# Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes*

Vladimir Herdt[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{vherdt,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—RISC-V is gaining huge popularity in particular for embedded systems. Recently, a SystemC-based *Virtual Prototype* (VP) has been open sourced to lay the foundation for providing support for system-level use cases such as design space exploration, analysis of complex HW/SW interactions and power/timing/performance validation for RISC-V based systems.**

**In this paper, we propose an efficient core timing model and integrate it into the VP core to enable fast and accurate performance evaluation for RISC-V based systems. As a case-study we provide a timing configuration matching the RISC-V HiFive1 board from SiFive. Our experiments demonstrate that our approach allows to obtain very accurate performance evaluation results while still retaining a high simulation performance.**

## I. Introduction

RISC-V is a free and open *Instruction Set Architecture* (ISA) [1], [2] that recently gained large momentum in both academia and industry, in particular for embedded systems. Embedded systems are typically small resource constrained systems that are highly specialized to implement application specific solutions, for example in the IoT domain. Hence, design flows for embedded systems require efficient and flexible design space exploration techniques to satisfy all application specific requirements. One important aspect of design space exploration is performance evaluation.

*Virtual Prototypes* (VPs) provide an industry-proven approach to obtain performance evaluation results through simulation [3]. VPs are essentially abstract models of the entire HW platform and predominantly created in SystemC TLM [4], [5]. To obtain performance evaluation results, a timing model is integrated with the VP to track timing information alongside the functional execution. Recently, an open-source RISC-V based VP has been presented [6] to enlarge the RISC-V ecosystem and provide the foundation for advanced SystemC-based analysis techniques for the RISC-V architecture. As a proof of concept, the RISC-V VP integrates a simple instruction accurate CPU core timing model that just assigns each instruction a fixed number of execution cycles.

**Contribution:** In this paper we propose an efficient core timing model and integrate it into the RISC-V VP core, to enable fast and accurate performance evaluation for RISC-V based systems[1]. Our timing model allows to consider pipelining, branch-prediction and caching effects, which makes it suitable for a large set of embedded systems. As case-study we provide a timing configuration matching the RISC-V HiFive1 board from SiFive. The HiFive1 core combines a five-level execution pipeline with an instruction cache and a branch prediction unit (for more information please consult the official documenation [7]). Our experiments demonstrate that our approach allows to obtain very accurate performance evaluation results (less than 5% mismatch on average compared against a real HiFive1 board) while retaining a high simulation performance (less than 20% performance overhead on average).

**Related Work:** There exist a number of RISC-V simulators such as the reference simulator Spike [8], RISCV-QEMU [9], RV8 [10], DBT-RISE [11] or Renode [12]. They differ in their implementation techniques and intended use-case which range from mainly pure CPU simulation (RV8, Spike) to full-system simulation (RISCV-QEMU, DBT-RISE) and support for multi-node networks of embedded systems (Renode). However, they are mainly designed to simulate as fast as possible to enable efficient functional validation of large and complex systems and predominantly employ DBT (*Dynamic Binary Translation*) techniques, i.e. translate RISC-V instructions on the fly to *x86_64*. This is however a trade-off as accurately modeling timing becomes much more challenging and thus these simulators do not offer cycle-accurate results.

A full-system simulator that can provide accurate performance evaluation results and recently got RISC-V support is *gem5* [13]. However, *gem5* integrates more detailed functional models (e.g. of the CPU pipeline) instead of abstract timing models that are sufficient for a pure performance evaluation and hence the simulation performance is significantly reduced.

Commercial VP tools, e.g. Synopsys Virtualizer or Mentor Vista, might also support RISC-V in combination with fast and accurate timing models but their implementation is proprietary.

Many recent state-of-the-art performance evaluation techniques focus on *Source-Level Timing Simulation* (SLTS) to enable a high-speed performance evaluation [14]–[18]. SLTS works by instrumenting the application source code with timing information that are typically obtained by a static analysis, then host compile the instrumented source code and natively execute the resulting binary without any emulation layer. However, due to the source-level abstraction it is very challenging to provide accurate models for peripherals and consider complex HW/SW interactions such as interrupts accurately.

