Flexible Generation of Fast and Accurate Software Performance Simulators From Compact Processor Descriptions

Conrad Foik, Robert Kunzelmann, Daniel Mueller-Gritschneder, *Senior Member, IEEE*, and Ulf Schlichtmann, *Senior Member, IEEE*

***Abstract*—To find optimal solutions for modern embedded systems, designers frequently rely on the software performance simulators. These simulators combine an abstract functional description of a processor with a nonfunctional timing model to accurately estimate the processor's timing while maintaining high simulation speeds. However, current performance simulators either inflexibly target specific processors or sacrifice accuracy or simulation speed. This article presents a new approach to the software performance simulation, combining flexibility with highly accurate estimates and high simulation speed. A code generator converts a compact structural description of the target processor's pipeline into sets of timing constraints, describing the processor's instruction execution. Based on these, it generates corresponding scheduling functions and timing variables, representing the availability of the modeled pipeline. The performance estimator uses these components to approximate the processor's timing based on an instruction trace provided by an instruction set simulator. Results for the state-of-the-art CV32E40P and CVA6 RISC-V processors show an average relative error of 0.0015% and 3.88%, respectively, over a large set of benchmarks. Our approach reaches an average simulation speed of 24 and 15 million instructions per second (MIPS), respectively.**

***Index Terms*—Design space exploration (DSE), instruction set simulation, performance simulation, RISC-V.**

I. Introduction

TO PUSH toward increasingly strict performance and power requirements, modern embedded systems move to highly workload-tailored designs, such as application-specific instruction set processors (ASIPs). To find optimal solutions for ASIPs, early simulation-based software (SW) profiling and design space exploration (DSE) of the target microarchitecture are essential. Simulations thus play a crucial role during the development of ASIPs, resulting in a new demand for fast and accurate simulators. Even though the processor's register transfer level (RTL) representation could theoretically be used for the cycle-accurate simulations, these simulations are very slow and, thus, unsuitable for DSE and fast SW profiling. A more appropriate alternative is the use of so-called instruction set simulators (ISSs) [1], [2]. This type of simulator models the processor's behavior based on its instruction set architecture (ISA) description. As a result of this increased level of abstraction, an ISS is considerably faster than the RTL simulations.

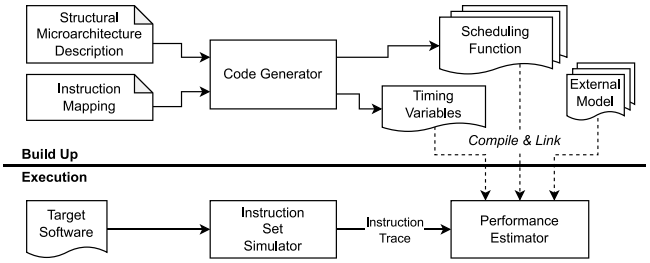

Fig. 1. Flexible performance simulation environment.

While an ISS correctly models the functional behavior of a processor, it does not capture microarchitectural aspects and is, therefore, incapable of delivering reliable estimates of the timing behavior. However, since convincing DSE and SW profiling require accurate performance estimates, several performance simulators have been proposed in [3] and [4]. These simulators typically combine the ISS approach with nonfunctional timing models of the target processor, offering a good combination of accurate performance estimates and high simulation speeds. A key challenge of DSE and SW profiling is the flexible adaptation of simulators to quickly evaluate many different processor variants, e.g., to explore custom instructions or pipeline variants. Thus, flexibility, along with high accuracy and simulation speeds, becomes a central requirement. However, the majority of performance simulators inflexibly target a specific processor variant. Earlier approaches focusing on flexibility either do not focus on the processor's timing behavior [5] or use computational heavy timing models sacrificing simulation speed [6].

In this article, we propose a new flexible approach to performance simulation illustrated in Fig. 1. It consists of two main components. A *code generator* translates a compact description of the target processor into an intermediate representation, based on which it creates timing variables

Manuscript received 7 August 2024; accepted 10 August 2024. This work was supported in part by the German Federal Ministry of Education and Research (BMBF) and ITEA within the Project GenerIoT under Contract 01IS22084G. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Conrad Foik.)*

The authors are with the Chair of Electronic Design Automation, Technical University of Munich, 80333 Munich, Germany, and also with the Institute of Computer Engineering, TU Wien, 1040 Wien, Austria (e-mail: conrad.foik@tum.de; daniel.mueller-gritschneder@tuwien.ac.at).

Digital Object Identifier 10.1109/TCAD.2024.3445255

and a set of instruction type-specific scheduling functions. A *performance estimator* utilizes these outputs to approximate the processor's performance based on a provided instruction trace. External models of complex dynamic microarchitecture aspects might further extend the estimator. The contributions of this article are the following.

1) As the key conceptual novelty, we introduce an intuitive and powerful constraint-based dependency graph as an intermediate representation of the target processor's timing behavior. It is directly derived from a compact and flexible pipeline description and is sufficiently powerful to model multi-issue concepts, unconsidered by the previous approaches.
2) We propose a fast and accurate performance estimation concept based on the completion-based timing variables, which represent the availability of the processor's components as well as the data dependencies and are updated once per instruction by simple algebraic scheduling functions. Both the timing variables and scheduling functions are directly generated from the above-mentioned intermediate representation and can be precompiled for ultrafast simulation speed.
3) Our approach is completely instruction trace-based and thus generically applicable regardless of the employed ISS.

Our approach offers several advantages. In terms of 1) *accuracy*, the constraint-based dependency graph used as an intermediate representation allows for an intuitive but highly accurate description of the target processor's timing behavior that provides timing estimates at the perinstruction level and not only for full program execution or program sections. It is capable of capturing single- as well as multi-issue concepts. Regarding 2) *simulation speed*, the constraint-based representation results in simple algebraic scheduling functions called once per instruction. This enables high simulation speeds, especially compared to the cycle-based simulators. Since the functions are independent of the instructions' execution order, they can be fully established and compiled before the simulation, further increasing the simulation speed. Concerning 3) *flexibility*, the compact nature of the code generator's input permits flexible and fast adaptation of the performance estimator to new microarchitecture versions, including multi-issue concepts. Further, the support of external models allows designers to quickly evaluate the effect of complex dynamic components, like caches, by exchanging the corresponding models. Finally, considering 4) *portability*, the proposed performance estimator can easily be connected to an existing functional ISS since it is fully trace-based. Additionally, pre-existing performance models might be incorporated as external models.

We demonstrate the advantages of our approach by applying it to the RISC-V processors CV32E40P [7] and CVA6 [8]. The experimental results show a consistent accuracy of above 99% for the low-power CV32E40P processor and 96% for the more complex CVA6 application class processor. The simulations reach average speeds of 24 and 15 million instructions per second (MIPS), respectively, which is comparable to manually created simulators and more than five times faster than the other generator-based approaches [6]. To highlight the ability of our approach to closely follow variations in the target SW, we also report the average absolute cycle error per instruction. We observe an average deviation of 0.0001 and 0.1 cycles per instruction, respectively. This highlights the capability of our approach to closely follow performance variations during the execution of the target SW, which is crucial for effective SW profiling.

II. Related Work

As RTL simulations are too slow for DSE, approaches, like [9] and [10], speed up these simulations. But they still require complete RTL designs, less suited for flexible explorations.

ISSs, like QEMU [1] and Spike [2], simulate the functional behavior of a processor at high speed but lack information about its microarchitecture. Therefore, ISSs usually roughly estimate the number of required clock cycles by multiplying the number of executed instructions by an assumed or obtained average cycle-per-instruction (CPI) ratio. However, this approach is generally too inaccurate and does not allow to identify performance variations over the cause of the target SW.

Statistical and machine-learning approaches can improve the accuracy of an ISS-based performance estimation [11], [12], [13]. However, these methods are black box models that offer little insights for DSE and require expensive training with an RTL implementation or accurate performance models as a reference.

The gem5 simulator [14] uses a detailed functional model of the target microarchitecture. This allows for accurate performance estimates but also drastically reduces simulation speed due to the low level of abstraction.

Performance simulators, such as [3] and [15], combine an ISS's abstract functional simulation with nonfunctional timing models of the target microarchitecture. The RISC-V VP [4] performance simulator also uses an ISS-based approach focusing on an RISC-V processor. These approaches can deliver accurate performance estimates while maintaining comparably high simulation speeds. However, to the best of our knowledge, neither of these approaches provides a generic method to derive the required timing models that would enable a flexible adaptation to new microarchitecture variants as provided by our approach.

The GVSoC simulator [5] offers configurability through a Python-based modular build setup but focuses on full-platform simulation instead of modeling the microarchitecture in detail.

