2026 Design, Automation & Test in Europe Conference (DATE 2026)

Automatic Extraction of Timing Models for WCET Estimation From a High-Level Synthesis Flow

Thomas Feuilletin*, Dylan Leothaud*, Simon Rokicki*, Steven Derrien†, Isabelle Puaut*
*Univ Rennes, Inria, CNRS, IRISA
†UBO, Lab-STICC

Abstract—**Real-time, domain-specific processors require faithful timing models for WCET analysis. However, existing models are typically hand-crafted from sparse documentation, making them error-prone and difficult to maintain. This work aims to automatically extract WCET timing models from single-issue in-order processor pipelines generated by High-Level Synthesis (HLS). By deriving timing models directly from the SpecHLS intermediate representation, the models are faithful by construction. Experimental results show that our timing-model extraction process generalizes across diverse RISC-V core variants and yields WCET estimates within 0.48% on average of those from a hand-crafted model, on the Mälardalen WCET benchmarks.**

I. Introduction

Embedded systems benefit from domain-specific CPU cores, which can be used to improve performance, energy efficiency, and/or reliability. Among these systems, some require guarantees on their timing behavior, which are obtained through Worst-Case Execution Time (WCET) analysis [1].

WCET analysis heavily depends on micro-architectural timing models, which vary significantly between designs. The design of timing models is currently performed manually by experts, extracting sparse information from the documentation of the target platform and crafting models from there, which makes it cumbersome and prone to errors [2]. With the widespread adoption of diverse open-source CPU cores following the emergence of the RISC-V standard, it is desirable to facilitate the extraction of such timing models. However, the low level of abstraction of Register-Transfer Level (RTL) representation, combined with the intricacies of Hardware Description Languages (HDLs) makes this approach challenging.

In addition, describing a CPU at the RTL level forces designers to write lengthy, intricate descriptions. Modifying the microarchitecture and/or the Instruction Set Architecture (ISA) at this stage is therefore also very difficult. *High-Level Synthesis (HLS)* tools address these challenges by allowing designers to describe hardware behavior in high-level languages such as C or C++. HLS tools can then generate multiple hardware implementations with different trade-offs. Current commercial HLS tools excel at producing efficient hardware for compute-intensive kernels with simple control flow and regular memory accesses. Still, they are ill-suited to kernels with complex control and memory behavior. For example, synthesizing a processor core from a purely behavioral Instruction-Set Simulator (ISS) description typically leads to inefficient and costly implementations [3].

Recent work has begun to lift these limitations by introducing dynamic and speculative execution mechanisms in HLS for both control-flow and memory dependencies [4]. In particular, the speculative loop pipelining transformation [4], [5] enables the synthesis of competitive pipelined processor cores directly from a C/C++ ISS description through a microarchitectural inference stage [3].

Our work addresses the automation of WCET timing-model generation for single-issue in-order processors by leveraging the internal representation of SpecHLS, an open-source research framework for speculative HLS [5]–[7]. More precisely, the timing model consists of two components: (i) a graph data structure extracted from the HLS Intermediate Representation (IR), and (ii) an analysis named PGA (Penalty Graph Analysis), which processes the graph to extract the WCET of a sequence of instructions. The resulting timing model is faithful to the synthesized hardware *by construction*, since it is derived from the HLS IR. Automating model construction enables us to leverage HLS-driven design-space exploration while eliminating manual modeling errors. Specifically, our contributions are as follows:

- We demonstrate that timing models for WCET estimation can be extracted automatically from an HLS flow, for a variety of single-issue in-order RISC-V processors.
- We show that, even with a basic pipeline alias detection technique, estimates are very close to those obtained using a hand-crafted timing model.
- We further show that the extraction is valid for a variety of core variants, allowing us to benefit from the automatic design-space exploration capabilities offered by HLS.

The paper is organized as follows. Section II provides background information. Section III details our approach for the automatic extraction of timing models, named PGA, for Penalty Graph Analysis. Section IV presents some experimental validation. Related work is presented in Section V. Section VI concludes the paper.

