Adaptive Simulation with Virtual Prototypes for RISC-V: Switching Between Fast and Accurate at Runtime

Vladimir Herdt[1] Daniel Große[1,2] Sören Tempel[3] Rolf Drechsler[1,3]
[1]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[2]Chair of Complex Systems, Johannes Kepler University Linz, Austria
[3]Institute of Computer Science, University of Bremen, Bremen, Germany
Vladimir.Herdt@dfki.de, daniel.grosse@jku.at, tempel@uni-bremen.de, drechsle@informatik.uni-bremen.de

***Abstract*—Recently, Virtual Prototypes (VPs) were introduced for the emerging RISC-V *Instruction Set Architecture* (ISA) and become an important part of the growing RISC-V ecosystem. A central component of the VP is the *Instruction Set Simulator* (ISS). VPs should provide a high performance and at the same time yield accurate results, which are conflicting requirements.**

To tackle this problem, we present an efficient VP-based adaptive simulation that is tailored for the RISC-V ISA and allows to seamlessly switch the accuracy setting in the ISS at runtime. This enables to selectively simulate the application as fast as possible and as accurate as necessary. In this paper we focus on the performance impact of different accuracy settings and leave the evaluation of accuracy results for future work. Our RISC-V experiments demonstrate that up-to 543x speed-up is possible with a JIT-based setting in the ISS.

I. Introduction

RISC-V is an open and royalty-free *Instruction Set Architecture* (ISA) that features an extremely modular design and gained enormous momentum in recent years. RISC-V became a game changer for embedded systems in several applications such as *Internet-of-Things* (IoT) to build highly specialized solutions. Beside a thorough functional validation, performance evaluations and optimizations are crucial to meet the application specific demands.

Therefore, mainly simulation-based methods are employed. The RISC-V ecosystem provides several simulators that enable SW execution early in design flow. In particular, *Virtual Prototypes* (VPs) play a very important role here [1], [2]. VPs are essentially abstract models of the entire HW platform and predominantly created in SystemC TLM (*Transaction Level Modeling*) [3], [4]. A central component of the VP is the *Instruction Set Simulator* (ISS), which is an abstract model of the processor (and hence responsible to fetch, decode and execute instructions one after another). By integrating appropriate timing models, VPs can be used for performance evaluations.

In general, VPs should provide a high simulation performance (to deal with complex SW) and at the same time yield accurate results (to do performance evaluations and optimizations), which are two conflicting requirements. In order to deal with this problem, a common approach is to build two (or more) separate VP models with different accuracy settings and use them accordingly. A fast VP model to perform functional validation and an accurate VP model to do performance evaluations. However, this approach becomes highly inefficient when only (a small) part(s) of the SW needs to be analyzed accurately, e.g. the execution of a specific kernel module in the Linux *Operating System* (OS), or a particular recurring interaction with a peripheral. Furthermore, due to the rising (SW) complexity this problem is further amplified. Thus, it is crucial to build efficient solutions that can adapt the accuracy in the VP on demand at runtime.

Contribution: We present an efficient VP-based adaptive simulation tailored for RISC-V that allows to seamlessly switch the accuracy setting in the ISS at runtime. The configuration for switching, i.e. when to perform a particular switching, is user-provided. Our approach allows to scale between a high-speed *Just-in-Time* (JIT) compilation-based setting down to a *Cycle-Accurate* (CA) interpreter-based setting in the ISS and is tailored for SystemC-based VPs. Carefully designed interfaces allow to regularly synchronize with the SystemC kernel and to collect required timing information in the ISS. In addition, the ISS is designed to work on a single execution state that stays consistent between switching. Thus, switching is a very lightweight operation that can be executed with a minimal performance overhead. In this paper we focus on the performance impact of different accuracy settings and leave evaluation of accuracy results for future work. Our RISC-V experiments demonstrate that up-to 543x speed-up is possible with a JIT-based setting in the ISS. To the best of our knowledge, our solution is the only freely available SystemC-based VP that is capable of booting Linux[1].

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001, and within the project Scale4Edge under contract no. 16ME0127.