The work in [6] proposes a generator-based approach to flexibly adapt the microarchitecture model. It describes the target processor using the HARMLESS domain-specific language (DSL) [16]. From this description, a finite-state machine (FSM)-based simulator is generated, which explicitly models the state of the pipeline for every simulated cycle. While highly flexible, this method results in low simulation speeds due to the computational overhead introduced by the FSM's frequent (cycle-based) updates and its large state space.

In contrast, our approach reduces the simulation overhead by utilizing simple algebraic scheduling functions applied only once per simulated instruction to update a small set of timing variables.

The ComCAS simulator [17] accelerates the FSM-based approach by compiling the simulator for a fixed target SW. Even though this increases the simulation speed, the generated simulator cannot be reused for another target SW. In comparison, our approach can be reused for any target SW compiled for the supported instruction set.

To the best of our knowledge, neither the aforementioned performance simulators, the GVSoC, nor the HARMLESS approach explicitly consider multi-issue concepts addressed by our approach.

III. Flexible Performance Simulation Environment

Fig. 1 presents our proposed flexible performance simulation environment. The two main components of the approach are a trace-based performance estimator, which is connected to an arbitrary ISS, and a corresponding code generator, which generates the microarchitecture-specific parts of the estimator.

The usage of the environment can be divided into two phases. During the *build up* of the simulation, the code generator flexibly adapts the performance estimator to a new target microarchitecture. For this purpose, the generator receives an abstract and simple-to-create input format consisting of a structural description of the target microarchitecture and a complementing instruction mapping. Based on this, the generator derives a series of timing constraints for each instruction type. Using as-soon-as-possible (ASAP) scheduling, the generator then transforms the constraints into a set of instruction type-specific scheduling functions, which model the timing behavior of instructions of the corresponding type. In addition, the code generator also provides timing variables, which represent the availability of the target microarchitecture's components. The generated outputs are then compiled and linked to the performance estimator. Besides the automatically generated outputs from the code generator, the presented environment also allows for the incorporation of *external models*. These manually created components typically describe complex timing behaviors that need to be determined dynamically, for example, the branch prediction schemes.

During the *execution* of the simulation, the performance estimator receives an instruction trace from an ISS, which executes the target SW. According to this trace, the estimator selects the appropriate scheduling function for each executed instruction and uses it to update the timing variables. If applicable, external models support this process by dynamically resolving more complex timing behaviors based on the additional information provided by the instruction trace and the models' internal states. It is worth noting that since the scheduling functions are only dependent on the instruction type, they are independent of the actual order of the executed instructions. As such, the estimator can use the already compiled scheduling functions to directly compute the timing estimates in an instruction-by-instruction manner instead of constructing complex trace-dependent functions during the run time. As mentioned above, this reduces the computational overhead during the simulation and thus increases the simulation speed.

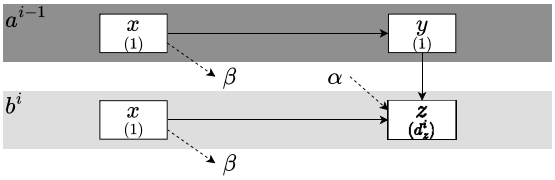

Fig. 2. Dependency graph expressing constraints during the execution of instructions a^{i-1} and b^i.

A. Constraint-Based Instruction Modeling

This section outlines the central concept of our approach based on a simple, generic example. It conceptually demonstrates how timing constraints can express the dependencies during the execution of instructions and, as such, model the timing behavior of a processor. It then shows how instruction type-specific scheduling functions are derived from the established timing constraints to allow for an effective calculation of the instructions' timing behavior.

As an example, we consider a sequence of two generic instructions, a^{i-1} and b^i. The two instructions are of *instruction type* a and b, respectively, where the instruction type (such as ADD, SUB, MUL) describes the functionality of the instruction. The index, i, indicates the order of execution. Thus, a^{i-1} is conceptually invoked before b^i.

To execute an instruction, a processor needs to complete a number of instruction type-specific *actions*, for example, utilizing a resource of the pipeline. With regards to the timing, it is reasonable to assume that a processor strives to complete these actions in an ASAP manner and ideally would complete them all at once. However, in reality, this is not possible due to dependencies between the actions, which constrain the execution of the instructions.

Fig. 2 illustrates this for the given example in the form of a *dependency graph*. Each node represents an action (of type x, y, or z), which the corresponding instruction must execute. The delay associated with each specific action is given in parentheses. Actions x and y have a constant delay of a single cycle, while the delay of z, d_z^i is dynamic and specific to the instruction b^i. The edges between the nodes represent dependencies between the actions.

Each of the presented dependencies corresponds directly to a timing constraint. For instance, the horizontal edge between x and z of b^i represents an *instruction-internal* dependency, implying that b^i must first complete x before starting z. Considering the delay of x, this yields the following timing constraint for x:

$$t^i_{z,\text{start}} \geq t^i_{x,\text{start}} + 1 \tag{1}$$

where $t^i_{x,\text{start}}$ and $t^i_{z,\text{start}}$ are the points in time when b^i can start x and z, respectively. However, for the purpose of this article,

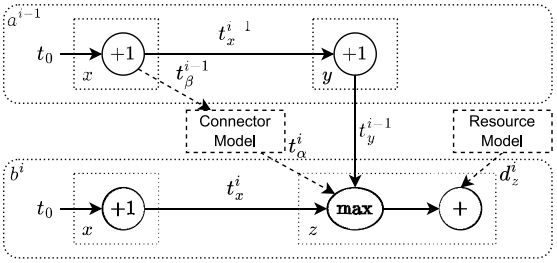

Fig. 3. Graph representation of the timing behavior computation over the instruction sequence.

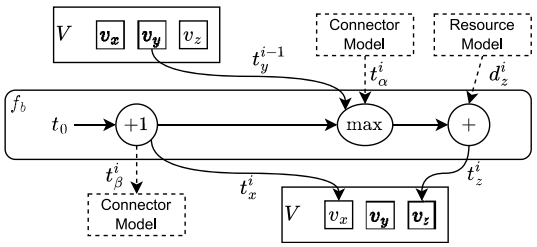

Fig. 4. Computation of the timing behavior of b^i based on scheduling function f_b and timing variable set V.

it is beneficial to adopt a *completion-oriented* representation and rephrase (1) to:

$$t_z^i \geq t_x^i + d_z^i \tag{2}$$

where t_x^i and t_z^i mark the points in time when b^i completes the respective actions.

Similarly, the vertical edge in Fig. 2 represents a *cross-instruction* dependency between the actions y of a^{i-1} and z of b^i. The corresponding timing constraint can be expressed as

$$t_z^i \geq t_y^{i-1} + d_z^i. \tag{3}$$

Finally, the dashed edges mark *dynamic* dependencies. They express a relation between an action and some token, typically describing a data dependency. For the given example, token β is provided by action x, while action z requires the availability of token α. It is further assumed that some relationship exists between α and β. However, this relationship is complex and must be dynamically resolved, and is therefore not explicitly expressed in Fig. 2.

The dependency on α results in the following constraint for z:

$$t_z^i \geq t_\alpha^i + d_z^i \tag{4}$$

where t_α^i marks the point in time when α is available for b^i.

Vice versa, the point in time, t_β^{i-1}, at which a^{i-1} makes β available is constrained by

$$t_\beta^{i-1} \geq t_x^{i-1}. \tag{5}$$

Based on these constraints, the completion time of the actions can be calculated. As mentioned above, it is reasonable to assume that each action will be completed as quickly as possible. Thus, by applying ASAP scheduling to the derived constraints, it is possible to express the completion time of each action as a simple single equation consisting of a max and an add operation. For instance, considering (2)–(4), the completion time of z in b^i can be expressed as

$$t_z^i = \max\left(t_x^i, t_y^{i-1}, t_\alpha^i\right) + d_z^i. \tag{6}$$

By establishing similar equations for each action and combining them, it is possible to compute the timing behavior of the considered sequence of instructions. Fig. 3 illustrates this calculation as a graph. It is assumed that each instruction starts its unconstrained actions (i.e., x) at some point in time, t_0, which marks the start of the processor's operation.

In addition, external models are incorporated into the computation. These manually created models can resolve the values dynamically during run time based on the information provided by the instruction trace and their initial state. For instance, a *resource model* provides the dynamic delay d_z^i. The relation between the tokens α and β is resolved by a *connector model*, which uses t_β^{i-1} as an input to determine t_α^i.

Conceptually, the computation presented in Fig. 3 would enable the calculation of the timing behavior of the considered instruction sequence as a whole. However, this would require knowledge of the execution order of the instructions. As stated above, it is, therefore, beneficial to split the computation into small instruction type-specific *scheduling functions*, which are independent of the instruction order.