II. Background

A. WCET analysis

The WCET of a task is its maximum execution time when varying its input data and hardware state. WCET estimation techniques can be divided into two categories [1]: *static* and *measurement-based*. In this paper, our focus is on static techniques because they provide safe upper bounds of execution times for real-time systems.

Most static techniques operate on the Control Flow Graph (CFG) of a sequential task, extracted from its binary code. The nodes in the CFG are Basic Blocks (BB), and the edges

978-3-9826741-1-7/DATE26/© 2026 EDAA

represent the control flow between them. Static techniques proceed in two main phases: the WCET of each BB is estimated using *abstractions* of the hardware state; the WCET of the entire task is then calculated by finding the longest path inside the CFG. The most commonly used technique for WCET computation is the Implicit Path Enumeration Technique (IPET) [8]. IPET computes a task's WCET by solving an Integer Linear Programming (ILP) system of equations. An ILP solver will explore all the possible paths *implicitly* to find the worst one. More precisely, the IPET ILP system maximizes the following objective function: $\sum_{i=1}^{N} x_i \times c_i$, where N is the number of BBs in the task, x_i is a variable in the ILP system representing the execution count of $\mathrm{BB_i}$ on the critical path, and c_i is the worst-case execution time of $\mathrm{BB_i}$, considered constant in the ILP system. The ILP system also contains a set of program structural constraints (e.g., constraints on flow among basic blocks, bounds on loop iterations).

In existing WCET estimation tools, the value c_i is evaluated using a hand-crafted timing model of the processor, which accounts for all processor micro-architectural features (caches, pipelines, branch prediction, etc.). For functional units with variable latencies, the timing model accounts for the worst-case latency. A typical analysis implemented in the timing model is the *pipeline analysis*. It models the flow of instructions in the processor pipeline, including dependencies between instructions. Another analysis required on architectures with caches is the *cache analysis*, which, for every memory access, determines if the access always results in a *hit* in the cache or may result in a *miss*. In situations where a functional unit has variable latency, the timing model selects the longest latency, which is safe unless the architecture features counter-intuitive behaviors named *timing anomalies* (e.g., a cache hit may result in a longer basic block latency than a cache miss [9]). The design of timing models is known to be a time-consuming and error-prone activity, taking weeks to months [10], and thus deserves automation.

B. High-Level Synthesis of processor cores

HLS translates high-level programs (e.g., C/C++ or SystemC) into RTL hardware. Automating resource scheduling, allocation, and binding enables systematic exploration of performance, area, and power trade-offs. HLS has become a widely-used approach for designing custom accelerators for FPGAs and ASICs, with performance that can approach hand-crafted implementations [3].

Although primarily intended for compute-intensive kernels, HLS can, in principle, generate hardware from arbitrary C/C++ programs. For example, providing an ISS as input to an HLS framework allows the automatic derivation of a processor core implementing the corresponding instruction set. However, current HLS tools perform poorly on applications with data-dependent control flow and irregular memory accesses, often falling back to microcoded-style architectures with limited efficiency.

The primary source of inefficiency is that HLS tools rely on *static* scheduling, which assumes the worst-case outcomes for both control flow and data dependencies. Many recent research works have addressed this issue, enabling the synthesis of circuits exploiting dynamic or even speculative schedules [11]–[14]. Speculative Loop Pipelining (SLP) is a recent optimization technique that enriches classical loop pipelining techniques with speculation on control flow and memory behavior to enable more aggressive hardware schedules [4], [5], [15]. Using SLP, one can automatically derive efficient in-order pipelined processor microarchitectures together with their associated hazard management logic [3]. Without loss of generality, the rest of this paper focuses on the techniques implemented in the open-source SpecHLS[1] flow [5] that will be used in the experimental evaluation.