II. Related Work

Considering RISC-V, there exist a number of simulators such as the reference simulator SPIKE [5], RISCV-QEMU [6], RV8 [7], DBT-RISE [8] or Renode [9]. They differ in their implementation techniques and intended use-case which range from mainly pure CPU simulation (SPIKE, RV8) to full-system simulation (QEMU, DBT-RISE) and even support for multi-node networks of embedded systems (Renode). However, they are mainly designed to simulate as fast as possible and thus do not offer a CA performance evaluation. A full-system simulator that can provide accurate performance evaluation results, and recently got RISC-V support, is *gem5* [10], [11]. Another viable option is the open source RISC-V VP [12], which is implemented in SystemC TLM. However, neither gem5 nor RISC-V VP support JIT-based techniques and hence the performance is significantly reduced compared to the high-speed simulators. In addition, none of the RISC-V simulators supports switching the timing accuracy setting at runtime. 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. Finally, there are approaches to formalize the RISC-V ISA semantics, e.g. SAIL [13] and GRIFT [14], which also provide or can generate simulator backends. However, they only offer limited performance and are not designed for timing accurate simulations.

Looking beyond RISC-V, the general idea of using a configurable simulator is not new. For example, [15] present a (full-system) simulator for the PowerPC architecture that can be extensively configured through compile time flags and command line arguments. In [16] a JIT-based re-implementation is described to speed-up pure-functional simulations. Fine-grained configuration of simulators (via generation of different models) to meet application specific demands is discussed in [17]. [18] present an adaptive simulation that can learn an approximate timing model (in combination with a JIT-based execution) to deliver approximate timing results. However, a VP-based runtime adaptive simulation that scales between CA and high-performance JIT-based execution tailored for RISC-V is not available to the best of our knowledge.

[1]Visit http://www.systemc-verification.org/risc-v to find our open source RISC-V VP and also our most recent RISC-V related approaches.

There have been proposed some approaches for runtime adaptive simulations in the context of VPs. [19] proposes to create TLM (bus) models at different levels of abstraction and switch between them at runtime (based on a user-provided configuration) to obtain fast and accurate results. [20] is conceptually similar but also considers power modeling in addition to performance evaluations. These methods are complementary to our approach, since they consider the accuracy of TLM transactions while we focus on the ISS. Other approaches leverage runtime adaptive simulations to switch between an ISS-based and RTL-based [21] as well as gate-level simulation [22] for the purpose of fast and accurate fault injection. These approaches operate on a different abstraction level than ours.

III. Background: SystemC and TLM

SystemC TLM is an industry-proven modeling standard to create VPs [1]. SystemC is not a new language, rather a C++ class library which includes an event-driven simulation kernel [3]. The structure of a SystemC design is described with ports and modules, whereas the behavior is modeled in processes which are triggered by events. Communication between SystemC modules is abstracted using TLM transactions at the cost of timing accuracy, but significant improvements in simulation speed, i.e. up to a factor of 1,000 in comparison to an RTL simulation. Transactions are routed on a bus system based on their address from an initiator to a target socket as defined in the SystemC TLM-2.0 standard.

Two optimization techniques are commonly utilized to improve the SystemC simulation performance: *Direct Memory Interface* (**DMI**) and *Time Quantum* (**TQ**). DMI allows to bypass the bus system for specific address ranges to directly and very efficiently access the memory through a pointer (instead of routing a TLM transaction). TQ allows to (locally) postpone the synchronization with the SystemC kernel by running ahead of the (global) simulation time for a configurable TQ value (e.g. to avoid synchronization after every executed instruction in the ISS).

IV. Adaptive VP-based Simulation

Here we present our VP-based adaptive simulation approach. We start with an overview (Section IV-A), then present the implementation of JIT (Section IV-B) and switching (Section IV-C).

A. Core Architecture

Fig. 1 shows an architecture overview with the most relevant components and relations between them. At the center is the ISS. The ISS essentially consists of three parts: state, interpreter and JIT. The state includes the (RISC-V) ISA state, i.e. *Program Counter* (PC) and register values, as well as interpreter and JIT related data structures. The interpreter fetches, decodes and executes instructions one after another. JIT enables to speed-up execution by translating instructions directly into host assembler instructions (alongside the interpreter execution), and re-using them henceforth.

The memory interface (Fig. 1 center) is leveraged by the ISS for instruction fetching as well as processing of load and store instructions. It works as follows: First the memory access address is translated from a virtual to a physical (**v2p**) address by using the *Memory Management Unit* (MMU). In systems without MMU or if the MMU is currently disabled, v2p is a no-op (since the system already works with physical addresses). Then the resulting physical address is compared against the available DMI address ranges, i.e. list of (start,end) address pairs. If it does match, the memory access is directly processed via DMI, thus bypassing the TLM bus. Otherwise, the memory access is translated into a TLM transaction and routed through the bus to the target.