Fig. 4 illustrates this approach. The scheduling function f_b contains all the operations corresponding to instruction b^i. However, it is important to note that f_b is valid for all the instructions of type b since the operations are directly related to the instruction's actions, which are identical for all the instructions of the same type.

A set of *timing variables*, V, is introduced to allow for a separate execution of the scheduling functions. For each action, x, y, and z, a corresponding timing variable, v_x, v_y, and v_z, is created, which acts as storage for the completion time of the action between the execution of two scheduling functions. For example, to compute the timing of instruction b^i, the scheduling function f_b reads the required completion time t_y^{i-1} from v_y. Vice versa, f_b updates the applicable timing variables, which a potential subsequent scheduling function might use.

Similarly to the timing variables, the scheduling function receives any dynamically derived values from the external models as input and, in turn, updates any applicable connector model.

B. Compact Microarchitecture Description

This section discusses the required input information of the generator flow. It consists of a structural pipeline description and a corresponding instruction mapping. While our tool currently uses a custom DSL, called CorePerfDSL [18], to compactly express the required information, it is important to note that any machine-readable microarchitecture description containing the required information could conceptually be used as an input format. In the remainder of this section, we, therefore, use a more convenient graphical representation instead of a textual description to discuss the required input.

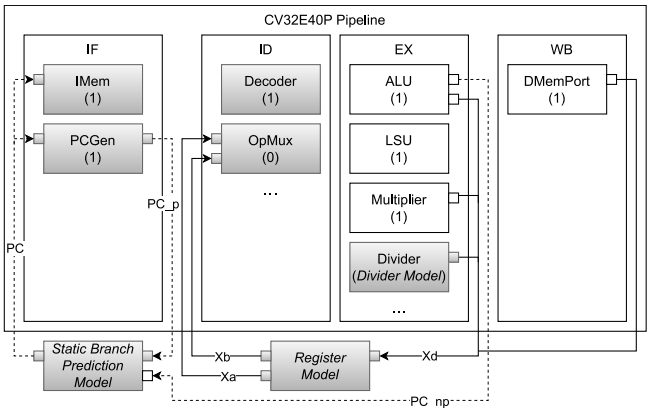

Fig. 5. Simplified structural description of the CV32E40P microarchitecture (Gray tinted components are the ones used by the DIV instruction type).

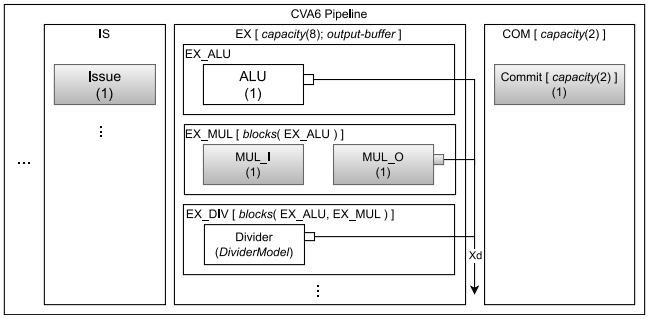

Fig. 6. Simplified structural description of the EX and COM stages of the CVA6 microarchitecture (Gray tinted components are the ones used by the MUL instruction type).

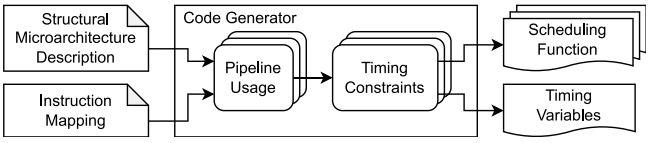

Fig. 7. Generation of scheduling functions and timing variables from an abstract structural microarchitecture description.

Fig. 5 shows a simplified structural description of the CV32E40P processor. The central element is the pipeline, which consists of the four stages of the processor. Each stage, in turn, contains its corresponding resources. For example, the arithmetic logic unit (ALU) is located in the execution stage, EX. The delay associated with each resource is shown in parentheses. While for most resources, the delay is static and represented as a fixed number of cycles, the delay is dynamically dependent for certain resources. For instance, the delay of the divider resource depends on the value of the involved operands. Dynamic delays are expressed as a reference to an external *resource model*, which provides the situation-specific delay during run time.

Connections depict data dependencies between the resources. For instance, a valid program counter (PC) is required to execute the resources in the instruction fetch stage, IF. The previous instruction generates PC, either during the IF stage or by using the ALU. As these kinds of dependencies across instructions typically are not static, external *connector models*, inserted between the resources, resolve them dynamically. For example, the CV32E40P applies the static branch prediction model to determine the PC dependency.

Both external resource and connector models are only referenced in the input description, as they are custom-made and directly incorporated into the performance estimator. The models' complexity varies since it depends on the corresponding processor component. However, it is important to note that the models are nonfunctional. For instance, the resource model of a cache only has to provide the delay of a memory access but not the corresponding value. Similarly, connector models only need to provide the time of availability of the dynamic dependencies.

A central multi-issue feature is the ability of stages to hold and operate on multiple instructions at a time. The *capacity* attribute expresses this capability as illustrated by Fig. 6 for the two final stages of the CVA6 processor. For instance, the EX stage implements a scoreboard buffer capable of holding up to eight instructions. This enables the stage to load and operate on a new instruction, even if prior instructions are not yet completed. Furthermore, once the EX stage has completed its operation on an instruction, it writes the results back to the scoreboard buffer. This frees the required resource for the following instruction, even if the current instruction cannot yet leave EX. The *output-buffer* attribute expresses this behavior.

To operate on multiple instructions, the EX stage uses subpipelining to arrange its resources. For example, we model the two-stage multiplier unit of the CVA6 as the EX_MUL subpipeline, containing two sequential resources. If a stage contains multiple subpipelines, the subpipelines might operate in parallel unless the *blocks* attribute is set accordingly. For example, while the EX_DIV subpipeline is active, EX_ALU and EX_MUL cannot start on a new instruction.

Finally, the CVA6's commit stage, COM, implements a resource capable of operating on two instructions simultaneously. Similar to the stages, the capacity attribute indicates this feature. As a result, the CVA6 is capable of committing up to two instructions in the same cycle.

An instruction mapping complements the structural description. It maps each supported instruction type to the required resources and connectors of the processor. Fig. 5 illustrates this by depicting all the corresponding components a DIV instruction needs as gray tinted. Similarly, Fig. 6 marks components required by instructions of type MUL.

C. Code Generator

The functionality of the code generator is outlined in Fig. 7. It first combines the structural microarchitecture description and the instruction mapping inputs, to establish for each supported instruction type a *pipeline usage* description. This description specifies which of the pipeline's components the corresponding instruction requires. Based on this, the generator creates a set of *timing constraints* for each instruction type. Finally, using the concept outlined in Section III-A, the generator derives the desired *outputs* from the established timing constraints.

$R_{\text{DIV}} = \{\text{IMem, PCGen, OpMux, Decoder, Divider}\}$;
$S_{\text{DIV}} = (\text{IF, ID, EX})$;
$R_{\text{DIV, IF}} = \{\text{IMem, PCGen}\}$; $R_{\text{DIV, ID}} = \{\text{OpMux, Decoder}\}$; $R_{\text{DIV, EX}} = \{\text{Divider}\}$;
$C_{\text{DIV, IMem, in}} = \{\text{PC}\}$; $C_{\text{DIV, PCGen, in}} = \{\text{PC}\}$; $C_{\text{DIV, PCGen, out}} = \{\text{PC\_p}\}$;
$C_{\text{DIV, OpMux, in}} = \{\text{Xa, Xb}\}$; $C_{\text{DIV, Divider, out}} = \{\text{Xd}\}$

Listing 1. Sets describing the usage of the CV32E40P pipeline by DIV instructions (empty connection sets are ignored).

1) Single-Issue Concepts: This section first presents the basic concepts of the generator, which apply to standard single-issue pipelines.

a) Pipeline Usage: The code generator first identifies which components of the microarchitecture are used by a specific instruction type. The instruction mapping itself already specifies for each instruction type, ι, a set of required resources, R_ι. Combining this information with the structure of the microarchitecture, the code generator derives for each instruction type a sequence of required stages, S_ι. Further, for each of these stages, s, a subset of resources, $R_{\iota,s}$, is created, containing all the required resources belonging to that specific stage.

In addition, for each required resource, r, two sets are derived, containing, respectively, all the required in-coming ($C_{\iota,r,\text{in}}$) and out-going ($C_{\iota,r,\text{out}}$) connections. Also, this information is directly given by the instruction mapping. Listing 1 lists all the sets and sequences describing the pipeline usage of the DIV instruction type for the CV32E40P microarchitecture as illustrated in Fig. 5.

b) Timing Constraints: Based on the derived sets of required components, a series of timing constraints can be derived for each instruction type. The first timing constraints result directly from the order in which an instruction progresses through the processor's pipeline. In other words, following the notation of Section III-A, these constraints describe *instruction-internal* dependencies for some instruction, ι^i.