Other recent approaches leverage DBT-based techniques. The papers [19], [20] discuss a combination of QEMU and the SystemC kernel. However, these methods either only provide rough performance estimates or incur significant synchronization overhead between QEMU and SystemC and lose the determinism of a SystemC simulation. Another DBT-based approach is [21] which integrates a timing model with a DBT-based execution engine to obtain near cycle accurate results. [22] is conceptually similar, but uses QEMU as execution backend. However, neither of these approaches are VP-based or target the RISC-V ISA.

## II. Core Timing Model for Fast and Accurate Performance Evaluation using VPs

Here we present our core timing model that enables a fast and accurate performance evaluation with VPs. We start with an overview.

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONFIRM under contract no. 16ES0565 and within the project VerSys under contract no. 01IW19001, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

[1]Visit http://www.systemc-verification.org/riscv-vp for our open source implementation and latest VP-based/RISC-V approaches.

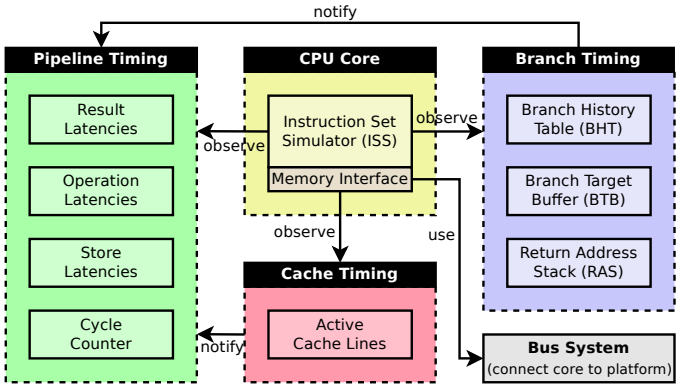


Fig. 1. Overview on our core timing model and the integration with the CPU core model.

### A. Overview

Our core timing model consists of a pipeline, branch predictor and cache timing model as shown in Fig. 1. They allow to consider timing information that depends on pipelining, branch-prediction and caching effects, respectively.

All three timing models are attached to the CPU core model, which essentially consists of an ISS and a memory interface. The ISS fetches, decodes and executes instructions one after another. The memory interface performs all memory access operations for the ISS, which involves instruction fetching and load/store operations. Thus, it wraps all operations into TLM transactions and puts them on the bus system, which routes them to the actual target (hence connects the ISS with the rest of the VP platform).

Executed instructions are observed in the ISS to update the pipeline and branch-prediction models accordingly. The branch predictor is only queried and updated on branch and jump instructions. The pipeline is updated for instructions that do not finish within a single execution cycle, as they can introduce latencies in case the next instructions depend on the result of a previous instruction before it is available. In this case the pipeline is *stalled*. Instruction with multiple cycles to completion are e.g. multiplication/division and load/store as well as access of special registers. We also observe the memory interface to update the cache timing model. The branch prediction and cache timing models have access to the pipeline model. In case of a branch misprediction or cache miss the pipeline is effectively *stalled*, which updates any pending result latencies in the pipeline and increases the cycle counter accordingly.

Please note, that timing models are more abstract compared to functional models. For example a cache timing model only needs to decide if a cache hit or miss occurs, thus the actual cache content is irrelevant (only the cached address ranges matter). Similarly, the pipeline only needs to decide if it is stalled and for how many cycles. Thus, it is not necessary to keep track of all the pipeline stages for each instruction, but only result dependencies between instructions and their computation latencies. These (functional) abstractions in the timing models significantly reduce the computational complexity and hence enable a high accuracy in combination with a low performance overhead. Next, we describe our three sub timing models in more detail.

### B. Pipeline Timing Model

The pipeline timing model keeps track of timing relevant execution dependencies between instructions to decide when and for how many cycles to stall the pipeline. A stall corresponds to *advancing* the pipeline timing model and is implemented by simply increasing

| Operation | Instr. | Reg. Operands | | | Cycle Counter | Result Latencies |
|---|---|---|---|---|---|---|
| | | RD | RS1 | RS2 | | |
| 1: a0=a1+a2 | add | a0 | a1 | a2 | 1 | [] |
| 2: a1=a0*a1 | mul | a1 | a0 | a1 | 2 | [a1 -> 5] |
| 3: a0=a0+a2 | add | a0 | a0 | a2 | 3 | [a1 -> 4] |
| 4: a2=a1*a2 | mul | a2 | a1 # | a2 | 8 | [a2 -> 5] |

# stall for 4 cycles due to the pending result latency on a1

Fig. 2. Example to illustrate the pipeline timing model.

the (total execution) cycle counter and accordingly decreasing the (local latency) counters of any pending dependencies (for illustration we show an example later in this section). We consider three kinds of dependencies: *result*, *operation* and *store*.