A central challenge in SLP is deciding where and when to speculate. This problem is addressed in SpecHLS through a custom IR based on Gated-SSA (GSSA) [16], [17], in which so-called γ-nodes expose speculation opportunities. These nodes, a refinement of SSA ϕ-nodes, occur at control-flow merge points and naturally capture branch-dependent values.

```
// CPU architectural state
int pc, ir, r[4], m[...];
// Simulation loop
while (1) {
  // Fetch instruction
  ir = m[pc];
  // Decode instruction fields
  opcode = ir & 0x3;
  imm = ir >> 7 ;
  ra = (ir >> 3) & 0x3;
  rb = (ir >> 5) & 0x3;
  // Execute
  pc = pc + 1;
  switch (opcode) {
    case 0: // ADD ra,rb
      r[rb] = r[ra] + r[rb];
      break;
    case 1: // MUL ra,rb
      r[rb] = r[ra] * r[rb];
      break;
    case 2: // BLT ra,rb,imm
      if (r[ra]<r[rb])
        pc = pc + imm;
      break;
  }
}
```

Fig. 1. Instruction Set Simulator for a simplified processor with only three instructions (ADD, MUL and BLT) and four registers r[x].

This principle is illustrated in Figure 1 with the ISS of a minimalist processor supporting three instructions (*BLT*, *ADD* and *MUL*) and a 4-register register file *r*. Its GSSA IR is displayed in Figure 2.

The IR captures the architectural state (pc, r) at the beginning of each instruction via μ-nodes. From this state, the machine performs an instruction fetch from memory, two register-file reads, and either an addition, a multiplication, or a comparison/branch, depending on the decoded opcode. The new value for r (if any) is committed in the α_r-node, and pc is updated according to whether the instruction branches.

[1]https://github.com/Lord-of-the-RISCs/setup

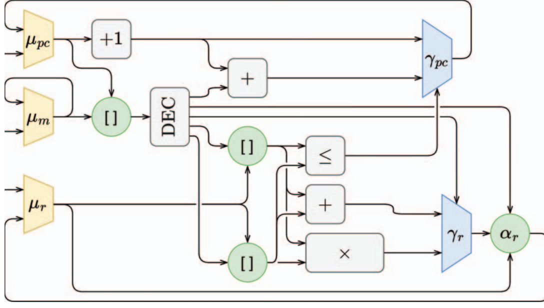

Fig. 2. Gated SSA IR derived from the ISS of Figure 1.

These computations produce the next architectural state for the following instruction. In this representation, the next pc and the instruction result r are selected by two γ-nodes in the IR, γ_{pc} and γ_r.

Speculation when using SLP consists of predicting the outcome of a γ-node to expose a *fast path* with a higher throughput (for example, the $\mathrm{pc}+1$ path in our example). On a misprediction, a hazard recovery mechanism is triggered, incurring execution penalties.

The SpecHLS flow uses a combination of profiling and static analysis to automatically identify the set of γ-nodes that are the best candidates for speculation [7], given a target clock speed constraint. This enables the inference of different microarchitectures with various cost/performance trade-offs.

In the SpecHLS IR, misprediction penalties are explicitly modeled using *penalty nodes*. These nodes represent the additional latency incurred upon hazard recovery, and are inserted into the IR whenever speculation is introduced. Figure 3 shows the IR after applying two speculations: one on γ_{pc} for the branch outcome (for PC) and one on γ_r for the type of arithmetic instruction (ADD or MUL). Given these two speculation decisions, SpecHLS automatically determines the set of penalty nodes needed to obtain a pipelined microarchitecture matching the target clock speed, and where a new instruction is issued at every cycle.

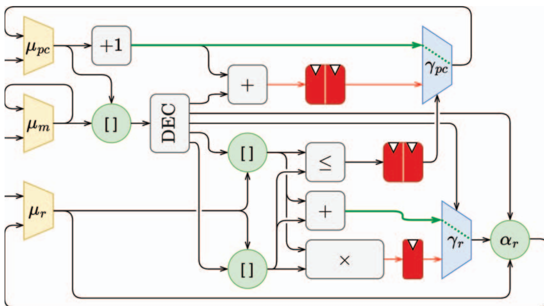

Fig. 3. IR with speculations enabled and their corresponding penalty nodes.

The presence of penalty nodes increases the reuse distance between operations in consecutive instructions (thus reducing the *Recursive Minimum Initiation Interval* [18] of the processor main loop). This increase leads the HLS scheduler to construct deeper pipelines. The number of penalty nodes added along a path corresponds to the speculation depth of the associated γ-node. Deeper speculation can enable higher clock frequencies, but it also amplifies the cost of mispredictions, since each recovery must flush more in-flight operations.