An instruction proceeds through the pipeline by completing one stage before moving to the next. To complete a stage, s, every required resource, r, of that stage must have finished to operate on the considered instruction. Thus, the following constraint must hold for any instruction ι^i:

$$\forall s \in S_\iota \quad \forall r \in R_{\iota,s} : t^i_s \geq t^i_r \tag{7}$$

where t^i_s and t^i_r are the completion times of stage s and resource r, respectively.

Similarly, the resources of a stage can only start to operate on an instruction once the instruction has completed the previous stage. The only exception is the first required stage, s_0, as it has no preceding stage. Using a completion-oriented notation similar to (2), this requirement can be expressed as

$$\forall s \in S_\iota \setminus \{s_0\} \quad \forall r \in R_{\iota,s} : t^i_r \geq t^i_{s_-} + d^i_r \tag{8}$$

where s_- denotes the stage preceding s in S_ι, and d^i_r represents the delay associated with resource r for the considered instruction.

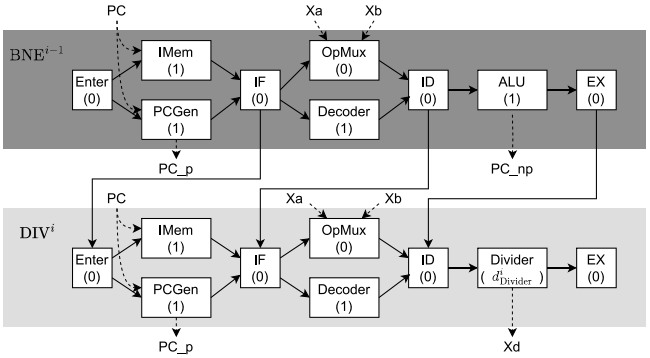

Fig. 8. Dependency graph for a DIV instruction preceded by a BNE instruction for the CV32E40P.

The resources of the first required stage, s_0, may start to operate as soon as the instruction enters the processor's pipeline at t^i_{Enter}. Thus, the equivalent of (8) for s_0 is

$$\forall r \in R_{\iota,s_0} : t^i_r \geq t^i_{\text{Enter}} + d^i_r. \tag{9}$$

Fig. 8 illustrates the constraints of (7) to (9) as instruction-internal dependencies in a dependency graph of an example instruction sequence. The actions represent the required resources (`IMem`, `PCGen`, `Decoder`, etc.) and the completion of the required stages (`IF`, `ID`, and `EX`). In addition, the action labeled `Enter` represents that the instruction enters the pipeline.

Aside from the instruction-internal dependencies, an instruction is also affected by other instructions currently in the pipeline. These *cross-instruction* dependencies cause additional timing constraints.

For an instruction to leave the current stage, s, the next required stage, s_+, must be available. For a typical single-issue pipeline, this means that the last instruction that employed s_+ must have completed the stage. The only exception to this rule is the last stage of S_ι, s_n, as it has no succeeding stage. Hence, the following constraint must hold:

$$\forall s \in S_\iota \setminus \{s_n\} : t^i_s \geq t^j_{s_+} \tag{10}$$

where j is the index of the last instruction using s_+.

With the same reasoning, the time at which the instruction enters the pipeline, t^i_{Enter}, can be constrained. Since instruction ι^i cannot start using the pipeline before its first required stage, s_0 is available, it must hold that

$$t^i_{\text{Enter}} \geq t^j_{s_0}. \tag{11}$$

It is worth noting that, most stages are used by every supported instruction type. For instance, for the CV32E40P, every instruction uses the `IF`, `ID`, and `EX` stages. This means that the final instruction using these stages must be the previous instruction, and hence, j can be simplified to $i-1$ for these stages.

For the given example sequence, Fig. 8 illustrates the constraints formulated in (10) and (11) as cross-instruction dependencies between the BNE and the DIV instruction.

Finally, as mentioned in Section III-B, additional connections between the resources may further constrain the

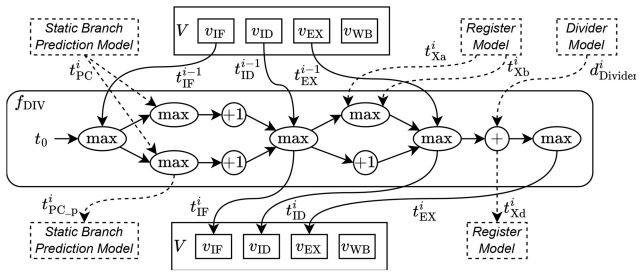

Fig. 9. Scheduling function for the DIV instruction type of the CV32E40P processor, f_{DIV}, and the set of timing variables, V.

```
void schedFunc_DIV(Estimator* estimator){
  uint64_t t_IF = estimator->variables.getIF();
  uint64_t t_PC = estimator->staBrPredModel.getPC();
  uint64_t n_1 = std::max({t_IF, t_PC});
  estimator->staBrPredModel.setPC_p(n_1);
  uint64_t n_2 = n_1 + 1;
  uint64_t t_ID = estimator->variables.getID();
  uint64_t n_3 = std::max({n_2 + 1, t_ID});
  estimator->variables.setIF(n_3); //...
}
```

Listing 2. Extract of the generated scheduling function for the DIV instruction type of the CV32E40P processor.

execution of the instruction. These connections typically describe the dynamic data dependencies.

Since a resource cannot start to operate on an instruction before every required in-coming connection, c_{in}, is available, the following constraint must hold:

$$\forall r \in R_\iota \quad \forall c_{\mathrm{in}} \in C_{r,\mathrm{in}} : t^i_r \geq t^i_{c_{\mathrm{in}}} + d^i_r \tag{12}$$

where $t^i_{c_{\mathrm{in}}}$ is the point in time at which the in-coming connection c_{in} is available for the instruction ι^i.

Vice versa, any out-going connection, c_{out}, provided by ι^i is first available once the corresponding resource has completed its operation

$$\forall r \in R_\iota \quad \forall c_{\mathrm{out}} \in C_{r,\mathrm{out}} : t^i_{c_{\mathrm{out}}} \geq t^i_r \tag{13}$$

where $t^i_{c_{\mathrm{out}}}$ is the time at which c_{out} is made available by ι^i.

The dynamic dependencies shown in Fig. 8 illustrate the constraints of (12) and (13), for the given example.

c) Output: Based on the established constraints for each instruction type ι, a corresponding scheduling function, f_ι, is derived by following the approach outlined in Section III-A. For example, Fig. 9 depicts the scheduling function f_{DIV} for the DIV instruction type of the CV32E40P microarchitecture. Listing 2 shows the corresponding generated code implementation after removing the redundant operations.

The complementing set of timing variables, V, provides the timing values of the previous scheduling functions required to solve cross-instruction dependencies. However, as shown in Fig. 8, in the case of a typical single-issue pipeline, cross-instruction dependencies are only caused by actions that mark the completion of a stage. It is, therefore, sufficient to create one timing variable per stage of the processor. Further, since the constraints formulated in (10) and (11) are only dependent on the *final* instruction using the stage (j), the timing variables can be implemented as simple single value variables, which

$S_{\mathrm{MUL}} = (..., \mathrm{IS}, \mathrm{EX}, \mathrm{COM})$;
$P_{\mathrm{MUL,\ EX}} = (\mathrm{EX\_MUL\_1}, \mathrm{EX\_MUL\_2})$;
$R_{\mathrm{MUL,\ EX\_MUL\_1}} = \{\mathrm{MUL\_I}\}$; $R_{\mathrm{MUL,\ EX\_MUL\_2}} = \{\mathrm{MUL\_O}\}$;
$B_{\mathrm{MUL,\ EX}} = \{\mathrm{EX\_DIV\_1}\}$;
$P_{\mathrm{MUL,\ COM}} = (\mathrm{COM\_1})$;
$R_{\mathrm{MUL,\ COM\_1}} = \{\mathrm{Commit}\}$;

Listing 3. Sets and sequence describing the usage of the EX and COM stages of the CVA6 by the MUL instruction type.

are overwritten by the following scheduling function that uses them.

Applicable external models provide the remaining required input values, as specified in the structural microarchitecture description (c.f. Fig. 5). For example, the delay of the `divider` resource, d^i_{Divider}, is determined by the `divider model`. The connector models the `static branch prediction model` and the `register model` provide the availability times of PC and the required operands (Xa and Xb), respectively. Vice versa, the scheduling function updates the connector models with the availability times of the following PC (PC_p) and the operation's result (Xd).

