SCIENCE CHINA
Information Sciences

January 2022, Vol. 65 110201:1–110201:17
https://doi.org/10.1007/s11432-020-3308-4

• REVIEW •
Special Focus on Cyber-Physical Systems

Advanced virtual prototyping for cyber-physical systems using RISC-V: implementation, verification and challenges

Vladimir HERDT[1,2*] & Rolf DRECHSLER[1,2]

[1]*Institute of Computer Science, University of Bremen, Bremen* 28359, *Germany;*
[2]*Cyber-Physical Systems, DFKI GmbH, Bremen* 28359, *Germany*

Received 10 October 2020/Revised 14 May 2021/Accepted 21 July 2021/Published online 23 December 2021

Abstract Virtual prototypes (VPs) are crucial in today's design flow. VPs are predominantly created in SystemC transaction-level modeling (TLM) and are leveraged for early software development and other system-level use cases. Recently, virtual prototyping has been introduced for the emerging RISC-V instruction set architecture (ISA) and become an important piece of the growing RISC-V ecosystem. In this paper, we present enhanced virtual prototyping solutions tailored for RISC-V. The foundation is an advanced open source RISC-V VP implemented in SystemC TLM and designed as a configurable and extensible platform. It scales from small bare-metal systems to large multi-core systems that run applications on top of the Linux operating system. Based on the RISC-V VP, this paper also discusses advanced VP-based verification approaches and open challenges. In combination, we provide for the first time an integrated and unified overview and perspective on advanced virtual prototyping for RISC-V.

Keywords virtual prototyping, RISC-V, SystemC TLM, verification

Citation Herdt V, Drechsler R. Advanced virtual prototyping for cyber-physical systems using RISC-V: implementation, verification and challenges. Sci China Inf Sci, 2022, 65(1): 110201, https://doi.org/10.1007/s11432-020-3308-4

1 Introduction

Modern Internet-of-Things (IoT) devices and cyber-physical systems (CPS) are highly interconnected devices that offer enormous innovations from the application perspective. However, from the design flow perspective the existing challenges are amplified. On the one hand the complexity of these systems is rising continuously to provide the necessary functionality and on the other hand several conflicting design requirements need to be dealt with simultaneously. Key requirements include a high processing efficiency in combination with real-time computing capabilities as well as extensive connectivity to provide smart functions. Furthermore, safety and security combined with a high reliability are indispensable. Yet, these devices have to be cheap and operate with limited resources in a very energy-efficient way. To meet these requirements, highly application specific solutions are required that are tailored for each specific system. In particular, optimizations and application specific extensions of the processor enable to bring huge benefits.

RISC-V [1, 2], a free and open-source instruction set architecture (ISA), has significant potential to become a game changer in this area. RISC-V is designed from the ground up to be a highly configurable and extensible ISA. It offers a base integer instruction set in combination with several optional standard instruction set extensions which can be selected on a very fine granular basis. Addition of custom instructions, for example to boost performance or energy efficiency, is a built-in feature in RISC-V that is commonly used. This extensive modularity and enormous flexibility, combined with the fact that RISC-V is license-free and royalty-free, make RISC-V almost predestined to build highly efficient application specific processors to meet the ambitious requirements of next generation CPS and IoT devices. RISC-V

* Corresponding author (email: vherdt@uni-bremen.de)

is backed by an already capacious ecosystem that nonetheless is growing rapidly to evolve further. On the hardware (HW) side, several processor implementations are available (including open-source, free and commercial). On the software (SW) side, the ecosystem offers compilers, operating systems and simulators among others. Recently, virtual prototypes (VPs) have been introduced into the RISC-V ecosystem to complement the existing simulators in dealing with advanced system-level use-cases (such as design space exploration).

Leveraging VPs early in the design flow is a major industry-proven approach [3,4]. A VP is essentially an abstract executable model that is designed from the ground up to represent the entire HW platform. In industrial practice, VPs are predominantly created in SystemC transaction level modeling (TLM) [5,6]. VPs offer a middle ground between high-speed functional simulators (like QEMU) and register-transfer level (RTL) simulations, by being more accurate and faster, respectively. They facilitate SW development by supporting analysis of complex HW/SW interactions. Advanced SystemC-based techniques that consider extra-functional aspects (e.g., [7–12]) are applied to support other use-cases such as power and timing analysis based on the VP. Beside modeling and simulation, verification is crucial to avoid bugs that can lead to functional failure or even open security vulnerabilities that might be exploited. Due to their ease of use and scalability, simulation-based methods still form the main backbone of the verification effort. However, it is difficult to achieve a high verification quality with comprehensive coverage results and therefore sophisticated verification techniques are required.