A result dependency denotes that an instruction N is accessing a register R that a previous instruction P writes to and P takes more than one cycle to compute the result, hence N has to wait for R. In this case we add a result latency for R to our pipeline model. In case a register is read/writen (during instruction execution) which has a result latency X, the pipeline model is stalled for X cycles. For better illustration Fig. 2 shows an example. It shows four instructions, two additions (*add*) and two multiplications (*mul*) where *mul* has a 5 cycle result latency and *add* has no result latency (i.e. the result of *add* is directly available for the next instruction in the pipeline). Fig. 2 (right side) reports the current cycle counter and result latencies after the instruction has finished. The cycle counter starts with zero and the result latencies are empty. Both *mul* instructions add a 5 cycle result latency on their destination register. After each instruction the pipeline is advanced one clock cycle, updating the cycle counter and pending result latencies accordingly. The fourth instruction additionally stalls the pipeline model for 4 cycles due to the pending 4 cycle result latency on register *a1*. Pending operation and store latencies are handled similarly to result latencies.

An operation dependency denotes that an instruction requires a specific resource and hence reserves it for a specific time T. Other instructions that require the same resource need to wait until the resource is released, i.e. T cycles pass. For example, the RISC-V HiFive1 board has only a single multiplication unit which has a 5 cycle result latency.

A store dependency denotes that an instruction has performed a memory write access, which is scheduled to be committed after X cycles. Load instructions, that access an overlapping memory address range before X cycles passed, are delayed for a specific configurable time by stalling the pipeline.

Please note, many instructions - like addition, shift, bit operations, etc. - can be effectively executed in a single cycle on embedded systems with a pipeline. For this common instructions no updates are necessary in the pipeline timing model, hence they effectively incur no performance overhead (besides incrementing the cycle counter), in particular if no active dependencies are pending.

### C. Branch Prediction Timing Model

Our branch predictor model is updated on branch and jump instructions. For jump instructions we distinguish the special cases of a (function) call and return. In case of a branch, the model has to predict if the branch will be taken and in case of a taken branch the address of the target instruction. For a jump (and its variants: call and return) it has to predict the target of the jump. A misprediction adds an execution penalty by *stalling* the pipeline (which mimics the timing effects of a pipeline flush due to the misprediction).

To perform the predictions, the branch predictor timing model manages three sub-models: BTB (*Branch Target Buffer*), BHT

```
i = 1;
while (i < 10)
  ++i;
```

```
li a0,1
li a1,10
loop:
bge a0,a1,end
addi a0,a0,1
j loop
end:
```

| loop-branch BHT entry | | Init | Loop Iteration | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | ... | 10 |
| Prediction | | / | T | N | N | N | N |
| Observation | | / | N | N | N | N | T |
| Counters | NN | 1 | 1 | 1 | 0 | 0 | 1 |
| | NT | 1 | 1 | 0 | 0 | 0 | 0 |
| | TN | 3 | 3 | 3 | 3 | 3 | 3 |
| | TT | 3 | 2 | 2 | 2 | 2 | 2 |
| History | | TT | NT | NN | NN | NN | TN |

Fig. 3. Example to illustrate the branch timing model.

(*Branch History Table*) and RAS (*Return Address Stack*). For the following description, we use these naming convention: *instr_pc* refers to the address of the branch/jump instruction, *target_pc* refers to the address of the branch/jump target, and *link_pc* is the address of the instruction following the branch/jump instruction in the code (due to potentially varying instruction lengths, *link_pc* typically cannot be inferred based on *instr_pc*).

The BTB is simply a fixed size mapping from *instr_pc* to (the predicted) *target_pc*. The size and replacement strategy is configurable. By default the lower bits of *instr_pc* are used to index the mapping. The BTB is queried to predict the target of branch and jump instructions and updated in case of mispredictions.

The RAS manages a fixed size stack of return addresses for the last function calls. RAS is updated on every *call* instruction and checked on every *return* instruction. In case of a misprediction an execution latency can be directly incurred or alternatively the BTB can be queried. RAS drops old entries in case a new entry is pushed on a full stack.