2) Multi-Issue Concepts: The constraints introduced in the previous section describe the timing behavior of fundamental pipeline concepts. This section demonstrates how these constraints can be modified and extended to cover some typical multi-issue concepts.

a) Pipeline Usage: Multi-issue pipelines typically arrange the resources of certain stages into subpipelines. To express the timing behavior of this feature, the pipeline usage description has to be extended. Instead of mapping a stage, s, to a set of resources, $R_{\iota,s}$, a sequence of substages, $P_{\iota,s}$ is derived. Stages that do not explicitly implement a subpipeline are represented as subpipelines with a single substage. Each substage, p, in $P_{\iota,s}$ is then mapped to a set of resources, $R_{\iota,p}$, following the structural microarchitecture description.

If s implements multiple parallel subpipelines, an additional set $B_{\iota,s}$ is defined. This set contains the last substages of each subpipeline that blocks the subpipeline required by instruction type ι. Listing 3 depicts the sets and sequence describing the usage of the EX and COM stages of the CVA6 by the MUL instruction type, as illustrated in Fig. 6.

b) Timing Constraints: A central multi-issue concept is the ability of stages to hold more than one instruction at a time. For the timing behavior, this means that the transition into a stage is no longer necessarily blocked by the previous instruction using that stage, but it depends on whether the stage has free capacities. Thus, assuming that stage s_+ can hold k instructions simultaneously, (10) is rephrased as

$$\forall s \in S_\iota \setminus \{s_n\} : t^i_s \geq t^{j_k}_{s_+} \tag{14}$$

where j_k is the index of the k^{th} final instruction using s_+. Equation (11) is modified accordingly. This expression can be further simplified if every instruction type uses s_+. In that case, j_k can be expressed as $i - k$.

The constraints for a design with substages can be derived based on the substage sequence as presented in Listing 3. Considering the transition of an instruction into a subpipelined stage, this transition is typically only possible once the first

substage of the required subpipeline is available. Using (10) as a reference, this new constraint can be expressed as

$$\forall s \in S_\iota \setminus \{s_n\} : t_s^i \geq t_{p_0'}^{j_k} \tag{15}$$

where p_0' is the first substage of the subsequent stage, s_+, and k denotes the number of instructions p_0' can handle simultaneously. For a substage, k is derived based on the capacity of its resources.

Following the same reasoning, transitioning into a stage with parallel subpipelines is only possible once no other subpipeline is blocking the required subpipeline. In other words, any blocking subpipeline must have been completed. Using the established set $B_{\iota,s}$, this constraint can be formulated as

$$\forall s \in S_\iota \setminus \{s_n\} \quad \forall p_n' \in B_{\iota,s_+} : t_s^i \geq t_{p_n'}^{j_k}. \tag{16}$$

Constraints for t_{Enter}^i, corresponding to (15) and (16), are derived using (11) as a reference.

In addition, when entering a subpipelined stage, s, an instruction only enables the resources of the first substage. Thus (8) is rephrased to

$$\forall s \in S_\iota \setminus \{s_0\} \quad \forall r \in R_{\iota,p_0} : t_r^i \geq t_{s_-}^i + d_r^i \tag{17}$$

where p_0 denotes the first element of $P_{\iota,s}$. To also express a similar constraint for s_0, (9) is modified accordingly.

Once an instruction has entered a substage, it transitions from substage to substage in a standard pipeline manner. As such, the instruction-internal dependencies between the substages and their resources correspond directly to the instruction-internal dependencies of a typical single-issue pipeline. The corresponding timing constraints can thus be expressed by reformulating (7) and (8) as

$$\forall s \in S_\iota \quad \forall p \in P_{\iota,s} \quad \forall r \in R_{\iota,p} : t_p^i \geq t_r^i \tag{18}$$

and

$$\forall s \in S_\iota \quad \forall p \in P_{\iota,s} \setminus \{p_0\} \quad \forall r \in R_{\iota,p} : t_r^i \geq t_{p_-}^i + d_r^i \tag{19}$$

respectively.

Furthermore, since an instruction can only leave a substage once the next substage is available, cross-instruction dependencies arise between the substages, similar to the dependencies between the stages. As such, these dependencies can be expressed as

$$\forall s \in S_\iota \quad \forall p \in P_{\iota,s} \setminus \{p_n\} : t_p^i \geq t_{p_+}^{j_k} \tag{20}$$

similar to (14).

An instruction completes the subpipeline of a stage, s, once it leaves the final substage, p_n, at $t_{p_n}^i$. For most standard stages, leaving p_n is only possible if the instruction simultaneously leaves the stage. In other words, for these stages, t_s^i and $t_{p_n}^i$ are identical and hence share the same constraints.

However, more advanced stages enable instructions to leave the subpipeline before the transition to the next stage is possible. We refer to this stage feature as an *output buffer*. An example of such a stage is the EX stage of the CVA6 processor (c.f. Fig. 6), which can store the results of an instruction in the processor's scoreboard if it cannot directly commit the instruction, allowing it to operate on the next instruction.

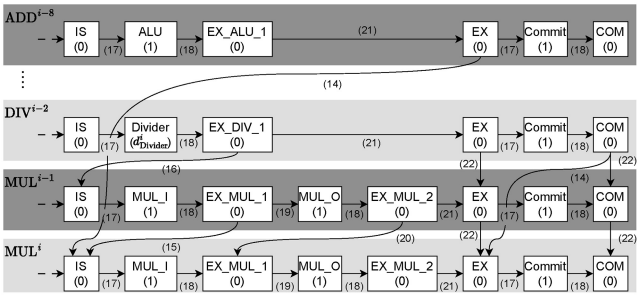

Fig. 10. Dependency graph for an instruction sequence in the *EX* and *COM* stages of the CVA6. Numbers on the edges refer to the equations of the corresponding generated constraints. (Dynamic dependencies omitted for readability).

Thus, depending on its specific output characteristics, the completion time of a subpipelined stage s is constraint by

$$\forall s \in S_\iota : \begin{cases} t_s^i \geq t_{p_n}^i, & \text{if } s \text{ has output buffer} \\ t_s^i = t_{p_n}^i, & \text{otherwise.} \end{cases} \tag{21}$$

Finally, while multi-issue stages can operate on multiple instructions in parallel, embedded processors typically ensure that instructions leave the stage in the same order in which they were invoked. Thus, an instruction can only leave a stage once the previous instruction is also ready to leave the stage. The following constraint expresses this behavior:

$$\forall s \in S_\iota : t_s^i \geq t_s^j. \tag{22}$$

Fig. 10 illustrates the above-derived constraints for an example sequence of instructions, using the EX and COM stages of the CVA6 processor. For readability, any dynamic dependencies that might apply are omitted.

As shown in the graph, the EX stage has a capacity of eight instructions. Further, as mentioned above, the stage implements an output buffer, which effectively decouples the completion of its subpipelines (e.g., action EX_MUL_2) from the completion of the stage itself (action EX). The graph also illustrates the effect of the subpipelined multiplier unit and how the DIV^{i-2} instruction blocks the subsequent MUL^{i-1} instruction from entering the EX stage until its operation is completed.

The COM stage has a capacity of two instructions. Since it does not implement an output buffer, the defined subpipeline of the stage (c.f. COM_1 in Listing 3) is completed simultaneously with the stage itself. Therefore, both incidents are represented as one action, COM.

c) Output: Based on the derived constraints, the scheduling functions for each supported instruction type can be generated similarly to a typical single-issue processor. Fig. 11 illustrates this for the MUL instruction type of the CVA6.

However, the generation of the set of timing variables, V, has to be modified slightly. As shown in Fig. 10, besides stages, also substages can cause the cross-instruction dependencies. Thus, in addition to the timing variables corresponding to the stages, timing variables for each substage of the processor have to be established. Further, for components with an instruction capacity, k, larger than one, cross-instruction dependencies are no longer dependent on the final instruction (j) but the k^{th} last

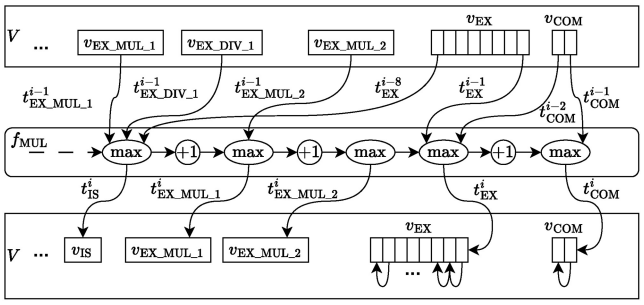

Fig. 11. Simplified representation of the scheduling function (EX and COM stage) for the MUL instruction type of the CVA6, f_{MUL}, and the corresponding timing variables.

