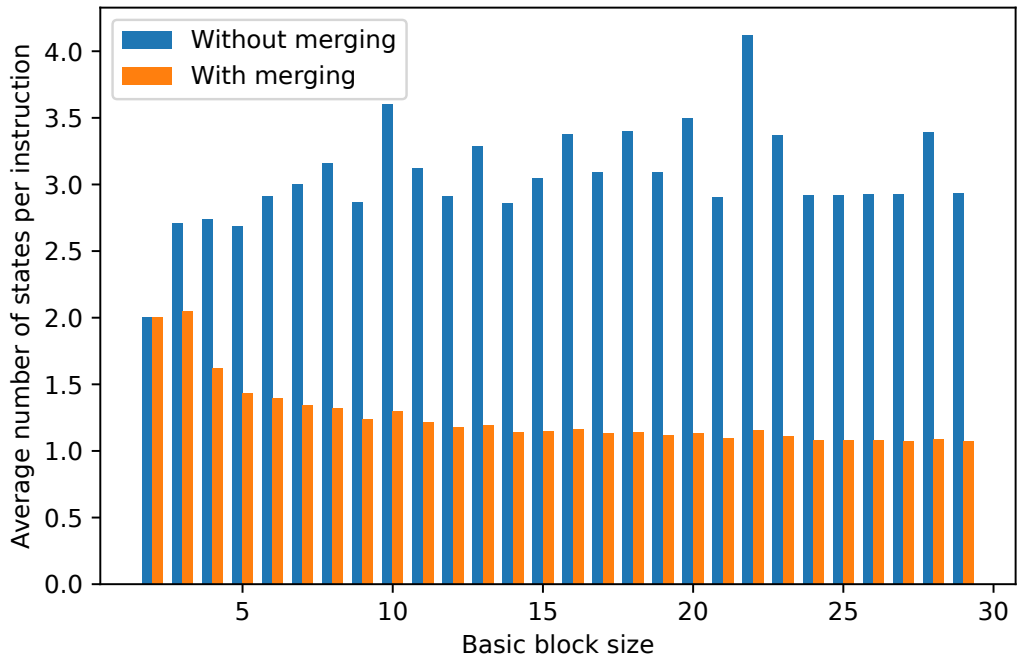

Figure 5 Average number of states explored per instruction for different block sizes. In blue, the number of states without the merging strategy, and in red, with the merging strategy.

Our analysis was applied to a simple five-stage RISC-V core. In this analysis, the different memory components (data memory and register file) are not abstracted away, and all accessed values are treated as unknown. For more complex microarchitectural features, such as caches or branch prediction, an appropriate level of abstraction is required to model array-based structures.

Lastly, the runtime of our analysis is tied to the complexity of the architecture and is likely to increase for more complex architectures featuring caches, superscalar execution, and out-of-order execution. The ability of HLS tools to generate such complex architectures has not yet been demonstrated and remains the subject of future work.

6 Related Works

6.1 Timing Models for WCET analysis

A timing model is required to estimate the WCET of basic blocks in timing analysis tools [21]. This model does not need to concretely implement all the functionality of the target hardware. Instead, a simplified model that is conservative regarding the hardware's timing behavior is sufficient.

The complexity of deriving a timing model strongly depends on the class of processor under consideration. For simpler processors featuring in-order pipelines and no caches, constructing the timing model is relatively straightforward, although still time-consuming. It essentially simulates the flow of instructions through the processor pipeline while accounting for variations in functional unit latencies and memory access times.

For more complex architectures that incorporate performance-enhancing features such as caches, the different hardware components can be analyzed separately (e.g., performing a dedicated cache analysis). Abstract interpretation is commonly used to avoid enumerating all possible architectural states (for instance, abstract cache states are employed instead of concrete cache states [1]). The derived worst-case behavior of each architectural feature can then be incorporated into the pipeline analysis. However, this compositional approach is valid only for architectures that do not exhibit *timing anomalies* [4]. A typical timing anomaly occurs when a locally favorable event (e.g., a cache hit) results in a longer overall execution time than a less favorable event (e.g., a cache miss).

In architectures exhibiting timing anomalies, constructing a timing model becomes significantly more complex. The analyses are less modular and require a more integrated treatment of architectural features. In particular, they involve some form of *exploration* of possible processor states (e.g., cache hit or miss information within the pipeline model). For example, Wilhelm [22] enumerates abstract pipeline states to account for timing anomalies and uses Binary Decision Diagrams (BDDs) to mitigate state-space explosion. WCET estimation techniques based on model checking have also been proposed [6, 5, 17], demonstrating that complex architectures (including caches and, in [17], branch prediction) can be modeled to derive precise WCET estimates. However, all authors report state-space explosion for certain benchmarks, even in the absence of timing anomalies. Our technique is similar to [22] in its use of abstract interpretation to enumerate abstract architectural states. Likewise, as in model-checking-based approaches, we explore all possible (abstract) architectural states. Our main distinguishing feature compared to these two classes of techniques is the automatic extraction of the timing model.