Both the ISS and memory interface can update the core timing model. It provides a set of interface functions tailored for the interpreter- and JIT-based execution settings. The timing model utilizes SystemC quantum keeper to synchronize with the SystemC kernel after a (configurable) TQ.

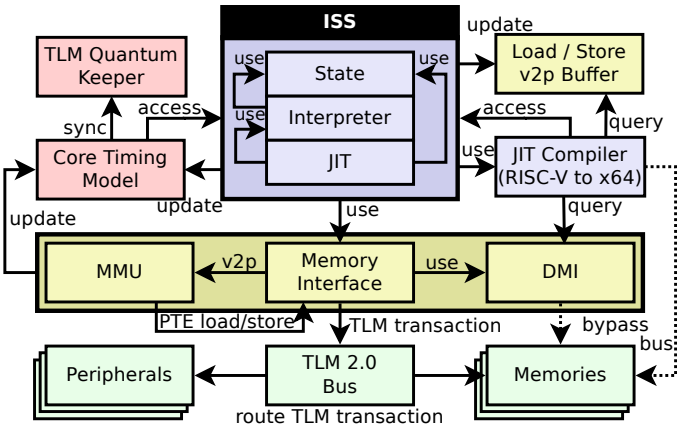

Fig. 1: Architecture overview

The JIT setting in the ISS leverages a JIT compiler (that in our case translates RISC-V instructions to x86_x64 assembly). The JIT compiler has access to the ISS state that is shared with the interpreter-based setting. It generates x86_x64 assembly instructions that directly read and write that shared state to ensure that the state stays consistent between switching. For optimization, the JIT compiler uses DMI.

Depending on the current setting, the main ISS simulation loop performs either an interpreter-based or JIT-based execution step in each iteration. In the following we present more details on the JIT implementation and how switching is implemented.

B. JIT-based Execution Setting

Fig. 2 shows the JIT-based execution step. Compared to the interpreter, a JIT step operates on *Basic Blocks* (BBs) instead of single instructions. A BB is essentially a linear sequence of instructions that ends with a jump or branch. First, the (virtual) PC is translated into a *Physical PC* (PPC) address in Line 3. In case the MMU is disabled or not available, this is a no-op.

If a JIT BB is already available for PPC, it will be directly executed (Line 9). Otherwise, the BB will be collected, compiled and stored for subsequent reference (Line 11). During collection, the readily available interpreter functions are used to directly execute, the one after another collected, basic block instructions. For performance reasons we manage two mappings for the BB lookup (Line 7). A fixed size (4096) *Direct Mapped Cache* (DMC) that is queried first and then a hash map fallback if PPC is not in DMC.

The BB is executed by calling the JIT compiled function (Line 15). It is compiled to directly operate on the ISS state, in particular the registers and PC. Hence, after the function returns (Line 15), the ISS state is fully consistent with the interpreter. Thus, execution can swap directly and arbitrary between the JIT-based and interpreter-based setting after each step.

A trap can interrupt the JIT execution (and hence cause a BB to leave early in Line 15) and will be immediately processed after the trapped instruction. Please note, a trap can also be caused by the v2p function of the MMU in Line 3. After each JIT step interrupts are checked and processed (Line 12).

Finally, we combine trace-based and traditional link optimization techniques (which are common in optimizing JIT-based compilers) to link multiple BBs together at runtime. We place exit guards in the JIT code at backward jumps to ensure a regular synchronization with the SystemC kernel. Both link optimizations work in-place, i.e. the new compiled function overwrites the existing BB function (Line 19). We start the link optimization process once the execution count of a BB reaches a certain threshold.

C. Switching Settings

A switching configuration essentially provides the initial start setting as well as settings for specific parts of the application, e.g.

```
function ISS::jit_run_step()
    try                 // PPC: Physical PC, v2p: virtual
        PPC ← mmu::v2p(PC, FETCH)          // to physical
    catch (SimulationTrap e)                // setup jump to
        process_trap(e)             // trap handler (sets PC)
        return      // continue with PC at trap handler
    bb ← BB_lookup(PPC)             // if JIT compiled BB
    if bb ≠ nil then                // is available for PPC
        BB_exec(bb)                 // then, execute the BB
    else        // otherwise, run the interpreter and
        BB_collect(PPC)       // collect the BB alongside
    process_interrupts()          // similar to interpreter

function ISS::BB_exec(bb)
    // execute compiled bb, return early on trap
    num_cycles ← bb::compiled_fn(PC)
    // update timing, maybe sync. with SystemC
    timing::add_cycles(num_cycles)
    // optional: try to optimize bb in place
    BB_link_optimize(bb)
```

Fig. 2: ISS JIT setting execute single step

functions or PC ranges. This can be implemented by providing switching commands, which are pairs of (trigger-PC, target-setting). It means to switch at runtime to the target setting in the ISS, when PC equals trigger-PC.