In this paper, we present and discuss enhanced virtual prototyping solutions tailored for RISC-V. In particular, we build upon our recent previous studies in the context of RISC-V virtual prototyping [13–24], which consider different modeling and verification aspects in isolation, and for the first time show an integrated and unified view on advanced virtual prototyping for RISC-V[1)]. The foundation is our RISC-V VP that is implemented in SystemC TLM and designed as a configurable and extensible platform (Section 4). It scales from small bare-metal systems to large multi-core systems that run applications on top of the Linux operating system. Our RISC-V VP is fully open source[2)] (MIT license) to expand the ecosystem of RISC-V and stimulate further development and research. Our solution is the only freely available SystemC-based VP that is capable of booting Linux, to the best of our knowledge. Using our RISC-V VP as foundation, in this paper we discuss advanced verification approaches tailored for RISC-V that cover the major stages of a VP-based design flow. This includes verification of the VP (Section 5), the embedded SW (Section 6) and a VP-based cross-level methodology for RTL verification (Section 7). We also review related work (Section 2), provide a discussion on a modern VP-based design flow with a focus on the CPS/IoT domain (Section 8) and sketch ideas for future work as well as open challenges (Section 9). In combination, this paper for the first time shows an integrated and unified view on advanced virtual prototyping for RISC-V.

2 Related work

In this section, we review related work on RISC-V simulation and verification as well as verification of SystemC designs and embedded SW with a focus on formal techniques.

2.1 RISC-V simulation

Simulators are very important tools for RISC-V and hence the RISC-V ecosystem already has several different simulators that have been designed with different use-cases in mind to complement each other.

One important part is high-speed instruction set simulators (ISS) such as QEMU[3)] (that meanwhile offers comprehensive RISC-V support), the official reference simulator SPIKE[4)] or RV8[5)]. Their primary use-case is high-speed simulation and for this reason they employ aggressive optimization techniques such as dynamic binary translation (from RISC-V to native x86_64). However, this makes integration of accurate models for extra-functional information much more challenging.

1) Our most recent RISC-V related approaches. http://www.systemc-verification.org/risc-v.

2) The GitHub link of our RISC-V VP as well as most recent RISC-V VP updates and related information. http://www.systemc-verification.org/riscv-vp.

3) RISCV-QEMU. https://github.com/riscv/riscv-qemu.

4) Spike RISC-V ISA simulator. https://github.com/riscv/riscv-isa-sim.

5) RV8. https://rv8.io.

Another direction is full-system simulators which include gem5 [25] and Renode[6]. The simulator gem5 is designed for architectural explorations. Therefore, gem5 provides detailed models, that can also be extended with extra-functional properties, for the memory and processor. Renode goes beyond the single system level and enables simulation of multiple embedded systems arranged as multi-node networks. However, neither gem5 nor Renode employ the standardized SystemC modeling style which precludes integration of advanced SystemC-based methods.

Yet another direction is approaches that aim to provide executable formalizations of the RISC-V ISA. FORVIS[7] and GRIFT[8] are implemented in Haskell. Beside being executable, they can serve as foundation for formal analysis techniques. SAIL-RISCV[9] is a sail-based implementation. Sail is a special language designed for describing ISAs. It offers support to generate simulation back-ends and definitions for theorem-prover.

A few approaches have been designed to work in combination with SystemC. ETISS [26][10] is a configurable ISS that provides RISC-V support, leverages DBT to achieve a high performance and is implemented in C++. ETISS can be used standalone or integrated with a SystemC-based simulation. It complements our approach, as we conceptually could use ETISS as replacement ISS in our VP. A similar direction to ETISS is pursued by DBT-RISE, which is a generic framework to implement efficient DBT-based ISSs and offers RISC-V support as well[11]. The ISS is designed to be embedded in a SystemC-based VP platform[12], which can be considered comparable to our effort. Another simulator implemented in SystemC TLM is RISC-V-TLM[13]. However, currently its support for the RISC-V ISA is very limited.

Finally, a set of commercial VP tools such as Mentor Vista or Synopsys Virtualizer is available, and may offer extensive RISC-V support, though their implementation is proprietary.

Our RISC-V VP is fully open source to expand the ecosystem of RISC-V and thereby provide a strong foundation for advanced system-level use-cases based on SystemC TLM-2.0, which is an industrial proven modeling standard (IEEE-1666). Also, our solution is the only freely available SystemC-based VP that is capable of booting Linux, to the best of our knowledge.