6.2 Automatic Extraction of Timing Models

The first attempt at automatically extracting timing models from a VHDL representation was presented by Schlickling et al. in [18]. Their technique first prunes the VHDL code to reduce its size. It then automatically extracts the state variables, along with part of the logic used to compute the next state of each pipeline stage. In contrast to our approach, theirs requires user interaction to complete the model extraction process and, moreover, produces very large models.

A technique for the reverse engineering of Chisel hardware designs is presented in [3]. This approach reconstructs the pipeline datapath from the Chisel source code. The resulting datapath must be manually validated. Moreover, the technique excludes control-path extraction, making it unsuitable for direct WCET estimation.

Amalou et al. propose an automatic derivation of timing models for WCET analysis based on machine learning techniques [2]. While the model derivation process is fully automated, the models are trained on empirical data and therefore do not provide formal safety guarantees. As a result, such models may not be suitable for highly time-critical software.

Feuilletin et al. [8] propose a technique that automatically extracts processor timing models from an HLS flow. Their analysis makes conservative assumptions regarding instruction dependencies. Moreover, it systematically relies on local worst-case assumptions to estimate the WCET of a basic block. As a result, the approach is not applicable to processors exhibiting timing anomalies. In contrast, our technique can handle timing anomalies, albeit at the cost of increased analysis time.

7 Conclusion

We present a general WCET analysis technique for processors generated using high-level synthesis. Our analysis is based on an exploration of abstract processor-core states to estimate the maximum number of stall cycles any basic block can encounter during execution. The number of states explored is kept small (on average 1.6 states per instruction), through an efficient merging strategy. Our WCET estimates are as precise as those obtained with a handcrafted timing model and are safe against timing anomalies, as they consider all possible sequences of abstract states for a basic block.

The class of core architecture considered is currently very simple. Our future work will explore more complex processors that feature caches, out-of-order execution, and branch prediction. In addition, while this paper focused on the processor core, we believe our analysis applies to any specialized design synthesized by an HLS flow implementing speculative loop pipelining.

References

1 Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Proceedings of the Third International Symposium on Static Analysis*, SAS '96, pages 52–66, Berlin, Heidelberg, 1996. Springer-Verlag.

2 Abderaouf N Amalou, Elisa Fromont, and Isabelle Puaut. CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers. In Alessandro V. Papadopoulos, editor, *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, volume 262 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2023.7.

3 Samira Ait Bensaid, Mihail Asavoae, Farhat Thabet, and Mathieu Jan. Deriving pipeline models for timing analysis from high-level hdl processor designs. In *2022 20th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–8, 2022. doi:10.1109/MEMOCODE57689.2022.9954598.

4 Benjamin Binder, Mihail Asavoae, Belgacem Ben Hedia, Florian Brandner, and Mathieu Jan. Is this still normal? putting definitions of timing anomalies to the test. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 139–148, 2021. doi:10.1109/RTCSA52859.2021.00024.

5 Franck Cassez, Pablo Gonzalez de Aledo, and Peter Gjøl Jensen. *WUPPAAL: Computation of Worst-Case Execution-Time for Binary Programs with UPPAAL*, pages 560–577. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-63121-9_28.

6 Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *Open Access Series in Informatics (OASIcs)*, pages 113–123, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2010.113.

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

8 Thomas Feuilletin, Dylan Leothaud, Simon Rokicki, Steven Derrien, and Isabelle Puaut. Automatic Extraction of Timing Models for WCET Estimation From a High-Level Synthesis Flow. In *DATE 2026 - Design, Automation and Test in Europe Conference*, Verona, Italy, April 2026. URL: https://hal.science/hal-05365718.

9 Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. Design exploration of risc-v soft-cores through speculative high-level synthesis. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–6. IEEE, 2022. doi:10.1109/ICFPT56656.2022.9974478.

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

11 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *Open Access Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2010.136.

12 Sebastian Hahn and Jan Reineke. Design and analysis of sic: A provably timing-predictable pipelined processor core. *Real-Time Systems*, 56(2):207–245, 2020. doi:10.1007/S11241-019-09341-Z.

13 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, pages 8:1–8:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASIcs.WCET.2017.8.

14 Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Speculative dataflow circuits. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 162–171. Association for Computing Machinery, 2019. doi:10.1145/3289602.3293914.

15 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, 1995. doi:10.1145/216636.216666.

16 Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*, pages 12–21. IEEE, 1999.

17 Armel Mangean, Jean-Luc Béchennec, Mikaël Briday, and Sébastien Faucou. WCET Analysis by Model Checking for a Processor with Dynamic Branch Prediction. In Kamel Barkaoui, Hanifa Boucheneb, Ali Mili, and Sofiène Tahar, editors, *Lecture Notes in Computer Science*, volume 10466 of *Lecture Notes in Computer Science*, pages 64–78, Montréal, Canada, August 2017. Springer. doi:10.1007/978-3-319-66176-6_5.

18 Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for wcet analysis. In *LCTES'10*, LCTES '10, pages 67–76, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1755888.1755899.

19 Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 47–55, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/207110.207115.

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

21 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008. doi:10.1145/1347375.1347389.

22 Stephan Wilhelm. *Symbolic representations in WCET analysis*. PhD thesis, Saarland University, 2012. URL: http://scidok.sulb.uni-saarland.de/volltexte/2012/4914/.