In general, switching can be implemented in one of two ways: 1) either by embedding the switching commands into the SW application, or 2) by passing/embedding the switching commands into the ISS. Both approaches can be implemented in different ways and have certain trade-offs. In general, embedding switching commands into the SW provides very precise control in combination with a high performance. However, it requires to re-compile the SW (which also means that source code must be available) and modifies the SW (which means the analyzed SW binary is slightly different to the actual SW binary). On the other hand, passing them to the ISS is more flexible and does not require to modify the SW application. However, it can cause additional performance overhead and may not always be applicable (e.g. when executing an application in Linux, it may not be clear at which address the application will be loaded).

In this paper we used the embedding approach for switching by using a custom system call (syscall). In RISC-V a syscall is triggered by the *ECALL* instruction and arguments are (typically) passed in the registers a7 (syscall number) as well as a0 to a3 (arguments for the syscall to select target setting). The syscall (instruction) is then intercepted in the ISS and processed accordingly.

V. Experiments

We have implemented our approach for adaptive simulation on top of the open source SystemC-based RISC-V VP [23], [24]. We used the *asmjit* [25] library as backend in our JIT compiler. First, we present a performance evaluation in Section V-A. It shows the performance impact of the different ISS execution settings in the SystemC-based VP, and it compares the performance against other RISC-V simulators. Then (Section V-B), we discuss a Linux-based case study that demonstrates the applicability and benefits of our approach. All experiments have been performed on a Linux system with an Intel i5-7200U processor. Please note, we focus on the performance impact of different accuracy settings and leave the evaluation of accuracy results for future work.

A. Performance Evaluation

We consider six different ISS settings in the VP for this evaluation: Base, +DMI, +TQ, +IA, +JBB, +JL. *Base* provides the most accurate simulation setting. It neither uses TQ nor DMI and integrates an example CA timing model (that considers pipeline, branch prediction and caching effects [26]). *+DMI* extends *Base* with DMI optimization for instruction fetching and main memory accesses. *+TQ* extends *+DMI* with TQ optimization to synchronize only every 10,000 cycles with the SystemC kernel. *+IA* modifies *+TQ* to replace the CA timing model with a lightweight *Instruction-Accurate* (IA) timing model (i.e. use a fixed number of cycles per instruction). *+JBB* extends *+IA* with a JIT-based ISS that translates single BBs. *+JL* extends *+JBB* with common BB link optimization. We use a JIT quantum of 10,000 cycles for *+JL* to match the TQ. In addition to comparing the ISS settings, we provide a performance comparison to four RISC-V simulators (designed for different use-cases, see Section II) in order to better relate the VP performance results: QEMU, SPIKE, gem5 and SAIL.

We use *Embench* [27], a standard benchmark suite specifically targeting embedded devices, for the evaluation. It is a collection of real instead of synthetic programs and the benchmarks have a varying degree of computational, branching and memory access complexity.Table I shows the results. The first column shows the benchmark name. The next four columns show the number of executed instruction (#X, measured on the RISC-V VP with +IA setting) and their classification into computational (#C), memory access (#M) and jump/branch (#J) instructions. The remaining ten columns report the execution time (in seconds) for the respective simulator and setting. Please note that we compare the performance difference of each setting in isolation in this performance evaluation, thus no actual accuracy switching occurs here (though switching support is available in the ISS). The last row shows the overall average performance in MIPS (*Million Instructions Per Second*).

It can be observed that both +DMI and +TQ optimizations have a significant impact on the execution performance, between 1.5x and 1.8x as well as 3.0x and 4.6x, depending on the benchmark. The CA timing model has an overhead of around 58% to the IA timing model (based on the MIPS). It needs to regularly update (per instruction) and synchronize the timing effects with the SystemC-based simulation. +JBB is already up to 37.6x faster than +IA (see the *nettle-sha256* benchmark). +JL further boosts performance, it is up to 7.8x and 57.2x faster than *+JBB* and *+IA*, respectively (see the *nsichneu* and *nettle-sha256* benchmark). +JL has an average of around 1040 MIPS. Overall, this evaluation shows the performance differences between the ISS settings and demonstrates the high performance of the JIT-based setting (+JL is up to 543x faster than the CA *Base* setting, see the *aha-mont64* benchmark).