2.2 RISC-V verification

Recently, a handful of verification approaches tailored for RISC-V have emerged. The most basic approaches are the official RISC-V unit[14] and compliance[15] test-suites. They are easy to use and recent reports indicate that a very high quality has been reached regarding the compliance test-suite[16]. However, using a predefined test-suite, the comprehensiveness of testing is still limited.

Therefore, automated test generation approaches have been designed for RISC-V. The Scala-based Torture Test generator[17] integrates predefined randomized test-sequences to generate tests. A combination of SystemVerilog with universal verification methodology (UVM) is leveraged by Google's RISCV-DV[18] to generate instruction streams which are specified using constrained-random descriptions. A commercial RTL simulator that provides support for SystemVerilog and UVM is required. In addition, in our previous studies we developed advanced test generation techniques for RISC-V that leverage fuzzing and constraint-based specifications [19–22]. Our techniques have been very effective in finding new bugs in RISC-V simulators and an industrial pipelined RISC-V core. In contrast to the aforementioned RISC-V simulation-based approaches, our techniques leverage advanced complementary test-generation techniques which enable them to find intricate bugs. We discussed these methods in Section 7.

A few formal verification approaches are available as well for RISC-V in addition to the test-generation approaches. Noteworthy approaches that are based on model checking techniques are the OneSpin 360

6) Renode. https://renode.io/.
7) Forvis: a formal RISC-V ISA specification. https://github.com/rsnikhil/RISCV-ISA-Spec.
8) GRIFT—galois RISC-V ISA formal tools. https://github.com/GaloisInc/grift.
9) Riscv sail model. https://github.com/rems-project/sail-riscv.
10) ETISS (extendable translating instruction set simulator). https://github.com/tum-ei-eda/etiss.
11) DBT-RISE-RISCV. https://github.com/Minres/DBT-RISE-RISCV.
12) RISCV-VP. https://github.com/Minres/HIFIVE1-VP.
13) Another RISC-V ISA simulator. https://github.com/mariusmm/RISC-V-TLM.
14) RISC-V ISA tests. https://github.com/riscv/riscv-tests.
15) RISC-V compliance task group. https://github.com/riscv/riscv-compliance.
16) Imperas delivers highest quality RISC-V rv32i compliance test suites to implementers and adopters of RISC-V. https://riscv.org/2019/11/imperas-delivers-highest-quality-risc-v-rv32i-compliance-test-suites-to-implementers-and-adopters-of-risc-v/. 2019.
17) RISC-V torture test generator. https://github.com/ucb-bar/riscv-torture.
18) RISCV-DV. https://github.com/google/riscv-dv.

DV RISC-V Verification App[19] and riscv-formal[20]. Recently, research approaches targeting RISC-V pipelined microarchitectures emerged that rely on formal methods and consider functional properties [27] as well as transient execution attacks [28]. However, formal verification methods should be complemented by simulation-based approaches due to their complexity and potential scalability issues.

Looking beyond RISC-V, several approaches for test-program generation have been proposed for the purpose of processor verification. For example, they integrate model-based techniques with constraint solving [29–32] or leverage coverage-guided test generation based on machine learning [33, 34] or fuzzing [35].

2.3 SystemC verification

Simulation is still prevalent for SystemC verification due to its ease of use and scalability [5][21]. Strong enhancements have been proposed to the basic simulation method by adding support for validation of TLM assertions, e.g., [36–39]. In the next step, approaches that utilize partial order reduction (POR) [40, 41] have been proposed [42, 43] to improve the simulation coverage further. They enable to efficiently explore all different process execution orders by pruning redundant orders. But still, it is necessary to provide representative inputs. To address this issue, formal verification approaches have been developed for SystemC TLM. The first approaches, e.g., [44–47], had problems in modeling the simulation semantics of SystemC accurately or suffered from severely limited scalability [48].

In the following, we briefly summarize the more recent formal verification approaches for SystemC. KRATOS [49] combines symbolic abstraction refinement with an explicit scheduler for efficient handling of cyclic state spaces. SCIVER [50] translates SystemC designs into sequential C models and applies state-of-the-art C model checker. SDSS [51] formalizes and encodes the complete state space of the SystemC design into an SMT formula. In [52], the approach was boosted with POR support. STATE [53] translates SystemC designs to timed automata and applies the UPPAAL model checker. In [54], a concolic testing approach tailored for bug hunting has been presented. Please refer to the survey [55] for more information on SystemC verification approaches.

