CorePerfDSL: A Flexible Processor Description Language for Software Performance Simulation

Conrad Foik, Daniel Mueller-Gritschneder, Ulf Schlichtmann
Chair of Electronic Design Automation, *Technical University of Munich*
Munich, Germany
{conrad.foik, daniel.mueller, ulf.schlichtmann}@tum.de

***Abstract*—Instruction set simulators (ISSs) model the functional behavior of embedded processors for early software development. While they offer high simulation speeds, an ISS usually does not model the timing behavior of the processor accurately. Existing software performance simulators are typically either specific to a certain microarchitecture or their description language mixes functional and microarchitectural aspects.**

In this paper, we introduce CorePerfDSL, an architecture description language (ADL) specifically designed to model the timing behavior of processor microarchitectures for software performance estimation. CorePerfDSL is clearly separated from any functional description of the modelled processor by a generic trace definition. As such, it is well suited to generate performance simulators that can be paired with an existing ISS, which supplies an execution trace. In addition, CorePerfDSL provides a high degree of flexibility, supporting the fast generation of models for various microarchitecture variants, which can be used for rapid architectural exploration. We demonstrate the flexibility of CorePerfDSL by describing several variants of a single-issue five-stage RISC-V microarchitecture and estimate their performances for a software benchmark program.

***Index Terms*—ISS, VP, ADL, Microarchitecture, Pipeline**

I. Introduction

To cope with the increasing complexity of today's embedded systems, and in order to meet strict time-to-market demands, modern design methodologies rely heavily on abstract computer models, so-called virtual prototypes (VPs), of the target hardware for early software development. Besides the possibility to verify the developed software early on, VPs also offer the opportunity to conduct an architectural exploration of the design space prior to the development of the system.

Typical tools used during abstract system modelling are so-called instructions set simulators (ISSs), which simulate the functional behavior of a processor at instruction set architecture (ISA) level [1] [2]. They can be used as processor models within a VP or as stand-alone simulators. While ISSs typically offer high simulation speeds, they usually do not model microarchitectural aspects of the target processor. As such, they are not able to provide reliable estimations regarding the software performance on the processor. To improve timing estimations, different performance simulators have been proposed. Some of these simulators use detailed functional models of the microarchitecture [3], at the cost of reduced simulation speed. Other approaches incorporate abstract, non-functional timing models [4] or statistical methods [5], in an attempt to maintain high simulation speeds.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0131.

978-1-6654-7332-3/22/$31.00 ©2022 IEEE

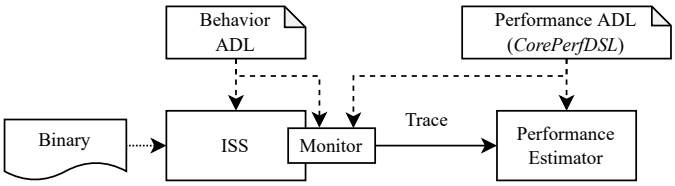

Fig. 1. Performance simulation setup with ISS and independent estimator.

As a support of rapid architectural exploration, so-called architecture description languages (ADLs) exist. Behavioral ADLs, which describe the functional behavior of a processor on an abstract level, are frequently used to automatically generate ISA-specific parts of an ISS [6] [7]. For performance simulators, similar approaches exist, using ADLs that also include a description of the processors microarchitecture [8] [9]. Since these languages attempt to describe both aspects, though, the description of the timing behavior becomes dependent on the functional description.

However, for a more flexible approach, an ADL that solely focuses on the processor's timing behavior is beneficial. Such an isolated description can be used to specify abstract, non-functional timing models of the processor, without unnecessary specification of functional aspects. It is, as such, well suited to explore the impact of different microarchitectures on software performance. In addition, it allows to generate ISS-independent performance estimators, which can, with low effort, be combined with existing simulation setups. Fig. 1 illustrates such a concept. A behavioral ADL is used to implement an ISS, while the estimator is setup independently through an ADL focusing on timing behavior. To provide a connection between ISS and estimator, an ISS-monitor is required, to generate a trace for the estimator. An efficient description of the estimator generically defines the required content of the trace. Combining such a generic trace definition with the behavioral ADL, enables the flexible adaptation of the ISS-monitor.

In this paper, we propose CorePerfDSL, an ADL, which is suited for the approach presented in Fig. 1. For this purpose,

CorePerfDSL contributes the following features:

- An isolated, non-functional description of the processor's timing behavior, capable of modelling essential microarchitectual aspects, including structural, data and control hazards.
- High flexibility of the description, suited for rapid architectural exploration, by dividing the instructions' workloads into small microactions which can be assigned to pipeline stages.
- A generic trace definition, for linking the performance simulator up with any ISS via an automatically generated ISS monitor.
- Mechanisms to incorporate external dynamic timing models to describe complex timing behaviors and to enable reuse of existing models.

We demonstrate the flexibility of CorePerfDSL, as well as its capability to generate efficient performance simulators, by creating proof-of-concept estimators based on CorePerfDSL. The estimators deliver well-founded performance estimates for 12 variants of a single-issue, five-stage RISC-V microarchitecture, and achieve simulation speeds of up to 15 million instructions per second (MIPS).