QEMU is a very mature and highly optimized simulator, specifically designed for high-speed pure functional simulations. Thus, +JL is expectedly slower than QEMU (though only around 1.4x, based on MIPS) due to the performance overhead of the SystemC kernel and the more accurate SystemC-based simulation timing.

Compared to SPIKE (high-speed interpreter-based reference simulator) and gem5 (designed for architectural exploration and analysis), the +JL setting is 7.4x and 416.0x faster, respectively (1040 MIPS compared to 141.4 MIPS and 2.5 MIPS). gem5 is a large and rather generic platform which aims to support different architectures besides RISC-V which causes additional overhead.

Compared to SAIL, +JL is consistently much faster (more than three orders of magnitude), which is not unexpected due to the focus on formal reasoning of SAIL.

B. Switching Case Study: Linux Application

As a case study, we consider a Linux-based embedded application. It consists of two parts: a userland program that contains the application logic, and a Linux kernel driver which provides access to the peripherals. The application logic contains a processing loop that copies data from a sensor peripheral to a UART peripheral. The processing loop is repeated multiple times and the sensor generates random data. The userland program accesses the peripherals through a character device provided by the Linux kernel driver. The character device in turn enables to interact with the peripherals through memory mapped I/O. Please note, this embedded application acts as an

TABLE I: Experiment results - all execution times reported in seconds, number of executed instructions (#X) in Billions (B). #C, #M and #J classify #X into computational, memory access and jump/branch instructions. T.O. = Time Out (2h = 7200s).

Benchmark	#X	Instruction Types			RISC-V VP						Other RISC-V Simulators			
		#C	#M	#J	Interpreter-based				JIT-based					
					Base	+DMI	+TQ	+IA	+JBB	+JL	QEMU	SPIKE	gem5	SAIL
aha-mont64	4.53B	90%	0%	10%	1195	818	226	121	9.9	2.2	1.2	18.9	1686	T.O.
crc32	4.18B	75%	13%	12%	1183	796	174	106	9.3	2.4	2.6	18.9	1857	T.O.
cubic	6.80B	71%	17%	12%	2047	1334	412	244	28.6	10.8	5.9	180.2	1711	T.O.
edn	3.56B	61%	29%	10%	1092	704	178	95	7.9	2.7	1.9	17.3	1641	T.O.
huffbench	2.47B	53%	26%	21%	798	493	117	73	9.6	3.3	1.9	12.2	1200	7083
matmult-int	3.18B	50%	38%	12%	1066	622	150	94	7.9	2.5	2.0	16.6	1502	T.O.
minver	5.01B	66%	16%	18%	1458	958	277	183	20.8	5.3	3.8	68.8	2198	T.O.
nbody	3.11B	76%	9%	15%	864	570	171	106	11.3	2.6	1.8	53.7	890	T.O.
nettle-aes	4.45B	78%	20%	2%	1279	813	218	139	4.3	2.5	2.1	21.5	ERR.	T.O.
nettle-sha256	4.05B	84%	14%	2%	1146	710	207	143	3.8	2.5	2.0	108.1	1812	T.O.
nsichneu	2.24B	0%	55%	45%	732	423	132	86	58.4	7.5	14.5	75.9	1190	T.O.
picojpeg	3.70B	60%	29%	11%	1134	684	201	121	9.4	3.4	2.3	29.0	1596	T.O.
qrduino	2.87B	64%	20%	16%	869	543	140	90	9.5	3.1	2.6	15.4	1273	T.O.
sglib-combined	2.49B	37%	37%	26%	794	470	136	86	12.0	4.1	2.8	15.2	ERR.	T.O.
slre	2.91B	44%	31%	25%	893	523	157	101	13.7	3.2	2.5	16.3	1241	T.O.
st	3.88B	76%	10%	14%	1116	657	211	141	10.6	3.5	2.6	37.3	1090	T.O.
statemate	1.79B	39%	52%	9%	578	326	107	69	4.4	3.0	2.0	10.2	810	6343
ud	3.32B	62%	19%	19%	984	586	176	113	12.9	11.2	2.2	20.5	1414	T.O.
wikisort	1.30B	58%	25%	17%	389	227	69	45	4.3	2.1	1.4	15.0	ERR.	4473
Overall Performance (average MIPS):					3.3	5.4	19.2	30.8	387.3	1040	1486	141.4	2.5	0.3

example that, in addition to the HW and SW, also integrates the OS and driver layer. It starts after Linux has booted.