A full description of the SpecHLS flow is outside the scope of this work; interested readers are referred to [3]–[5] for details. However, the key point for the scope of this paper is that the IR shown in Figure 3 captures the cycle-level behavior of the final microarchitecture.

The final step consists of applying a classical retiming algorithm on the modified IR. It is during this final scheduling stage that the precise locations of pipeline stage boundaries are obtained. Figure 4 illustrates the resulting datapath and an example execution trace for the microarchitecture produced on our running example using SpecHLS. In this instance, the tool infers a simple 3-stage pipeline microarchitecture. More complex ISAs and microarchitectures are also supported by the SpecHLS flow. For example, the flow supports the RV32I base ISA and can handle register data hazards and forwarding, as well as limited memory speculation (see Table I in Section IV for more details).

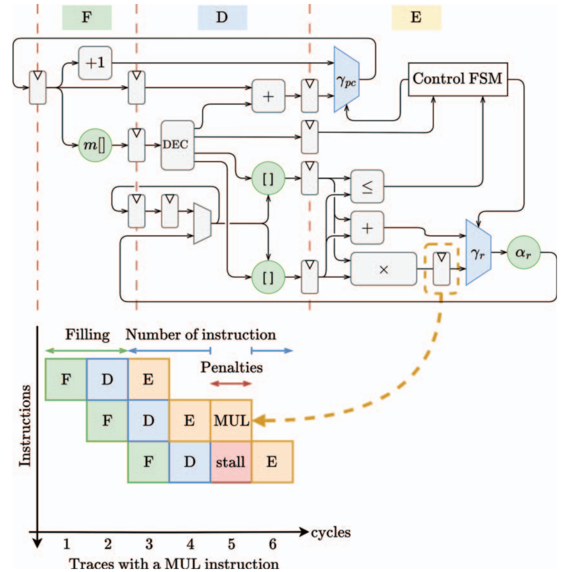

Fig. 4. Resulting 3-staged pipeline datapath and execution trace for our toy ISS. The first stage (F) fetches the instruction from memory, the second stage (D) decodes the instruction and finally the last stage (E) executes the instruction.

III. Penalty Graph Analysis

We exploited the SpecHLS IR through a new analysis called PGA. Its objective is to determine the worst-case number of penalties occurring in a basic block as compared to a perfect pipeline issuing one instruction per cycle. The lower part of Figure 4 shows that the execution time of a sequence of instructions executed in an in-order, single-issue pipeline is the sum of the following elements: the number of cycles needed to fill the pipeline without the last stage, the number of instructions in the sequence, and the number of cycles lost due to pipeline hazards (6 cycles for the code sequence given at the bottom of Figure 4).