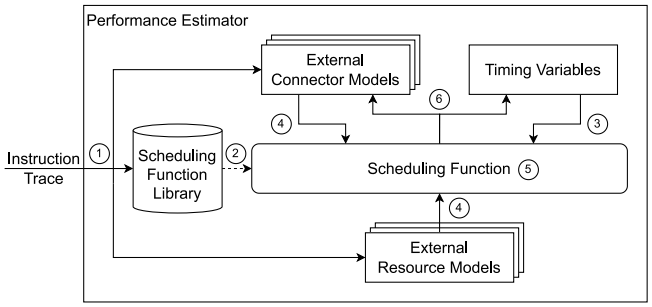

Fig. 12. Instruction trace-based performance estimator.

instruction (j_k). As such, the corresponding timing variables must be capable of holding the time values provided by the previous k scheduling functions. Implementing these timing variables as k-element buffers as illustrated by v_{EX} and v_{COM} in Fig. 11, fulfills this requirement.

D. Performance Estimator

Once the code generator has created the scheduling functions and timing variables for the target microarchitecture during the simulation build-up, they are compiled and linked to the performance estimator. In addition, any manually generated external models referenced in the structural microarchitecture description are also linked to the performance estimator. During the execution of the simulation, the performance estimator uses the established library of scheduling functions to update the timing variables according to the order of the executed instructions of the target SW. The external models support this process by dynamically resolving complex timing behaviors based on the additional information provided in the instruction trace.

Fig. 12 outlines the functionality of the performance estimator during the simulation execution. The performance estimator is invoked when the ISS executing the target SW updates the instruction trace as follows. ① Whether the ISS provides these updates continuously for every instruction or transmits them block-wise is conceptually irrelevant to the estimator. For each new instruction the performance estimator receives, it first identifies the instruction type, ι. Based on this information, it selects the corresponding scheduling function from the library. ② In preparation for the function's execution, it then reads the current values of all the relevant timing variables and passes them as inputs to the scheduling function. ③ If applicable, the estimator requests any additional input value from the corresponding external model. ④ Upon such a request, the external model reads the appropriate information from the instruction trace to compute the requested value and update its internal state. For example, a resource model representing an instruction cache typically requires the instruction's PC to establish whether the access would result in a hit or a miss. Based on the outcome, it returns the respective delay to the scheduling function. In case of a miss, it likely also updates its internal state to express that the instruction has been loaded to the cache, which will thus result in a cache hit on the next access. Once all the required inputs are provided, the performance estimator executes the selected scheduling function. ⑤ After the execution, the estimator updates the appropriate timing variables and, if applicable, any associated external connector models with the outputs provided by the scheduling function. ⑥ The estimator might then start to operate on the next instruction.

E. Scope and Current Limitations

Our approach targets the performance behavior of single-core processors. However, conceptually, the presented method can also be applied to the multicore processors by duplicating the performance estimator. If the cores provide individual instruction traces, they could be represented by separate performance estimators synchronized through a shared external resource model of the memory system.

Our approach is currently limited to the in-order processors. The implication of handling the instruction in order of the ISS trace is that the timing of an instruction can only be affected by the previous instructions but not by subsequent ones. This, however, does not apply to out-of-order processors. As a potential solution for the future work, the performance estimator could be extended to speculatively handle the instructions in trace order and retro-actively correct erroneous assumptions.

IV. Experimental Results

We demonstrate the flexibility of our approach and its capability of generating accurate and fast performance simulators by applying it to the two open-source state-of-the-art RISC-V processors. The CV32E40P [7] is a 32-bit single-issue four-stage processor with a static branch prediction scheme suitable for low-power use cases. In contrast, the CVA6 [8] is a 64-bit application class processor. While it is classified as a single-issue processor because its issue stage, IS, can only issue one instruction per cycle (cf. Fig. 6), it implements several multi-issue concepts, such as a scoreboard and dual-commits. It features a dynamic branch prediction scheme as well as L1 instruction and data caches.

A. Setup and Flexibility

Following the presented approach, we generate scheduling functions and timing variables for the two target processors. Table I shows the parameters of the input description, external models, and generated items. Due to the limited documentation

TABLE I
PARAMETERS DURING ESTIMATOR SETUP

Processor	Input Description		Ext. Models	Num. Generated Items	
	Structural [LOC]	Total [LOC]	(combined) [LOC]	Scheduling Functions	Timing Variables
CV32E40P	72	177	219	52	4
CVA6	150	248	803	66	19

of the third-party processors, considerably more effort was required to extract microarchitectural details from the RTL than to generate the required input description and external models. However, the small number of lines of code (LOC) indicates that designers familiar with the target processor should be able to establish these files quickly. Once established, modifications to evaluate the processor variants require little effort.

Our simulation environment contains the proposed performance estimator and an ISS, called ETISS [19], which executes the target SW. An additional RTL simulation provides a cycle-accurate baseline to evaluate the estimator's accuracy. It uses Verilator [10] representations of the target processors embedded in a simple SystemC environment and executes the same target SW as the ISS. All the simulations run on a Linux computer with an Intel i5-7500 processor. We use the *Embench* benchmark suite [20] as the target SW and compile it with *gcc* for the RV32IM and RV64IM instruction sets, respectively.

B. Accuracy

Table II lists the observed cycle counts from the RTL simulation and the corresponding estimates for both the target processors. Our approach can consistently provide accurate estimates, even though the CPI varies over the benchmarks. This also highlights the benefits of our approach compared to the pure ISS-based simulations, which assume a fixed CPI.

For the CV32E40P processor, the performance estimator delivers highly accurate cycle counts with a negligible relative error of maximal 0.0035% and 0.0015% on average.

For the CVA6, the error rate is on average 3.88%. We observe a maximal error of 8.91%. The larger error stems from two effects as follows.

1) The CVA6 implements more complex microarchitectural components, such as an instruction fetch queue and the more advanced load-store unit. Some timing deviations for the CVA6 are caused by unidentified details of the RTL behavior, resulting in missing timing constraints or imperfect external models. However, once identified, our approach can conceptually cover these missing details by improving the pipeline model.
2) Part of the deviations arise from effects that cannot be captured by the current version of our approach. One such effect is caused by unintentionally fetched instructions due to branch mispredictions. While these instructions are never committed, they still alter the state of processor components, such as the instruction cache. However, an ISS only simulates instructions that execute to completion. Unintentionally fetched instructions are thus not part of the provided instruction trace, and the performance estimator cannot consider their effect on the processor's timing.

Even though the error rates for the CVA6 are higher compared to the CV32E40P, the results show a clear improvement over a pure ISS-based simulation, both when using the naive assumption that the CPI $= 1$ and average CPI as well as over the existing approaches as presented in Section IV-D.

C. Simulation Speed

Table III shows the combined execution times of the performance estimator and the ISS for both the target processors, running the benchmarks with varying iterations. The columns marked with *short* list the execution times corresponding to the simulations that yield the results presented in Table II. For these simulations, the required execution times are similar for both the target processors, averaging 0.73 and 0.75 s, respectively. Considering the number of simulated instructions, the execution times correspond to an average simulation speed of roughly 4 MIPS.

While these simulation speeds are already considerably faster than an RTL simulation, they do not represent the performance estimator's performance. Instead, they are primarily caused by the ISS. The ISS applies binary translation to the target SW and caches the result for reuse. This imposes a computational overhead at the beginning of the simulation. However, the overhead diminishes for each iteration of the target SW. We compensate for the imposed overhead by increasing the number of iterations for the benchmarks. To better compare our results to the other approaches, we increase the number of instructions to a similar amount as used during the evaluation of the RISC-V VP [4]. The columns in Table III marked *long* present the corresponding results. For the CV32E40P, the performance estimator achieves an average simulation speed of roughly 24 MIPS. Due to the increased complexity of the utilized scheduling functions and the required external models, the average simulation speed for the CVA6 settles at around 15 MIPS.

D. Comparison

We relate our approach to the other existing methods for performance simulation based on the results presented above. Table IV presents an overview of this comparison. As discussed above, our approach provides significantly more accurate performance estimates than a pure ISS-based approach. At the same time, our approach utilizes a nonfunctional timing model, which allows it to run considerably faster than the RTL simulations.