The rest of the paper is structured as followed. Sec. II contains a brief overview of related work. Then, CorePerfDSL is presented in detail in Sec. III. The results of the proof-of-concept simulations are presented in Sec. IV. Finally, Sec. V concludes this paper.

II. Related work

Today a number of ISSs exist, which are used in academical and industrial contexts. Examples are QEMU [1], Spike [10], ETISS [2] or DBT-RISE [11]. As discussed, these simulators focus on modeling functional behavior and are designed for high simulation speeds. They are generally not able to provide reliable estimations regarding the timing behavior of the processor by themselves, but can be incorporated into performance simulation setups (ref. Fig. 1).

The SMARTS [5] and ESECS [12] performance simulators combine ISSs with cycle-accurate models. Statistical sampling on the cycle-accurate model is applied in order to increase simulation speed. The work in [13] also incorporates cycle-accurate and abstract models, but uses a machine learning method. These approaches, however, still rely on the existence of a cycle-accurate implementation or performance model.

The gem5 [3] simulator integrates detailed functional models of the microarchitecture of the target processor. As this level of detail comes at the cost of reduced simulation speed, this approach is not well suited for rapid software development. In contrast, the performance simulators in [4] [14] and [15] extend functional ISS-environments with detailed timing models, while abstractly modeling the functional behavior on ISA-level. This approach is similar to the one described in Fig. 1. However, to the best of our knowledge, none of these simulators provides an independent description of the timing models, which would enable the combination with other existing ISSs.

Several ADLs have been proposed in the literature. They are commonly classified as structural, behavioral or mixed ADLs.

MIMOLA [16] is a *structural* ADL which describes the processor's structure as a netlist of connected modules. It targets hardware synthesis and not primarily the generation of performance simulators. Also, as its description is comparatively low-level, it is laborious to modify and not well suited for architectural exploration.

Examples of *behavioral* ADLs are nML [6], ISDL [7] or SAIL [17]. These languages are designed to describe a processor's behavior in terms of its instruction set. They are frequently used to generate ISSs. While ISDL allows to specify the latency of operations, behavioral ADLs are generally not able to capture enough information to model instruction dependencies or concurrencies as they occur in pipelines.

Mixed ADLs, as for example LISA [8], EXPRESSION [9], MADL [18] or ArchC [19], combine structural and behavioral aspects. These languages are in general capable of generating performance simulators of the described processors. However, these approaches do not clearly separate the description of functional and timing behavior. HARMLESS [20] isolates the description of the processor's microarchitecture, to allow for a purely functional description of the processor. However, it is not possible to describe the timing behavior independently of the functionality.

None of the presented ADLs are as such able to provide an isolated, non-functional description of the processor's timing behavior, suited for rapid architectural exploration.

III. The CorePerfDSL language

In this section, we first give an overview over the CorePerfDSL language and then discuss the concepts in detail.

A. Overview

Fig. 2 presents an overview of the components of the CorePerfDSL language. Conceptually, these components can be grouped into four sections: Microaction, external model, instruction and core section.

The *microactions* are used to break down each instruction of the processor into small, manageable sub-behaviors, for example the fetch of the instruction word. For each such microaction the required resources can be defined, which model the required hardware timing, e.g., the delay of the instruction read interface. Additionally, so-called connectors can be defined to describe data dependencies, e.g., for an instruction fetch the program counter (PC) value must be known while an PC increment microaction supplies the next PC value. Resources and microactions can also be declared as *virtual* components, which act as placeholders for quickly swapping definitions of resources or microactions without having to redefine the complete pipeline model, which is discussed in Sec. III-E.

The capability to include *external models* within the CorePerfDSL provides the ability to incorporate dynamic timing models written in an external language (e.g. C++). This enables the modelling of more complex, system specific timing

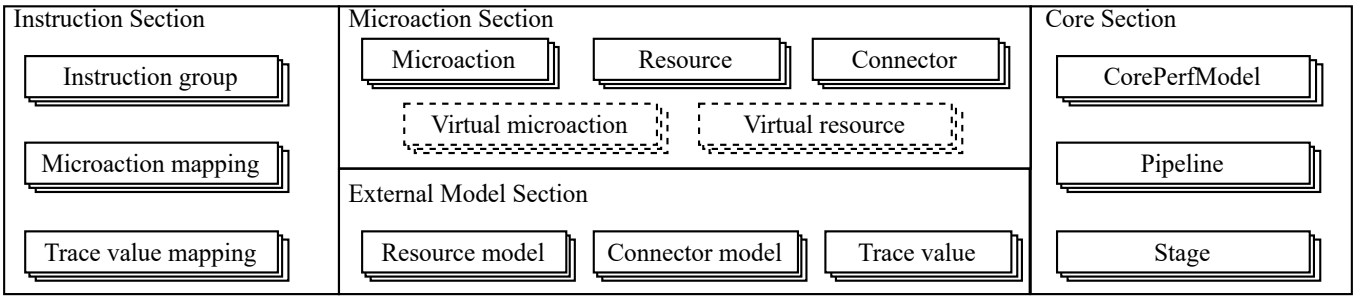

Fig. 2. Overview of the CorePerfDSL language.

behaviors which cannot easily be generalized. For example, for an instruction fetch, the timing could be determined by calling an external instruction cache model. This calling mechanism allows the reuse of already existing models, for instance from an existing functional VP simulation. External models can be specified either as resource or connector model components. Trace values of the ISS, which are required by the external timing models, e.g. the value of the PC for the instruction cache, are defined through corresponding components.