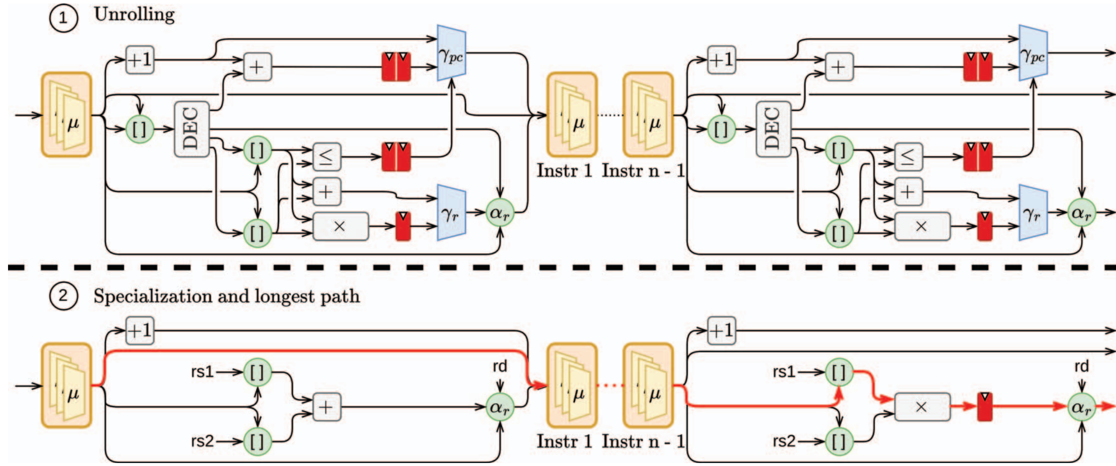

Fig. 5. Different steps of PGA

Given that our IR captures the pipeline hazard penalties, estimating the worst-case execution time of a sequence of instructions in this model amounts to finding the path with the highest number of penalties through the model. Without any additional insight, a safe analysis would assume that any executed instruction results in the maximum number of penalties that can occur for any instruction (one in our example), which would be overly pessimistic. In the rest of this Section, we present the different steps of PGA to determine a more accurate, still safe, WCET estimate for a sequence of instructions.

A. *Basic principle of PGA*

PGA proceeds in three steps, which are illustrated in Figure 5 on a basic block with n instructions. In the example, the first instruction is an addition and the last is a multiplication.

1) **Unrolling:** Duplication of the processor model with one instance per instruction in the basic block. Concretely, the model's main loop is unrolled n times and the μ-nodes are connected, so that the outgoing state of iteration i becomes the incoming state of iteration $i+1$.
2) **Specialization:** For each unrolled iteration, the instruction-memory read (array `m[]`) is replaced by the binary representation of the fetched instruction. Then, standard compiler optimizations (*constant propagation* and *dead-code elimination*) are applied to prune unused logic. In the example, if the instruction is an addition, the multiplication unit and its following γ-node are removed.
3) **Path exploration:** On the specialized, unrolled model, the path that maximizes the number of penalty nodes encountered during execution is identified. First, the μ-nodes of each iteration are merged to obtain single-entry, single-exit regions. The resulting graph is acyclic, so PGA computes the longest path by traversing nodes in topological order. In the example, PGA finds zero penalties for the first instruction and one for the last one.

B. *Handling of data hazards*

SpecHLS implements a technique to handle speculation over read-after-write (RAW) dependencies in the context of pipelined processors [6]. A penalty occurs when an instruction reads a value in the register file that was written by a previous instruction. Control logic is inserted to detect such dependencies. Moreover, δ-nodes are introduced to model the alias-detection logic. A δ-node delays its input by one clock cycle. The handling of RAW dependencies is illustrated in the leftmost part of Figure 6, in which the number of actual penalties may be 1, 2, or 3. Both δ-nodes and μ-nodes influence the execution of subsequent instructions, but they convey different semantics:

- A μ-node updates once per *instruction*.
- A δ-node updates every *clock cycle*.

This subtle difference matters for WCET analysis, as illustrated in the rightmost part of Figure 6. The top-right timeline depicts the execution trace for a sequence of three RISC-V instructions found in real code (see Section IV): the first one (`SUBI`) modifies register `sp`, and the two following ones (`SW`) use `sp`. The first `SW` suffers from a two-cycle penalty because `sp` is modified by the first instruction. Because of this penalty the second `SW` does not suffer from any penalty.

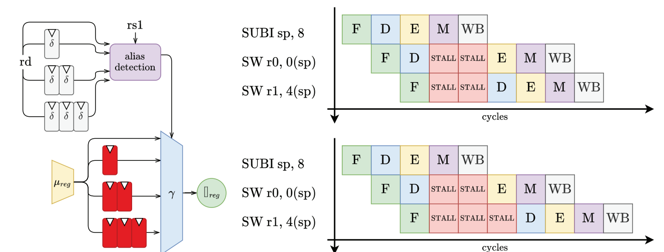

Fig. 6. Handling of data hazards (left), actual execution trace (top-right), and assumed execution trace (bottom-right).