The goal in this example scenario is to carry out a performance analysis of the embedded application, which includes the interaction with the Linux kernel driver and low-level peripheral access, inside of the Linux execution environment. Our approach allows to do so very efficiently by using the high performance +JL setting for the boot (and shutdown) of Linux and the CA Base setting for executing the application. Therefore, we instrumented the userland program with inline assembler instructions, that trigger a specific system call for switching the execution setting (i.e. choice 1 from Section IV-C). Upon entry, the Base setting is selected and upon exit, the +JL setting is restored. By switching settings, our approach reduces the performance analysis time by a factor of 24.3x while still executing the application with the CA timing. The overall execution time reduces from 63.2 to 2.6 seconds. The accuracy tradeoff for the performance gain is a less accurate timing result for the Linux boot (and shutdown), though this has no impact on our goal in carrying out a performance analysis of the actual application.

Please note, the general idea of this case-study is applicable independent of the actual employed CA timing model (which is highly platform dependent due to its non-functional nature). Finally, switching settings is a very lightweight operation with essentially negligible performance overhead that can be placed on a very fine granular basis for precise control.

VI. Conclusion and Future Work

We presented an efficient VP-based adaptive simulation for RISC-V that allows to seamlessly switch the accuracy setting in the ISS at runtime. We demonstrated the high performance of our JIT-based setting and the efficacy in reducing the performance evaluation time on a Linux case study. Next, we plan to:

- Provide complete accuracy results and evaluate the VP-based accuracy against other RISC-V simulators.
- Use a more extensive benchmark set for the performance evaluation that also considers sophisticated OS benchmarks.
- Consider advanced JIT optimizations for further speed-up, keeping a regular synchronization with the SystemC kernel.
- Integration of adaptive TLM models (beside the ISS) to enable adaptive simulations with full platform settings.
- Support for easy specification and integration of custom RISC-V instruction set extensions with the JIT engine.

References

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

[2] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

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

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

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

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

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

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

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

[10] 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, pp. 1–7, 2011.

[11] "gem5," https://github.com/gem5/gem5.

[12] "RISC-V virtual prototype," https://github.com/agra-uni-bremen/riscv-vp.

[13] "Riscv sail model," https://github.com/rems-project/sail-riscv.

[14] "GRIFT - galois RISC-V ISA formal tools," https://github.com/GaloisInc/grift.

[15] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, "Mambo: A full system simulator for the powerpc architecture," *SIGMETRICS Perform. Eval. Rev.*, 2004.

[16] A. Kumar, A. Gheith, and M. Kistler, "Experiences with dynamic binary translation in a full system simulator," in *IPDPS*, 2013, pp. 2168–2175.

[17] G. Martin, N. Nedeljkovic, and D. Heine, *Configurable, Extensible Processor System Simulation*. Springer US, 2010, pp. 293–308.

[18] N. Topham, B. Franke, D. Jones, and D. Powell, *Adaptive High-Speed Processor Simulation*. Springer US, 2010, pp. 145–159.

[19] M. Radetzki and R. S. Khaligh, "Accuracy-adaptive simulation of transaction level models," in *DATE*, 2008, pp. 788–791.

[20] G. Beltrame, D. Sciuto, and C. Silvano, "Multi-accuracy power and performance transaction-level modeling," *TCAD*, vol. 26, 2007.

[21] D. Mueller-Gritschneder, U. Sharif, and U. Schlichtmann, "Performance and accuracy in soft-error resilience evaluation using the multi-level processor simulator ETISS-ML," in *ICCAD*, 2018, pp. 1–8.

[22] M. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *HPCA*, 2009, pp. 105–116.

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

[24] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, 2020.

[25] "Complete x86/x64 JIT and AOT assembler for C++," https://github.com/asmjit/asmjit.

[26] V. Herdt, D. Große, and R. Drechsler, "Fast and accurate performance evaluation for RISC-V using virtual prototypes," in *DATE*, 2020, pp. 618–621.

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