BHT tracks the recent execution history for branches, i.e. whether the branch has been taken or not, to predict the outcome of next branches. BHT is queried and updated on every branch instruction. A correct BHT prediction is followed by a BTB query to predict the actual branch target. A BHT misprediction incurs an execution cycle penalty by stalling the pipeline. Our BHT implementation has a fixed size mapping indexed by *instr_pc* (i.e. using the lower bits of the address of the branch instruction). Each mapping entry stores the recent (execution) history and a set of (prediction) counters for every execution history variant. The size of the mapping as well as the length of the execution history and counters is configurable.

As example consider a BHT configuration with two history bits and two counter bits for each mapping entry. Two history bits allow to distinguish four different cases (by interpreting a 1 as branch Taken and a 0 as Not taken). Thus, four two bit counters are provided (one for each possible history case). Each two bit counter can store four possible values: 0 (strongly Not taken), 1 (Not taken), 2 (Taken), and 3 (strongly Taken). Half of the values result in a branch Taken prediction (2,3) and the other half a branch Not taken prediction (0,1). Fig. 3 shows a concrete example for further illustration. It shows a simple C/C++ loop (left top) with corresponding (RISC-V) assembler code (left bottom) and a table that shows relevant data for the loop branch instruction (right side). It shows the current history and counter values for the BHT entry of the loop branch after each loop iteration (the initial state is chosen arbitrarily in this example) as well as the predicted and really observed branch direction. Based on the BHT entry in iteration N a prediction is made for iteration N+1. For example after iteration 1 the history is NT hence the NT counter decides the prediction N due to the counter value 1. Based on the observation N in iteration 2, the counter NT is updated accordingly (decremented) and the N is shifted into the history to obtain the BHT entry state after iteration 2.

### D. Cache Timing Model

The cache timing model is queried on memory access operations to cached regions. Typically, an embedded system provides at least an instruction cache, hence the timing model will be queried for every instruction fetch. Thus, the timing model should be very efficient in deciding a cache hit or miss to reduce the performance overhead. A cache miss adds an execution penalty by *stalling* the pipeline (which mimics the timing effects of waiting for a cache line fetch).

We provide a model matching a two-way associative cache with LRU replacement strategy which is a common choice for embedded systems. However, our cache timing model can be easily generalized to an N-way associative cache with a different replacement strategy. The cache table consists of cache lines which in turn consist of two entries (for a two-way associative cache). We only keep track of the memory address that the entries represent, since the actual data is not relevant for our timing model. The number of lines and their size in bytes is configurable. Each memory access address is translated to an index into the cache table by taking the lower bits of the address. Finally, we compare the memory access address with the address of both cache line entries to decide if the memory access is a cache hit or miss. A miss (none of the entries does match) updates one of the two entries (i.e. associate the entry with a new address range which matches the access address) based on the replacement strategy.

## III. Results and Future Work

We have implemented our proposed core timing model and integrated it into the open-source RISC-V VP. Furthermore, we configured our core timing model to match the RISC-V HiFive1 board. This section presents experiments to evaluate the accuracy and performance overhead of our model. We obtain accuracy results by comparing against a real RISC-V HiFive1 board and we obtain the performance overhead results by comparing against the RISC-V VP without integrating our core timing model. All experiments are evaluated on a Linux machine with an Intel i5-7200U processor.

For evaluation we use the *Embench* benchmark suite [23][2]. *Embench* is a freely available standard benchmark suite specifically targeting embedded devices. It is a collection of real instead of synthetic programs (like *Dhrystone* or *Coremark*) to ensure that more realistic workloads are considered. It contains for example benchmarks that perform error checking, hashing, encryption and decryption, sorting, matrix multiplication and inversion, JPEG and QR-code as well as regex and state machine operations. The benchmarks have a varying degree of computational, branching and memory access complexity [24]. Each benchmark starts with a *warm-up* phase that executes the benchmark body a few times to warm-up the caches. Then, the real benchmark starts by executing the benchmark body again several more times. The number of iterations is configurable and depends on the CPU frequency. Furthermore, it varies between different benchmarks to ensure a mostly uniform runtime complexity across the benchmark suite. We set the CPU frequency constant to 320 MHz to match the HiFive1 board.