Handling δ-nodes in PGA to produce the real number of timing penalties is challenging, since it needs to maintain a

history of the penalties incurred by previous instructions, as illustrated in the top-right part of the Figure. The solution we have adopted in a first approach is to detect aliases at the *instruction level* (i.e., handling δ-nodes as if they were μ-nodes). On the example from Figure 6, operating at the instruction-level results in the detection of an alias between the first and last instruction (as depicted in the bottom-right of the Figure), that does not occur in actual executions. This results in pessimistic estimates. The experimental study presented in IV demonstrates that the resulting pessimism is limited.

By construction, the single-issue in-order pipelines generated by SpecHLS enforce the *monotonicity* of the pipeline timing behavior [19]. This ensures that the processor is free from timing anomalies and, consequently, that the overapproximation of aliases by PGA results in safe WCET estimates.

IV. Experimental validation

A. Implementation and experimental setup

PGA is implemented as a pass within the SpecHLS flow, which uses the MLIR compilation framework[2]. The overall flow is depicted in Figure 7. A front-end first extracts the GSSA IR. The transformation passes of SpecHLS then introduce speculation as sketched out in Section II-B. SpecHLS and PGA are implemented using an MLIR/CIRCT dialect. The last step of the SpecHLS flow generates the transformed C/C++ code from the IR, which is then fed to an HLS backend.

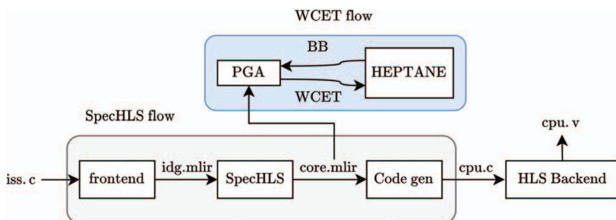

Fig. 7. PGA and WCET estimation flow.

PGA calculates the WCETs of individual basic blocks as described in Section III. The basic blocks' WCETs are fed into the open-source WCET estimation tool Heptane [20], which implements the IPET WCET estimation technique. The timing model for a 5-stage in-order 32-bit RISC-V core without forwarding is added to Heptane, and serves as a baseline for the evaluation of PGA on the same micro-architecture. PGA is evaluated using the subset of the Mälardalen benchmarks [21], with loop bound annotations given in the Heptane format.

The evaluation focuses on CPU cores that perform only integer operations and lack hardware support for floating-point operations. It also considers optional hardware support for multiplication and division, rather than their software-implemented equivalents. We chose to remove from the Mälardalen benchmarks all benchmarks that involve floating-point operations, to avoid the annotation of loop bounds for software-emulated floating-point operations.

[2]https://mlir.llvm.org/

B. Comparison of PGA with a handcrafted timing model

PGA automates the task of generating timing models, which enables productivity gains in the design of WCET estimation tools. This Section evaluates if the improved productivity gains come with a loss of precision for WCET estimates, both at the basic-block level and program level. We compare PGA with a handcrafted pipeline model for a in-order 32-bit RISC-V core with neither caches nor forwarding mechanisms.

Figure 8 compares PGA with the hand-coded pipeline analysis at the basic block level for all basic blocks of benchmarks. PGA, in most cases, successfully estimates the same WCET as the baseline. However, due to the pessimism of alias detection, PGA can, in some cases, assume an alias where none exists, and thus, produce a larger WCET. For instance, in the first BB of a function, the program decreases `sp` and then saves the contents of some registers through several `SW x, y(sp)` instructions. Two penalties happen for the first `SW` instruction since the previous instruction modifies `sp`. Due to these penalties, the remaining `SW` instruction does not suffer from any penalty; this is detected by the baseline analysis that operates at the cycle level, and not by PGA. Moreover, we did not observe any estimation strictly lower than the baseline, with the baseline considered as safe.

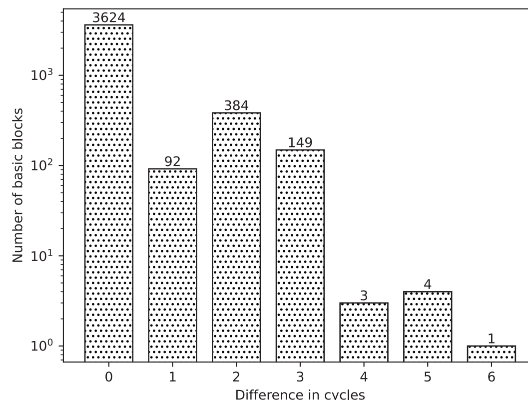

Fig. 8. Histogram of differences in cycles, between the baseline and PGA. PGA has 0.32 more cycles per BB, or 1.54% more cycles per BB on average.