The *instruction section* defines firstly how each instruction is broken down into the microactions. Secondly, the trace values of the external timing models are mapped to the simulation variables of the ISS. This mapping enables the definition of a generic trace, which can be used for the automatic generation of an ISS-monitor (ref. Fig 1). To avoid unnecessary coding overhead, similar instructions can be bundled through instruction group components.

The purpose of the *core sections* is to describe the actual microarchitecture in form of one or several so-called core performance models (CorePerfModel). To this mean, the specified microactions are assigned to stages, which in turn are organized in pipelines. The CorePerfModel selects one of these pipeline models as well as the appropriate connector models from the external model section. In addition, the CorePerfModel resolves any virtual microactions or resources by replacing them with a non-virtual counterpart, allowing to rapidly explore different variants by quickly swapping microaction and resource definitions without having to redefine the complete pipeline.

B. Microaction Section

1) Microaction definition: Microactions are a central component of our CorePerfDSL. They break down the instructions into the sub-steps that can be executed within one stage of the processor pipeline. The mapping of these microactions on pipeline stages later allows to compute the timing behavior of the instructions as it is caused by a pipelined execution.

Since they represent actions, which are performed by the processor, microactions are typically associated with a hardware resource, as for example the arithmetic logic unit (ALU) or memory ports. In CorePerfDSL, this relation is expressed by assigning a resource component to one or several microactions. Assigning a specific resource to multiple microactions is explicitly allowed, in order to describe resource sharing, which may result in structural hazards.

In addition, some microactions can only start to execute when certain data is available, e.g. the PC. These kind of dependencies can cause data and control hazards that add additional delays due to pipeline stalls. To express these dependencies, CorePerfDSL provides connector components which can be assigned to microactions, either as inputs or outputs.

It is important to recall that CorePerfDSL focuses on the description of the timing behavior of a microarchitecture. As such, resource components merely model the delay of the corresponding hardware. Similar, connector components do not pass the actual data values (e.g. the value of the next PC), but simply indicate the availability of the associated data. The functional behavior of the microactions is, thus, explicitly not described.

Based on the assigned resources and connectors, the timing behavior of the microactions can be described. When an microaction gets activated, it waits until the data value, modelled by its input connector, is available. If the microaction does not have an input connector assigned to it, this waiting period is dropped. Once the input connector is available, the microaction starts executing. This is modelled through the resource, which adds a resource specific delay. After this delay has passed, the microaction is considered to be completed. If the microaction has an output connector, the corresponding data value is now set as available.

It is important to note that the output connector of a microaction is not directly connected to an input connector. Instead, connectors are related via so-called connector models (see Sec. III-C). This allows the description of dynamic dependencies. A typical example is the modeling of branch prediction schemes. In this context, it makes sense to define two output connectors: One to model the availability of the predicted PC, and one to describe the availability of the actually computed PC, in case of a misprediction. The branch prediction scheme itself is represented by a connector model. Depending on whether a misprediction is detected, the connector model maps the appropriate output connector to the input connectors of PC-dependent microactions.

Listing 1 illustrates the definition of microactions in CorePerfDSL. The microaction `uA_ID` models the instruction decoding. It has no data dependencies that can lead to pipeline hazards and, therefore, only gets assigned a resource, `Decoder`. In contrast, the microaction that describes the instruction fetch, `uA_IF`, requires the PC, which can be subject

```
Connector  {PC, PC_p, PC_np, R_a, R_b, R_d}
Resource   {RAM_R, PC_Gen, Decoder, ALU, ...}
Microaction{
    uA_PC_Gen        (PC -> PC_Gen -> PC_p),
    uA_IF            (PC -> RAM_R),
    uA_ID            (Decoder),
    uA_OF_A          (R_a),
    uA_ALU_branch    (ALU -> PC_np),
    uA_ALU_arith_fw  (ALU -> R_d)
    ...
}
```

Listing 1. Definition of connector, resource and microaction components

to control hazards for branch instructions. This dependency is described by the PC connector, which is placed as an input in front of the RAM_R resource. The generation of the PC value is modelled by uA_PC_Gen. After receiving the current PC value, PC, this microaction generates the PC for the next instruction, which is described by the PC_p output connector, placed behind the PC_Gen resource. However, as discussed above, for a branch instruction, PC_p merely represents a predicted PC. In case of a misprediction, a control hazard occurs and the actual (not predicted) PC, PC_np is first available once the branch is evaluated. uA_ALU_branch describes the evaluation of the branch, using the ALU resource.