Table I shows the results. Starting from the left, the columns show the benchmark name (Column 1), the *Lines of Code* (LoC) in C (Column 2) and RISC-V Assembler (Column 3) and the number of executed RISC-V instructions (Column 4, measured by the RISC-V VP). Then, Table I shows the simulation time in seconds for running the benchmark on the RISC-V VP without (Column 5) and with (Column 6) integrating our core timing model, respectively. The simulation time denotes how long (wall time) it takes to run the

[2]We omitted three of the nineteen benchmarks from the comparison due to problems in executing them on the HiFive1 board.

TABLE I
EXPERIMENT RESULTS – ALL SIMULATION TIME RESULTS REPORTED IN SECONDS. THE COLUMNS ARE NUMBERED TO ENABLE REFERENCING THEM.

| 1: Benchmark | LoC | | 4: #Instrs. | VP | VP + Core Timing Model | | HiFive1 Board | Difference | |
|---|---|---|---|---|---|---|---|---|---|
| | 2: C | 3: ASM | | 5: Time | 6: Time | 7: #Cycles | 8: #Cycles | 9: #Cycles | 10: Time |
| aha-mont64 | 306 | 4,887 | 1,083,316,249 | 27.46 | 32.05 | 1,130,126,565 | 1,192,518,796 | -5.2% | +16.7% |
| crc32 | 256 | 3,890 | 1,093,865,270 | 30.02 | 33.51 | 1,283,897,183 | 1,284,037,753 | -0.0% | +11.6% |
| edn | 440 | 4,655 | 1,009,188,407 | 30.81 | 34.12 | 1,772,167,527 | 1,847,419,217 | -4.1% | +10.7% |
| matmult-int | 330 | 4,076 | 1,039,623,303 | 30.61 | 33.68 | 1,761,624,764 | 1,780,530,752 | -1.1% | +10.0% |
| minver | 343 | 6,262 | 948,801,750 | 32.90 | 39.78 | 1,346,343,318 | 1,298,778,561 | +3.7% | +20.9% |
| nbody | 322 | 6,427 | 995,233,372 | 32.25 | 39.18 | 1,356,400,421 | 1,364,440,490 | -0.6% | +21.5% |
| nettle-aes | 1,173 | 5,831 | 1,203,436,482 | 34.82 | 42.10 | 1,219,069,707 | 1,223,022,326 | -0.3% | +20.9% |
| nettle-sha256 | 503 | 6,252 | 1,078,331,659 | 36.65 | 43.57 | 1,101,319,048 | 1,106,544,095 | -0.5% | +18.9% |
| picojpeg | 2,337 | 9,482 | 957,883,248 | 30.23 | 36.09 | 1,208,694,004 | 1,158,481,492 | +4.3% | +19.4% |
| qrduino | 1,091 | 7,726 | 1,142,278,239 | 34.28 | 38.58 | 1,399,022,255 | 1,544,185,605 | -9.4% | +12.5% |
| sglib-combined | 2,002 | 6,501 | 906,977,088 | 29.28 | 35.69 | 1,190,579,040 | 1,209,895,484 | -1.6% | +21.9% |
| slre | 661 | 5,518 | 1,110,788,540 | 35.71 | 43.69 | 1,443,734,072 | 1,272,198,737 | +13.5% | +22.3% |
| st | 271 | 6,643 | 1,051,545,074 | 33.60 | 40.94 | 1,425,869,796 | 1,333,455,328 | +6.9% | +21.8% |
| statemate | 1,456 | 5,866 | 789,501,877 | 27.68 | 38.04 | 988,087,257 | 1,077,693,835 | -8.3% | +37.4% |
| ud | 250 | 4,774 | 841,910,324 | 27.02 | 30.55 | 1,241,243,866 | 1,184,148,661 | +4.8% | +13.1% |
| wikisort | 1,021 | 8,482 | 833,317,333 | 26.65 | 32.89 | 1,057,727,371 | 1,187,236,518 | -10.9% | +23.4% |

benchmark on the VP. The next two columns give the number of execution cycles for the benchmark as reported by the RISC-V VP with integrating our core timing model (Column 7) and the real HiFive1 board as comparison (Column 8). RISC-V provides a special register that holds the current number of execution cycles. We query this register before and after running the benchmark to obtain the spent execution cycles. Finally, the last two columns report the difference in the estimated number of execution cycles with the really measured execution cycles (Column 9) and the performance overhead of core timing model (Column 10).