As depicted in Figure 9, WCETs at the task-level are also slightly more pessimistic when using PGA (0.48% on average). One may notice that the pessimism at task-level is lower than at the BB-level. A manual investigation of difference have shown that the BBs with the higher pessimism (e.g. function start BBs) do not have a high number of occurrences on the critical path.

C. Exploration of core variants

SpecHLS automatically generates hardware descriptions of processors directly from an ISS. We perform Design Space Exploration (DSE) at the ISA-level by enabling ISA extensions, e.g., the M extension, which provides hardware support for multiplication and division. Exploration can also be performed at the micro-architectural level, for example, by varying pipeline depth or turning forwarding mechanisms on or off. We

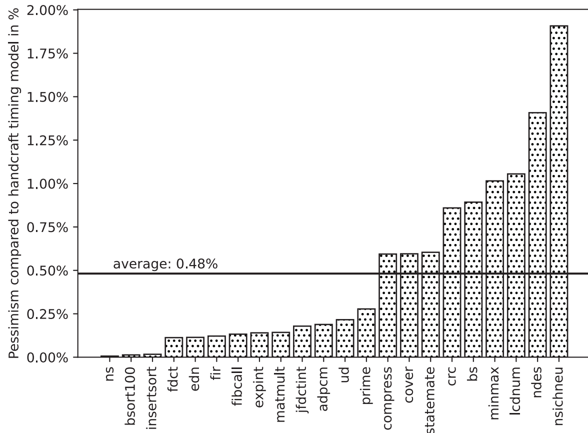

Fig. 9. Pessimism introduced by PGA, in %, at the task-level compared to the baseline: $((\mathrm{WCET_{PGA}}/\mathrm{WCET_{baseline}}) - 1) \times 100$.

have evaluated four different configurations, listed in Table I, all generated by SpecHLS and analyzed by PGA.

TABLE I
EVALUATED MICRO-ARCHITECTURE VARIANTS

Name	ISA	Characteristics
`cpu5`	RV32I	5-stage, no forwarding
`cpu4`	RV32I	4-stage, no forwarding
`cpu5f`	RV32I+mult	5-stage, multiplier, forwarding
`cpu5fm`	RV32IM	5-stage, mult/div, forwarding

Table II gives the benchmarks' WCETs for the four core variants listed in Table I. PGA was able to produce WCETs for all core variants. Note that the WCET values for different microarchitectures should not be compared, since the operating frequency is not taken into account here. This experiment demonstrates that PGA can analyze the impact of design choices on the benchmarks' WCETs seamlessly.

V. RELATED WORK AND DISCUSSION

The design of timing models for WCET estimation is well-known to be time-consuming and error-prone, lasting from weeks to months, depending on the complexity of the microarchitecture [10]. Prior work aiming at facilitating the construction of timing models has been conducted in the past. A semi-automatic technique to extract a timing model from VHDL is presented in [10]. The technique first prunes the VHDL code for size reduction and then generates an abstract timing model using abstract interpretation. Unlike our work, the timing model generation is not fully automatic. A technique for the reverse engineering of hardware designs is presented in [22]. This technique reconstructs the pipeline from a Chisel model, which limits its applicability to processors designed using Chisel. Moreover, the output of the technique is a formal model of the pipeline, which can be used for detecting timing anomalies, but is not directly usable for WCET analysis, unlike PGA. Gruin et al. present a framework for validating formal timing models through a systematic comparison of timed traces [23].