Listing 1 also illustrates two microactions related to arithmetic calculations. uA_OF_A models the fetch of the operand A form the processor's registers. If the operand is the result of a previous instruction, a data hazard can occur. This data dependency is described by assigning the connector R_a, which is interpreted as an input, to uA_OF_A. The arithmetic operation itself is modelled by uA_ALU_arith_fw. This microaction uses the ALU resource to update the destination register of the instruction, R_d. Note that uA_ALU_arith_fw describes an arithmetic operation for a pipeline with data forwarding, as the data associated with R_d is available to the next instruction right after it has been calculated by the ALU.

2) Resource behavior and virtualization: The resources of Listing 1 have been defined without their timing behavior. As a default, a timing of one clock cycle is assigned. However, CorePerfDSL also allows the specification of larger or dynamic delays. This is illustrated in Listing 2. In this example, the resource Multiplier has been defined with a static delay of three clock cycles. Similarly, it is also possible to assign a dynamic delay, by providing the name of a resource model component (see Sec. III-C), which will be used to calculate the delay. For instance, in Listing 2 the read port of the data memory is provided as a dynamic resource, DRAM_R_dyn, using the DRAM_model resource model.

```
Resource {Multiplier(3)}
Resource {DRAM_R_sta, DRAM_R_dyn(DRAM_model)}
virtual Resource vDRAM_R
Microaction{
    uA_MEM_R     (vDRAM_R),
    uA_MEM_R_fw  (vDRAM_R -> R_d)
}
virtual Microaction{vuA_MEM_R, vuA_ALU_arith}
```

Listing 2. Definition of dynamic resources and virtual components

In order to flexibly describe different microarchitectures, it is often useful to postpone the decision on how to model a specific resource until the definition of the CorePerfModel (Sec. III-E). CorePerfDSL allows to do this through so-called virtual resources. Virtual resources are essentially placeholders, which can be assigned to microactions. During the definition of the CorePerfModel, they are then replaced with non-virtual resources. In Listing 2, the virtual resource vDRAM_R is used as a placeholder for the read port of the data memory. This can later be replaced, for instance to use a dynamic model (DRAM_R_dyn) or to use the same memory port as the instruction fetch (RAM_R), in order to model resource sharing.

Similarly, CorePerfDSL also allows the definition of virtual microactions, which can be used as placeholders during the definition of microaction mappings (Sec. III-D) and stages (Sec. III-E). For example, Listing 2 defines the virtual microaction vuA_MEM_R. It can later be replaced by a microaction modelling a data load operation with (uA_MEM_R_fw) or without data forwarding (uA_MEM_R).

C. External Model Section

The external model section provides the ability to incorporate models which are implemented in a different language (e.g. C++) into the CorePerfDSL description. This enables the modelling of complex, system specific behaviors, as for example branch prediction schemes, and also allows the reuse of existing timing models. External models can be specified either for resources or connectors.

As discussed in combination with Listing 2, *resource models* are used to assign a dynamic behavior to the resources of the microaction section. Listing 3 shows the definition of a resource model for the data memory, DRAM_model. Besides its name, a resource model get assigned two attributes. First, using the link keyword, a descriptor of the file containing the external model is specified, in this case a C++ file. Second, under trace, any number of so-called trace values can be specified.

The *trace values* define which parameters need to be provided by the ISS for computing the timing. A memory model like the DRAM_model, for instance, is likely to require the address of a memory access in order to calculate the corresponding delay. It is worth noticing that the name of the defined trace values does not need to correspond to actual state names of the ISS. This relation will be established by the instruction section (Sec. III-D).

```
TraceValue {mem_addr}
ResourceModel DRAM_model(
    link : SimpleDRam.cpp
    trace: mem_addr
)
```

Listing 3. Definition of a resource model component

As mentioned in Sec. III-B, the intention of *connector models* is to model data and control hazards. A connector model determines when an input of one microaction becomes available based on the availability of one more more outputs of other microactions, in order to model the flow of data. Listing 4 shows two example connector models.

The Register_model connector model describes data hazards. To do so, it has the result register R_d connector assigned to it as an input, and the operand registers R_a and

```
TraceValue {pc_val, ra_addr, rb_addr, rd_addr}
ConnectorModel Register_model(
    link        : StandardRegister.cpp
    trace       : {ra_addr, rb_addr, rd_addr}
    connectorIn : R_d
    connectorOut: {R_a, R_b}
)
ConnectorModel DynBranchPredict_model(
    link        : DynamicBranchPredict.cpp
    trace       : pc_val
    connectorIn : {PC_p, PC_np}
    connectorOut: PC
)
```

Listing 4. Definition of connector model components

R_b as outputs. The Register_model can thus set the availability of R_a and R_b, based on R_d and the addresses of the used registers (ra_addr, rb_addr, rd_addr).