Similar to our proposal, the HARMLESS approach is generator-based and thus offers high flexibility. However, the complexity of its FSM-based model results in comparatively low simulation speeds. The performance of the approach is evaluated in [6] by modeling a simplified artificial six-stage single-issue processor supporting a set of ten instructions. The resulting FSM requires 43 200 states, and the corresponding simulator achieves a simulation speed of roughly 4 MIPS over a simulation of 100 million instructions. Similar results

TABLE II
OBSERVED AND ESTIMATED CYCLE COUNTS FOR THE CV32E40P AND CVA6 PROCESSORS OVER THE EMBENCH BENCHMARK SUITE

Benchmark	CV32E40P								CVA6							
	Num. Instr.	RTL Cycles	CPI	ISS, CPI=1 Error	ISS, CPI=1.29 Error	Perf.Sim. Cycles	CPI	Error	Num. Instr.	RTL Cycles	CPI	ISS, CPI=1 Error	ISS, CPI=1.45 Error	Perf.Sim. Cycles	CPI	Error
aha-mont64	4,534,817	5,313,414	1.17	14.7%	10.1%	5,313,364	1.17	0.000941%	1,918,426	2,643,667	1.38	27.4%	5.22%	2,634,848	1.37	0.334%
crc32	4,183,376	4,881,112	1.17	14.3%	10.6%	4,881,062	1.17	0.00102%	4,184,058	6,106,933	1.46	31.5%	0.656%	5,932,284	1.42	2.86%
cubic	6,854,333	8,189,299	1.19	16.3%	7.97%	8,189,249	1.19	0.000611%	3,743,102	5,364,025	1.43	30.2%	1.18%	5,594,859	1.49	4.30%
edn	3,447,366	4,132,445	1.20	16.6%	7.61%	4,132,395	1.20	0.00121%	3,367,939	4,141,912	1.23	18.7%	17.9%	3,818,705	1.13	7.80%
huffbench	2,302,258	2,963,665	1.29	22.3%	0.211%	2,963,615	1.29	0.00169%	2,297,463	3,366,148	1.47	31.7%	1.03%	3,343,952	1.46	0.659%
matmult-int	3,602,152	4,377,284	1.22	17.7%	6.16%	4,377,234	1.22	0.00114%	4,334,701	5,252,905	1.21	17.5%	19.7%	4,878,131	1.13	7.13%
minver	2,509,246	3,581,566	1.43	29.9%	9.62%	3,581,516	1.43	0.00140%	2,287,078	3,572,886	1.56	36.0%	7.18%	3,488,838	1.53	2.35%
nbody	3,177,393	3,783,510	1.19	16.0%	8.33%	3,783,460	1.19	0.00132%	2,248,434	3,173,887	1.41	29.2%	2.72%	3,146,675	1.40	0.857%
nettle-aes	4,406,553	4,628,360	1.05	4.79%	22.8%	4,628,310	1.05	0.00108%	5,333,799	5,509,595	1.03	3.19%	40.4%	5,478,349	1.03	0.567%
nettle-sha256	3,965,640	3,991,463	1.01	0.647%	28.2%	3,991,413	1.01	0.00125%	4,022,440	4,192,764	1.04	4.06%	39.1%	4,545,013	1.13	8.40%
nsichneu	2,241,955	3,619,320	1.61	38.1%	20.1%	3,619,270	1.61	0.00138%	2,566,060	6,248,656	2.44	58.9%	40.5%	6,556,321	2.56	4.92%
picojpeg	3,595,305	4,215,381	1.17	14.7%	10.0%	4,215,331	1.17	0.00119%	3,642,847	5,029,281	1.38	27.6%	5.03%	4,752,829	1.30	5.50%
qrduino	2,822,182	3,487,235	1.24	19.1%	4.40%	3,487,185	1.24	0.00143%	3,088,750	5,073,925	1.64	39.1%	11.7%	4,711,397	1.53	7.14%
sglib-combined	2,344,372	3,290,199	1.40	28.7%	8.08%	3,290,149	1.40	0.00152%	2,394,797	3,814,301	1.59	37.2%	8.96%	3,963,025	1.65	3.90%
slre	2,376,079	2,920,434	1.23	18.6%	4.96%	2,920,384	1.23	0.00171%	2,470,357	3,535,047	1.43	30.1%	1.33%	3,446,210	1.40	2.51%
st	3,969,153	4,905,662	1.24	19.1%	4.37%	4,905,612	1.24	0.00102%	2,728,276	3,987,953	1.46	31.6%	0.801%	3,912,536	1.43	1.89%
statemate	2,098,097	2,305,566	1.10	9.00%	17.4%	2,305,516	1.10	0.00217%	1,876,459	2,431,164	1.30	22.8%	11.9%	2,395,081	1.28	1.48%
ud	923,462	2,034,580	2.20	54.6%	41.4%	2,034,530	2.20	0.00246%	1,384,912	2,279,751	1.65	39.3%	11.9%	2,076,639	1.50	8.91%
wikisort	1,056,343	1,413,159	1.34	25.2%	3.57%	1,413,109	1.34	0.00354%	924,018	1,366,640	1.48	32.4%	1.96%	1,336,376	1.45	2.21%
Average	3,179,478	3,896,508	1.29	20.0%	11.9%	3,896,458	1.29	0.00148%	2,884,943	4,057,444	1.45	28.9%	12.1%	4,000,635	1.43	3.88%

TABLE III
SIMULATION SPEED OF ISS WITH PERFORMANCE ESTIMATOR FOR THE EMBENCH BENCHMARK SUITE

Benchmark	CV32E40P						CVA6					
	Short			Long			Short			Long		
	Num.Instr.	Time [s]	MIPS	Num.Instr.	Time [s]	MIPS	Num.Instr.	Time [s]	MIPS	Num.Instr.	Time [s]	MIPS
aha-mont64	4,534,817	0.609	7.45	1,449,845,025	46.3	31.3	1,918,426	0.566	3.39	612,508,352	31.6	19.4
crc32	4,183,376	0.856	4.89	1,337,427,927	52.8	25.3	4,184,058	0.728	5.75	1,337,537,069	73.1	18.3
cubic	6,854,333	1.41	4.87	2,191,978,814	114	19.3	3,743,102	1.02	3.68	1,196,292,703	91.6	13.1
edn	3,447,366	0.790	4.36	1,101,272,788	35.5	31.1	3,367,939	0.676	4.98	1,075,743,860	59.5	18.1
huffbench	2,302,258	0.575	4.00	735,170,945	25.9	28.4	2,297,463	0.643	3.57	733,660,789	42.2	17.4
matmult-int	3,602,152	0.563	6.39	1,147,821,977	36.8	31.2	4,334,701	0.698	6.21	1,382,120,584	76.7	18.0
minver	2,509,246	0.704	3.56	800,982,197	38.6	20.8	2,287,078	0.772	2.96	729,764,984	54.9	13.3
nbody	3,177,393	0.747	4.25	1,013,015,887	46.1	22.0	2,248,434	0.715	3.14	717,319,808	50.0	14.3
nettle-aes	4,406,553	0.694	6.35	1,407,427,888	57.9	24.3	5,333,799	0.867	6.15	1,704,525,580	105	16.2
nettle-sha256	3,965,640	0.813	4.88	1,267,684,141	63.7	19.9	4,022,440	0.864	4.66	1,285,772,416	96.5	13.3
nsichneu	2,241,955	0.758	2.96	716,150,558	42.6	16.8	2,566,060	0.932	2.75	819,751,870	74.6	11.0
picojpeg	3,595,305	0.807	4.46	1,149,107,079	48.2	23.8	3,642,847	0.998	3.65	1,164,269,479	76.0	15.3
qrduino	2,822,182	0.836	3.37	901,802,462	36.4	24.8	3,088,750	0.785	3.93	986,996,400	59.3	16.6
sglib-combined	2,344,372	0.600	3.91	747,919,467	30.7	24.4	2,394,797	0.670	3.58	764,057,381	49.3	15.5
slre	2,376,079	0.630	3.77	759,091,930	33.6	22.6	2,470,357	0.772	3.20	789,153,068	53.7	14.7
st	3,969,153	0.735	5.40	1,268,783,419	56.6	22.4	2,728,276	0.714	3.82	871,611,864	58.9	14.8
statemate	2,098,097	0.613	3.42	669,964,154	31.2	21.5	1,876,459	0.632	2.97	598,946,208	38.9	15.4
ud	923,462	0.528	1.75	294,185,267	14.1	20.9	1,384,912	0.597	2.32	441,749,101	31.1	14.2
wikisort	1,056,343	0.652	1.62	334,246,419	15.4	21.6	924,018	0.649	1.42	293,043,169	19.6	14.9
Average	3,179,478	0.733	4.30	1,015,467,281	43.5	23.8	2,884,943	0.752	3.80	921,306,562	60.1	15.5

TABLE IV
OVERVIEW OF APPROACHES TO SOFTWARE PERFORMANCE SIMULATION

Simulator	Target	Accuracy	MIPS	Flexibility of timing model
ISS + Perf. Est. (*ours*)	CV32E40P	>99%	24	Generateable
	CVA6	96%	15	Generateable
ISS	CV32E40P	80%	104	Fixed
	CVA6	71%	106	Fixed
Verilator (RTL)	CV32E40P	100%	0.2	Fixed
	CVA6	100%	0.01	Fixed
HARMLESS [6] [16]	e200z1	95%	4	Generateable
GVSoC [5]	PULP CL	90%	25	Configurable
RISCV-VP [4]	HiFive1	95%	27	Fixed

are reported for the simulation of the four-stage single-issue e200z1 processor [16]. In comparison, our approach is more than five times faster, considering a simulation of similar complexity and length.