TABLE II
ESTIMATED WCETS PER VARIANT FOR EACH BENCHMARK (CYCLES)

Benchmarks	Cycles			
	`cpu5`	`cpu4`	`cpu5f`	`cpu5fm`
crc	359k	299k	275k	275k
cover	393k	356k	357k	357k
fibcall	1510	1413	1413	1413
statemate	20k	18k	18k	18k
ns	63k	52k	43k	43k
bsort100	1577k	1347k	1059k	1059k
adpcm	28226k	24984k	10746k	344k
jfdctint	252k	228k	9k	9k
ndes	248k	224k	214k	214k
edn	12553k	10897k	290k	290k
matmult	17966k	15573k	562k	562k
bs	452	396	376	376
fir	360k	314k	43k	8k
lcdnum	4k	3k	3k	3k
fdct	346k	300k	9k	9k
prime	3896k	3501k	2493k	48k
minmax	1990	1731	229	229
expint	1077k	951k	700k	117k
nsichneu	40k	35k	32k	32k
compress	766k	666k	484k	315k
insertsort	11k	10k	8k	8k
ud	700k	609k	149k	42k

In contrast to our work, it does not generate timing models; instead, it solely validates them. Amalou et al. propose an automatic derivation of timing models for WCET analysis using machine learning [24]. Model derivation is automatic, but since the models are trained using empirical data, there is no safety guarantee on the obtained models, which should not be used for the most time-critical software.

VI. CONCLUSION

In this paper, we have focused on the automatic extraction of WCET timing models from single-issue, in-order processor pipelines generated through HLS. By combining HLS's capability to explore a wide range of microarchitectural implementations with the efficiency of our analysis – which estimates WCET directly from the intermediate representation used in HLS flows – we achieve a significant gain in productivity with minimal pessimism (0.48% on average). Our ongoing work consists in improving the precision of alias detection. Moreover, as HLS evolves to support more complex cores, our future work will also aim at extending our approach to designs featuring caches, out-of-order execution, and dynamic branch prediction. Particular emphasis will be put on the detection and management of timing anomalies through formal analysis of both the extracted model and the PGA analysis.

ACKNOWLEDGMENT

This work is partly funded by the French National Research Agency (ANR), as part of the LOTR project.[3] (ANR-23-CE25-0016)

[3]https://lotr.irisa.fr/

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[2] S. Hahn and J. Reineke, "Design and analysis of sic: A provably timing-predictable pipelined processor core," *Real-Time Systems*, vol. 56, no. 2, pp. 207–245, 2020.

[3] J.-M. Gorius, S. Rokicki, and S. Derrien, "Design exploration of risc-v soft-cores through speculative high-level synthesis," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022, pp. 1–6.

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

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

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

[7] D. Leothaud, J.-M. Gorius, S. Rokicki, and S. Derrien, "Efficient design space exploration for dynamic & speculative high-level synthesis," in *34th International Conference on Field-Programmable Logic and Applications*, 2024.

[8] Y.-T. S. Li and S. 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*, 1995, pp. 88–98.

[9] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*. IEEE, 1999, pp. 12–21.

[10] M. Schlickling and M. Pister, "Semi-automatic derivation of timing models for wcet analysis," ser. LCTES '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 67–76. [Online]. Available: https://doi.org/10.1145/1755888.1755899

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

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

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

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

[15] Y. She, J. Liu, Y. Huang, R. C. Cheung, and H. Yan, "A speculative loop pipeline framework with accurate path modeling for high-level synthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 18, no. 2, pp. 1–33, 2025.

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

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

[18] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 63–74.

[19] S. Hahn and J. Reineke, "Design and analysis of sic: A provably timing-predictable pipelined processor core," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 469–481.

[20] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, pp. 8–1.

[21] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. Open Access Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010, pp. 136–146. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2010.136

[22] S. A. Bensaid, M. Asavoae, F. Thabet, and M. 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)*, 2022, pp. 1–8.

[23] A. Gruin, T. Carle, C. Rochange, and P. Sainrat, "Validation of Processor Timing Models Using Cycle-Accurate Timing Simulators," in *21th International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, ser. Open Access Series in Informatics (OASIcs), P. Wägemann, Ed., vol. 114. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 2:1–2:12. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2023.2

[24] A. N. Amalou, E. Fromont, and I. Puaut, "CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. V. Papadopoulos, Ed., vol. 262. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 7:1–7:20. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2023.7