DynBranchPredict_model is used to model the branch prediction schemes and control hazards within the processor pipeline. With regards to Listing 1, the next PC, PC_p, is generated by the uA_PC_Gen microaction, unless a misprediction occurs during the execution of a branch instruction. In that case, uA_ALU_branch provides the next PC, PC_np. DynBranchPredict_model takes both PC_p and PC_np as inputs, and, depending on whether a misprediction occurred, determines when the PC connector at its output is available.

D. Instruction Section

In the instruction section the breakdown of the instructions into microactions is defined. Furthermore, for each instruction, a mapping of the trace values to observable variables in the ISS need to be specified, to enable the generation of an ISS-monitor that can provide the required trace for timing computation.

```
InstrGroup{
   Arith (ADD, SUB, [?]),
   Branch (BNE, BEQ, ...),
   ...
}
MicroactionMapping{
   [ALL] : {uA_PC_Gen, uA_IF, uA_ID},
   Arith : {uA_OF_A, uA_OF_B vuA_ALU_arith, vuA_WB},
   Branch: {uA_OF_A, uA_OF_B, uA_ALU_branch},
   LW    : {uA_OF_A, vuA_MEM_R, ...},
   ...
}
```

Listing 5. Definition of microaction mapping components

Listing 5 presents an example of the specification of the required microactions for an instruction. As similar instructions typically use the processor pipeline in a similar manner, they often have the same microactions assigned to them. To avoid an unnecessary coding overhead, CorePerfDSL allows to bundle similar instructions into instruction groups. For example, the Arith instruction group contains the instructions ADD and SUB. In addition, the [?] keyword defines that any instruction, which is not explicitly mapped or part of another instruction group is also a member of Arith. Based on this, the microactions can be assigned either to individual instructions (e.g. LW) or instruction groups. For instance, for the execution of the instructions of the Arith group two operands are fetched (uA_OF_A and uA_OF_B), the arithmetic operation is carried out (vuA_ALU_arith) and the result is written back to the processor's registers (vuA_WB). Microactions which are required by every instruction of the ISA can be assigned using the [ALL] keyword. Finally, it is worth noticing that virtual microactions have been used for the mapping of Arith and LW, to increase flexibility.

The mapping of the trace value follows a similar approach as that of the microactions. For every defined trace value a descriptor has to be specified in form of a string. In principle, this description could differ from instruction to instruction. However, in many ISS implementations, variables are consistently named, and as such the same descriptors can be used for all instructions of the ISA. Listing 6 shows an example of the definition of a trace value mapping component, assuming a consistent naming of the ISS variables.

```
TraceValueMapping{
    [ALL] : { pc_val : "PC",
              ra_addr : "REG_S1",
              ...
            }
}
```

Listing 6. Definition of a trace value mapping component

E. Core Section

1) CorePerfModel definition: In the core section, one or several CorePerfModels are defined. Each of these CorePerfModels describe a variant of a microarchitecture.

The central component of a microarchitecture is its pipeline. Listing 7 shows the definition of a pipeline component. As a first step, the microactions are assigned to appropriate stages. Microactions that are grouped into the same stage are considered to be executed in parallel. For instance, it is common that the instruction fetch and the generation of the next PC are executed simultaneously. Listing 7 therefore specifies IF_stage to contain uA_IF and uA_PC_Gen. The other stages are defined in a similar manner. It is worth noticing, that the virtual microaction vuA_ALU_arith is used during the definition of EX_stage. The stages are then arranged into a pipeline, as illustrated by the SimplePipeline example. At this point, CorePerfDSL only supports the definition of in-order single-issue pipelines. This means that each stage can only handle one instruction at the time, and a stage can only be completed, if the next stage is available.

```
Stage{
   IF_stage  (uA_IF, uA_PC_Gen),
   ID_stage  (uA_ID, uA_OF_A, uA_OF_B),
   EX_stage  (vuA_ALU_arith, uA_ALU_branch, ...),
   MEM_stage (...),
   ...
}
Pipeline SimplePipeline (IF_stage -> ID_stage -> EX_stage
                                  -> ...)
```

Listing 7. Definition of stage and pipeline components

Listing 7 shows the definition of a single pipeline. However, it should be pointed out that other stage and pipeline variants could easily be added. For example, further stage variants could be defined to merge EX_stage and MEM_stage, or to split IF_stage into two stages. Additional pipeline variants could then be specified that pick the appropriate stages to describe a four-stage and six-stage pipeline, respectively. The

only limitation in this context is that a pipeline cannot use multiple stages which contain the same microaction.

Based on the pipeline components, one or several CorePerfModels can be specified. Listing 8 presents an example of the definition of a CorePerfModel, SimpleRISCV. With the keyword use, the CorePerfModel specifies the used pipeline as well as which external connector models are to be employed (ref. Listing 4). Note that external resource models are not selected in this manner, as they are implicitly specified through the resources of the used microactions. If the specified pipeline contains virtual components, either through the used stages or microactions, these components are replaced by non-virtual counterparts through the assign keyword. SimpleRISCV, for instance, replaces vDRAM_R to use the same static model for the instruction and data memory read port (ref. Listing 1), and selects the appropriate microactions to describe a microarchitecture with data forwarding (ref. Listing 2).

```
CorePerfModel SimpleRISCV(
  use Pipeline : SimplePipeline
  use ConnectorModel : {
    Register_model,
    DynBranchPredict_model
  }
  assign Resource : {
    vDRAM_R = RAM_R,
    ...
  }
  assign Microaction : {
    vuA_ALU_arith = uA_ALU_arith_fw,
    ...
  }
)
```

Listing 8. Definition of a CorePerfModel component

2) Conceptual view: The definition of the CorePerfModel concludes the CorePerfDSL description. Based on the described components, it is now possible to generate a conceptual view of the modelled microarchitecture. This is depicted in Fig. 3, for the CorePerfModel specified in Listing 8.