It can be observed that our core timing model provides a very accurate estimation of the number of execution cycles. In summary, we observed a minimum and maximum difference of 0.0% and 13.5%, respectively, with an average of 4.7%. These results demonstrate the effectiveness of our approach.

The remaining accuracy gap is mostly for two reasons: 1) incomplete specification of the timing related functionality of the HiFive1 board (e.g. the replacement strategy in the branch prediction buffers), and 2) because we primarily focus on the core timing model in this paper and thus approximate the timing effect of a cache miss (which causes a fetch of the cache line from the flash memory through the bus system) with a fixed number of cycles (an average value that we sampled on the HiFive1 board). We believe, that with a more complete specification we can adapt the configuration of our core timing model appropriately to obtain even more accurate results. The second issue can be addressed by integrating a (complementary) more accurate timing model for the (TLM-based) bus system.

Besides high accuracy, our core timing model also retains the high simulation performance of the VP by introducing a reasonably small performance overhead (average simulation performance of 27 MIPS compared to 32 MIPS). We observed a minimum and maximum overhead of 10.0% and 37.4%, respectively, with an average of 18.9%. The performance overhead primarily depends on the number of data flow dependencies between subsequent instructions as well as number of branching instructions. These parameters are highly dependent on the actual benchmark. Furthermore, we have an additional (almost constant) time overhead for each instruction fetch to check (and potentially update) if the fetch results in a cache miss and hence incurs an execution time penalty.

For future work we plan to boost the performance by investigating further abstractions of our timing model and generation of (e.g. basic block) summaries to simplify and reduce the number of model updates as well as consider integration of DBT into the VP's ISS. Moreover, we plan to build configurations matching other RISC-V cores and extend our timing model to consider additional features like out-of-order execution. Finally, we also want to consider simulation-based and formal test-generation methods, such as [25]–[27], for testing the timing model.

## REFERENCES


[1] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[2] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[3] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[4] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[5] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[6] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.

[7] "SiFive FE310-G000 Manual v2p3," https://sifive.cdn.prismic.io/sifive%2F4d063bf8-3ae6-4db6-9843-ee9076ebadf7_fe310-g000.pdf.

[8] "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim.

[9] "RISCV-QEMU," https://github.com/riscv/riscv-qemu.

[10] "RV8," https://rv8.io, accessed: 2018-05-13.

[11] "DBT-RISE," https://github.com/Minres/DBT-RISE-Core.

[12] "Renode," https://renode.io/.

[13] 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*, pp. 1–7, 2011.

[14] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Müller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems," in *DATE*, 2015, pp. 1698–1707.

[15] S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, and O. Bringmann, "Context-sensitive timing simulation of binary embedded software," in *CASES*, 2014.

[16] Z. Wang, K. Lu, and A. Herkersdorf, "An approach to improve accuracy of source-level tlms of embedded software," in *DATE*, 2011, pp. 1–6.

[17] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann, "Accurately timed transaction level models for virtual prototyping at high abstraction level," in *DATE*, 2012.

[18] Z. Wang and J. Henkel, "Fast and accurate cache modeling in source-level simulation of embedded software," in *DATE*, 2013, pp. 587–592.

[19] M. Chiang, T. Yeh, and G. Tseng, "A QEMU and SystemC-based cycle-accurate ISS for performance estimation on soc development," *TCAD*, pp. 593–606, 2011.

[20] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, "Fast virtual prototyping for embedded computing systems design and exploration," in *RAPIDO*, 2019, pp. 3:1–3:8.

[21] 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 *SAMOS*, 2010, pp. 1–10.

[22] D. Thach, Y. Tamiya, S. Kuwamura, and A. Ike, "Fast cycle estimation methodology for instruction-level emulator," in *DATE*, 2012, pp. 248–251.

[23] "Embench: A modern embedded benchmark suite," https://www.embench.org/.

[24] J. Bennett, P. Dabbelt, C. Garlati, G. S. Madhusudan, T. Mudge, and D. Patterson, "Embench: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative," https://content.riscv.org/wp-content/uploads/2019/06/9.25-Embench-RISC-V-Workshop-Patterson-v3.pdf.

[25] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, 2017, pp. 1–8.

[26] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Maximizing power state cross coverage in firmware-based power management," in *ASP-DAC*, 2019, pp. 335–340.

[27] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019, pp. 188:1–188:6.