The GVSoC [5] uses manually created models but offers some level of flexibility through a configurable simulation setup that enables the quick selection of models as well as the adaptation of architectural parameters, such as the latency of instructions. Experimental results indicate that the GVSoC achieves higher simulation speeds than our approach but has a reduced accuracy. However, a direct comparison between the GVSoC and our approach is difficult since the GVSoC primarily focuses on simulating the multicore platforms.

The authors of the RISC-V VP [4] evaluate their approach by modeling the HiFive1 board and executing the Embench benchmark suite. The HiFive1 board implements the 32-bit single-issue four-stage SiFive E31 RISC-V processor with dynamic branch prediction and data and instruction caches [21]. As such, the E31 provides more features than the CV32E40P but is less complex than the CVA6. Even though the RISC-V VP achieves a higher average simulation performance, the simulation speeds of our approach are in a similar range. Likewise, the simulators achieve similar levels of accuracy, with our approach resulting in a slightly lower error rate for designs with comparable complexity. However, while the RISC-VP is manually created, our approach can be generated and thus flexibly adapted to new microarchitecture variants.

E. Average Absolute Cycle Error Per Instruction

A major advantage of the proposed approach is that it does not only supply average timing values over program sections or full program execution but the cycle estimates at the instruction level. To illustrate this, we also provide the error of each individual instruction as the deviation between the observed and estimated number of cycles added by that instruction. Fig. 13 presents the distribution of this deviation over the

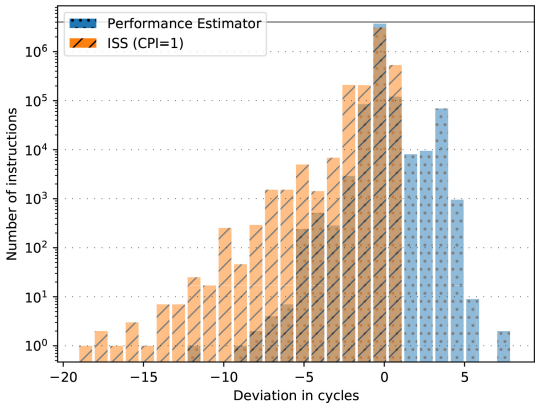

Fig. 13. Distribution of cycle deviation over the nettle-sha256 benchmark for CVA6 (solid line marks total number of instructions).

TABLE V
AVERAGE ABSOLUTE ERROR PER INSTRUCTION (IN CYCLES)

Benchmark	CV32E40P		CVA6	
	ISS (CPI=1)	Perf. Est.	ISS (CPI=1)	Perf. Est.
aha-mont64	0.17	0.000070	0.38	0.0066
crc32	0.17	0.000080	0.71	0.042
cubic	0.19	0.000010	0.53	0.16
edn	0.20	0.00010	0.51	0.27
huffbench	0.29	0.00015	0.88	0.12
matmult-int	0.22	0.000090	0.57	0.099
minver	0.43	0.00013	0.71	0.073
nbody	0.19	0.00011	0.50	0.045
nettle-aes	0.050	0.000080	0.33	0.013
nettle-sha256	0.0066	0.000080	0.31	0.13
nsichneu	0.61	0.00015	1.44	0.40
picojpeg	0.17	0.000090	0.70	0.17
qrduino	0.24	0.00012	0.90	0.13
sglib-combined	0.40	0.00014	0.91	0.26
slre	0.23	0.00014	0.65	0.12
st	0.24	0.000080	0.60	0.079
statemate	0.099	0.00016	0.97	0.045
ud	1.2	0.00036	0.95	0.20
wikisort	0.34	0.00032	0.78	0.11
Average	0.29	0.00013	0.70	0.13

nettle-sha256 benchmark for the CVA6, using a logarithmic scale to account for the outliers. Even though the benchmark has an observed overall CPI close to one (c.f. Table II), the purely ISS-based approach inserts an absolute error of one or more cycles to 24% of the simulated instructions. On average, the ISS mispredicts each instruction by about 0.3 cycles. In contrast, our approach correctly predicts 93% of all the instructions, with an average absolute error of roughly 0.1 cycles per instruction. This illustrates that reporting accuracy over full program execution can include compensation effects due to accumulating negative and positive errors.

Thus, even though the ISS reports a higher accuracy in terms of the overall cycle count for the nettle-sha256 benchmark, our approach more accurately represents variations in the processor's performance during the program. Table V lists the average absolute error per instruction for all the benchmarks, showing consistently better results for our approach than for an ISS.

V. CONCLUSION

This article presents a new flexible approach to SW performance simulation, supporting quick adaptation to new microarchitecture variants, based on the code generation.

We apply our approach to the state-of-the-art low-power and application class RISC-V processors and achieve highly accurate performance estimates with an average error rate of 0.0015% and 3.88%, respectively. Our approach reaches high simulation speeds comparable to manually created simulators.

REFERENCES

[1] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, p. 41.
[2] "Spike RISC-V ISA simulator." Accessed: May 5, 2022. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim
[3] I. Böhm, B. Franke, and N. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in *Proc. Int. Conf. Embed. Comput. Syst., Archit., Model. Simul.*, 2010, pp. 1–10.
[4] V. Herdt, D. Große, and R. Drechsler, "Fast and accurate performance evaluation for RISC-V using virtual prototypes," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2020, pp. 618–621.
[5] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "GVSoC: A highly configurable, fast and accurate full-platform simulator for RISC-V based IoT processors," in *Proc. IEEE 39th Int. Conf. Comput. Design (ICCD)*, 2021, pp. 409–416.
[6] R. Kassem, M. Briday, J.-L. Béchennec, Y. Trinquet, and G. Savaton, "Simulator generation using an automaton based pipeline model for timing analysis," in *Proc. Int. Multiconf. Comput. Sci. Inf. Technol.*, 2008, pp. 6570–664.
[7] OpenHW Group. "CV32E40P user manual." 2023. Accessed: Mar. 20, 2024. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/
[8] OpenHW Group. "CVA6 user manual." 2023. Accessed: Mar. 20, 2024. [Online]. Available: https://docs.openhwgroup.org/projects/cva6-user-manual/
[9] S. Vinco, V. Guarnieri, and F. Fummi, "Code manipulation for virtual platform integration," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2694–2708, Sep. 2016.
[10] W. Snyder. "Verilator." 2003. Accessed: Jun. 14, 2024. [Online]. Available: https://verilator.org
[11] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. 30th Annu. Int. Symp. Comput. Architect.*, 2003, pp. 84–95.
[12] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using time-based sampling," in *Proc. 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2013, pp. 448–459.
[13] D. C. Powell and B. Franke, "Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators," in *Proc. 7th IEEE/ACM Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, 2009, pp. 315–324.
[14] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.
[15] D. Thach, Y. Tamiya, S. Kuwamura, and A. Ike, "Fast cycle estimation methodology for instruction-level emulator," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, 2012, pp. 248–251.
[16] R. Kassem, M. Briday, J.-L. Béchennec, G. Savaton, and Y. Trinquet, "Harmless, a hardware architecture description language dedicated to real-time embedded system simulation," *J. Syst. Archit.*, vol. 58, no. 8, pp. 318–337, Sep. 2012.
[17] A. Bullich, M. Briday, J.-L. Béchennec, and Y. Trinquet, "A compiled cycle accurate simulation for hardware architecture," in *Proc. 5th Int. Conf. Adv. Syst. Simul. (SIMUL)*, 2013, pp. 213–225.
[18] C. Foik, D. Mueller-Gritschneder, and U. Schlichtmann, "CorePerfDSL: A flexible processor description language for software performance simulation," in *Proc. Forum Specif. Design Lang. (FDL)*, 2022, pp. 1–8.
[19] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, "The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping," in *Proc. Int. Symp. Rapid Syst. Prototyp. (RSP)*, 2017, pp. 79–84.
[20] (Free and Open Source Silicon Found., Halifax, U.K.). *Embench: A Modern Embedded Benchmark Suite*. (2021). Accessed: Mar. 20, 2024. [Online]. Available: https://www.embench.org/
[21] (SiFive, Santa Clara, CA, USA). *SiFive FE310-G002 Manual*. (2022). Accessed: Mar. 22, 2024. [Online]. Available: https://www.sifive.com/documentation