The described microarchitecture, SimpleRISCV, contains a pipeline and two connector models. The pipeline, SimplePipeline, is divided into five stages, each containing one or several microactions. The microactions contain resources (rectangular boxes) and connectors (boxes with rounded edges). The input and output connectors of the microactions are mapped to the outputs and inputs of the appropriate connector models, respectively.

Combining this view of the mircoarchitecture with the microaction mapping, as shown in Listing 5, allows to describe the usage of the microarchitecture by a given instruction in terms of required microactions. For instance, in Fig. 3 all microactions which are used by an ADD instruction are depicted gray tinted.

Based on this view, the timing behavior of a given instruction can be described. For the given example, the ADD instruction enters the IF_stage as soon as the stage is available, and the two microactions uA_IF and uA_PC_Gen are activated. As discussed in Sec. III-B, the microactions will be executed once the PC connector is available. The execution is modelled by adding the delay of the corresponding resource. After the execution of uA_PC_Gen, PC_p is set and passed to DynBranchPredict_model, in order to determine PC for the next instruction. The ADD instruction moves to the next stage, ID_stage, once both microactions in IF_stage have been executed and ID_stage is available. The behavior of the other stages follows the same concept.

It is worth noticing that this view allows to estimate performance in an instruction-by-instruction manner, rather than keeping track of the pipeline's state cycle-by-cycle.

IV. Experimental results

In order to show that the language is capable of modelling the timing behavior of typical microarchitectures, as well as to demonstrate the flexibility of the approach, we use CorePerfDSL to estimate the performance of a variety of microarchitecture variants. As an early proof-of-concept, the used estimators are implemented manually, using the CorePerfDSL descriptions as guidance. However, since the estimators strictly follow the conceptual view of the CorePerfDSL, as presented in Fig. 3, code generators can be used in the future to automate this process.

The following subsection describes the used microarchitecture variants. In Sec. IV-B, the simulation setup and the obtained performance estimates are presented.

A. Architecture Variants

An artificial, but characteristic, microarchitecure, referred to as SimpleRISCV, is defined as a basis for our experiments. SimpleRISCV is a single-issue five-stage implementation of the RISC-V ISA. To demonstrate the flexibility of CorePerfDSL, several variants of SimpleRISCV are described and modelled, combining the subsequently described features. Note that the example presented in Fig. 3 describes a variant of SimpleRISCV.

SimpleRISCV can be described with and without data forwarding. In case of no data forwarding, data provided by an instruction will not be available to the following instructions before it is written to its destination register during the write-back stage (WB_stage). If data forwarding is applied, the result of an instruction will be available for the next instruction right after it has been computed in the execution stage (EX_stage) or after it has been read from memory (MEM_stage). Fig. 3 depicts a variant of SimpleRISCV using data forwarding.

Three branch prediction schemes can be applied to SimpleRISCV: Non, static or dynamic branch prediction. If no branch prediction is used, a branch instruction does not generate a predicted PC (PC_p) in IF_stage. Thus, for these type of instructions, the next PC will be generated by EX_stage, causing the next instruction to always stall for two cycles. In case of static variant, a fixed not-taken prediction is made, i.e. it is always predicted that the branch is not taken. For the dynamic branch prediction, a common approach using a 2-bit branch history buffer is applied [15], which incorporates earlier evaluations of the branch into its prediction. The example in Fig. 3 shows a variant with dynamic branch prediction, modelled by DynBranchPredict_model.

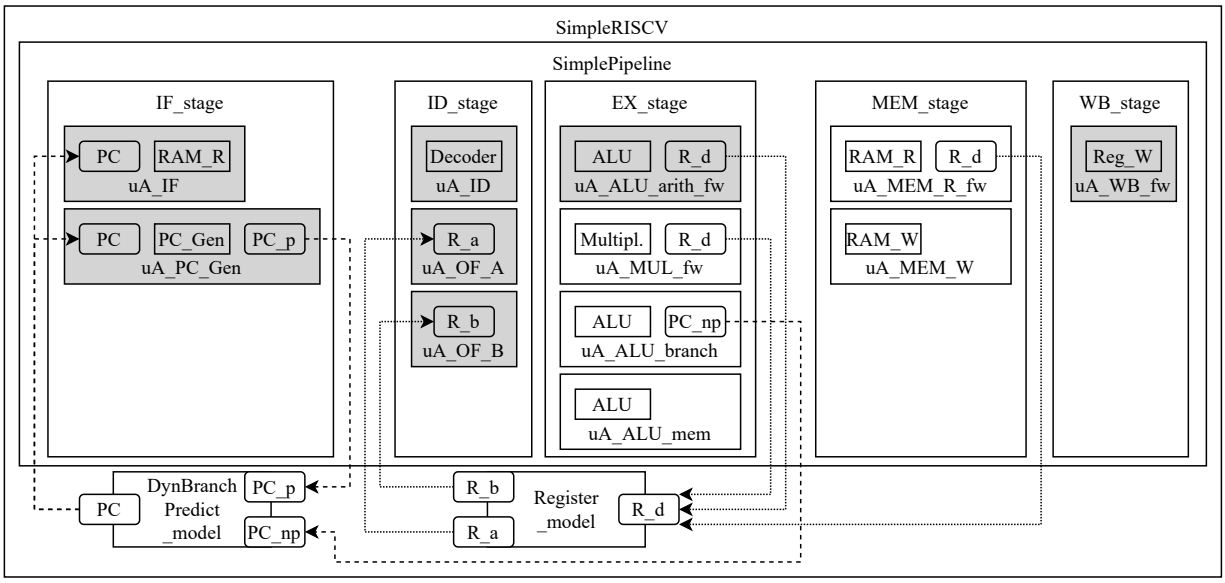

Fig. 3. Conceptual view of the modelled microarchitecture. (Gray tinted mircoactions are the ones used by an ADD instruction)

As another feature, `SimpleRISCV` can either be described with a Harvard or a von Neumann architecture. In case of the von Neumann architecture, `IF_stage` and `MEM_stage` share the same memory read port. As such, no instruction fetch can be carried out while a load instruction is in `MEM_stage`, causing a structural hazard. Using a Harvard architecture, both stages use unique read ports. Assigning the same resource to multiple microactions, as depicted in Fig. 3, can be used to describe resource sharing and, in this case, to model a von Neumann architecture.

Finally, the `SimpleRISCV` can be used with different resource models. As a default, a delay of one clock cycle is assumed both for the processors multiplier as well as for the memory ports. Alternatively, a static three cycle delay for the multiplier can modelled. For the memory ports, a dynamic model can be used, which returns a delay of five cycles for a cache miss with a cache miss rate of 1/100 and otherwise a delay of one cycle. The `SimpleRISCV` variant presented in Fig. 3 uses the default models.

B. Performance Simulation

To conduct the performance simulation for the various `SimpleRISCV` variants, we pair our estimators with an ISS called ETISS [2]. This simulator offers a number of plugin interfaces, which make it possible to observe the executed instructions with rather low effort, and it is therefore well suited to generate the instruction trace for the performance estimation. As a test program, the Dhrystone benchmark [21], using one million loops, is executed by ETISS. Executing the simulation setup without the performance estimation activated, ETISS achieves a simulation speed of approximately 41.12 million instructions per second (MIPS).

1) Simulator evaluation: As a first evaluation of the developed approach, an oversimplified model of the `SimpleRISCV` is simulated. This model does not consider structural, data or control hazards, and assumes that every resource has a fixed delay of one clock cycle. This means that every instruction only spends one cycle in each state. The results of this initial simulation are presented in Tab. I. The estimated cycle-per-instruction (CPI) is at 1, as an estimate of the performance of the target processor. The number of cycles exceeds the instruction count by exactly four cycles as expected for a five-stage pipeline to account for the ramp-up phase. For this scenario, we reached a simulation speed of approximately 15 MIPS, which characterizes the penalty due to the ISS-monitor and the performance estimation.

TABLE I
RESULTS FOR A SIMPLIFIED VERSION OF SIMPLERISCV

Number of instructions	710,086,326
Estimated number of cycles	710,086,330
Est. performance of target processor (CPI)	1
Simulation performance (MIPS)	15.1433

2) Modelling of essential timing behavior: Next, we model 12 variants of the `SimpleRISCV` combining the data forwarding and branch prediction features, described in Sec. IV-A, with a Harvard and von Neumann architecture, respectively. This illustrates the ability of CorePerfDSL to model structural, data and control hazards. Note that all resources are modelled with a delay of one cycle in this scenario. The results of the simulations with these 12 variants are shown in Tab. II, presenting the estimated performance of the modelled microarchitecture in CPI, and the achieved simulation speed in MIPS.

The obtained estimates fit typical expectations for these kind of microarchitectures. For instance, for the von Neumann variant with data forwarding and dynamic branch prediction, a performance of 1,27 CPI is estimated. Given the fact that load instructions make up roughly 27% of the executed benchmark, and taking into consideration that structural hazards are the dominating cause for pipeline stalls for this `SimpleRISCV` variant, this estimate is well-founded.

We observe a drop of simulation performance when using the von Neumann architecture, to roughly 6 MIPS. This is

TABLE II
RESULTS FOR SIMPLERISCV MICROARCHITECTURE VARIANTS

Architecture	Forwarding	Branch prediction	Est. target perf. (CPI)	Simulation speed (MIPS)
Harvard	No	No	1.59	15.07
		Static	1.52	13.97
		Dynamic	1.48	12.27
	Yes	No	1.23	15.09
		Static	1.15	13.92
		Dynamic	1.11	12.72
von Neumann	No	No	1.68	6.67
		Static	1.60	6.44
		Dynamic	1.57	6.13
	Yes	No	1.36	6.75
		Static	1.30	6.42
		Dynamic	1.27	6.08

due to the fact that the simulator needs to run a scheduling algorithm for the shared read port every time an instruction fetch or a data load access is carried out, i.e. at least once per instruction. However, given the proof-of-concept nature of our current simulator, room for optimization of this algorithm exists, and it is likely that the simulation performance can be further improved.

3) Simulation of static and dynamic resources: Finally, we simulate a `SimpleRISCV` variant with different resource models. As the base variant, we use the `SimpleRISCV` with Harvard architecture, which applies both data forwarding and dynamic branch prediction. For the resource models, a single-cycle or a three-cycle multiplier can be used, as well as a static or a dynamic memory model, as discussed in Sec. IV-A. The results of the simulations are summarized in Tab. III. Also in this case the estimates seem well-founded. For example, multiplications make up roughly 1% of the benchmark. This fits the increase of 0.02 CPI when switching the multiplier model to use two extra cycles, while applying the static memory model (with the dynamic memory model, hazards may overlap, explaining an increase below 0.02 CPI).

TABLE III
RESULTS FOR SIMPLERISCV WITH VARYING RESOURCE MODELS

Memory	Multiplier	Est. target perf. (CPI)	Simulation speed (MIPS)
Static	1-cycle	1.11	12.74
	3-cycle	1.13	12.54
Dynamic	1-cycle	1.17	12.37
	3-cycle	1.18	11.60

V. CONCLUSION

In this paper, we have presented CorePerfDSL, a microarchitecure description language specifically designed to model the timing behavior of processors during software performance evaluation. CorePerfDSL provides a high degree of flexibility in the description of the microarchitecture and is, therefore, well suited for rapid architectural exploration.

The flexibility of CorePerfDSL has been demonstrated through the modelling of multiple variations of a single-issue five-stage pipeline implementation of the RISC-V ISA. The results show that CorePerfDSL is capable of modelling the essential timing behavior of this kind of processors, including structural, data and control hazards. The simulation performance of our approach reached up to 15 MIPS.

REFERENCES

[1] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference*, Apr. 2005, p. 41.

[2] 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 *International Symposium on Rapid System Prototyping (RSP)*, Oct. 2017, pp. 79–84.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, May 2011.

[4] 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 *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2010, pp. 1–10.

[5] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *30th Annual International Symposium on Computer Architecture*, June 2003, pp. 84–95.

[6] A. Fauth, J. Van Praet, and M. Freericks, "Describing Instruction Set Processors Using nML," in *Proceedings the European Design and Test Conference*, Mar. 1995, pp. 503–507.

[7] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability and Architecture Exploration," *Des. Autom. Embed. Syst.*, vol. 6, no. 1, pp. 39–69, Sept. 2000.

[8] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures," in *Design Automation Conference*, June 1999, pp. 933–938.

[9] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability," in *Design, Automation and Test in Europe Conference and Exhibition*, Mar. 1999, pp. 485–490.

[10] "Spike RISC-V ISA Simulator," https://github.com/riscv-software-src/riscv-isa-sim, accessed: 13. May 2022.

[11] MINRES Technologies, "DBT-RISE," https://github.com/Minres/DBT-RISE-Core, accessed: 13. May 2022.

[12] E. K. Ardestani and J. Renau, "ESESC: A fast Multicore Simulator Using Time-Based Sampling," in *19th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 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 *7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, Oct. 2009, p. 315–324.

[14] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 593–606, Mar. 2011.

[15] V. Herdt, D. Große, and R. Drechsler, "Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes," in *Design, Automation Test in Europe Conference and Exhibition (DATE)*, Mar. 2020, pp. 618–621.

[16] G. Zimmermann, "The MIMOLA Design System: A Computer Aided Digital Processor Design Method," in *16th Design Automation Conference*, June 1979, pp. 53–58.

[17] K. E. Gray, P. Sewell, C. Pulte, S. Flur, and R. Norton-Wright, "The Sail instruction-set semantics specification language," https://www.cl.cam.ac.uk/~pes20/sail/manual.pdf, Mar. 2017, accessed: 19 May 2022.

[18] W. Qin, S. Rajagopalan, and S. Malik, "A Formal Concurrency Model Based Architecture Description Language for Synthesis of Software Development Tools," *ACM SIGPLAN Notices*, vol. 39, p. 47, July 2004.

[19] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC Architecture Description Language and Tools," *Int. J. Parallel Program.*, vol. 33, no. 5, pp. 453—-484, Oct. 2005.

[20] 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," *Journal of Systems Architecture*, vol. 58, no. 8, pp. 318–337, May 2012.

[21] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Commun. ACM*, vol. 27, no. 10, p. 1013–1030, Oct. 1984.