2.4 Embedded SW verification

Most SW verification methods focus on non-embedded SW. This kind of SW has none or only limited interaction with the HW. KLEE [56] and SAGE [57] are representative candidates that made symbolic execution techniques applicable to large SW. Subsequent work focused on improving scalability further and mostly targeted binary level SW to obtain accurate results, e.g., S2E [58], Mayhem [59] or Angr [60]. Another active research area is verification of multi-threaded programs, e.g., [61], which has applications to interrupt verification as well [62].

To deal with embedded SW, specialized HW/SW symbolic/concolic co-validation approaches have been devised. Their primary difference is on the method for integration of the underlying HW. Virtual peripheral models that are extracted manually from QEMU are leveraged by [63, 64]. In [65], instead HW Verilog models are used. In [66], a symbolic execution environment that is powered by KLEE and tailored for MSP430 microcontroller is provided. Physical devices are integrated by [67] to enable hybrid binary concolic testing. Our concolic testing approach specifically targets RISC-V embedded binaries (Section 6).

Beside formal methods, different random/fuzz testing approaches targeting embedded systems have been proposed as well, e.g., [68–71]. Though they neither target RISC-V nor SystemC-based VPs.

3 Preliminaries

This section presents relevant background information on the RISC-V ISA (Subsection 3.1) and SystemC TLM (Subsection 3.2).

19) OneSpin 360 DV RISC-V verification App. https://www.onespin.com/solutions/risc-v.
20) RISC-V formal verification framework. https://github.com/SymbioticEDA/riscv-formal.
21) Accellera Systems Initiative. SystemC. http://www.systemc.org.

3.1 RISC-V

RISC-V is a very modular, configurable and extensible ISA. The foundation is the mandatory base integer (I) instruction set. It is available in 32, 64 and even 128 bit with corresponding register widths, denoted by RV32, RV64 and RV128, respectively. In addition, optional standard instruction set extensions such as multiply/divide (M), atomic operations (A), compressed instructions (C) and floating points with single (F) or double (D) precision are available. Designated instruction set encoding spaces are reserved for custom instruction set extensions. The standard instruction set combination is denoted by the single letter G = IMAFD. More information on the standard instruction sets is available in the RISC-V user-level ISA specification [1].

Another important part of the RISC-V ISA is the privileged architecture description [2]. It describes important concepts and instructions that are used for advanced operations such as trap/interrupt handling and operating system support. In particular, control and status registers (CSRs) play a central role here. CSRs are registers that serve a special purpose to interface between HW and SW. For example, mtvec is configured by the SW to store the trap handler address and mtval provides exception specific information to the SW in case of a trap. Beside the mandatory machine mode (M), the supervisor (S) and user (U), operation modes are defined to support different privilege levels.

3.2 SystemC and TLM

SystemC is a standardized C++ based modeling language that together with TLM is an industrial proven combination to build VPs [4]. At the heart of SystemC is the event-driven simulation kernel [5] that orchestrates the execution of processes. Processes are the foundation to describe behavior in SystemC. They are triggered by events and scheduled non-preemptively by the kernel, i.e., a process has to give back the control explicitly by calling wait. Modules and ports are leveraged to describe the structure of a SystemC design.

TLM transaction is the main communication method between different SystemC modules [72]. Compared with an RTL simulation, drastic speed-ups in simulation performance, i.e., up to a factor of 1000, are possible by this abstraction. A transaction object contains the necessary data to implement different memory access operations. This includes the access type (e.g., read or write) and address as well as the data (payload). Based on the address, the transaction is routed from the initiator to the target on a bus system. A delay can be passed alongside the transaction to obtain a more accurate estimation of the passed simulation time. All this is specified in the SystemC TLM-2.0 standard to ensure compatibility and interoperability of different components.

4 RISC-V VP implementation

In this section, we briefly summarize the main features of our open-source RISC-V VP. For a more in-depth discussion on the implementation aspects, please refer to our implementation paper [13].

4.1 Architecture overview

Figure 1 (left side) gives an overview on the main architecture of our RISC-V VP. It is designed as configurable and extensible platform with a TLM 2.0 bus at the center. A 32 and 64 bit ISS with all RISC-V standard instruction set extensions is provided. Multiple ISSs can be instantiated to build a multi-core platform. Each ISS is attached to the bus via a memory interface. It is responsible to translate load and store instructions into TLM transactions. A memory management unit (MMU) is provided with the ISS to translate virtual to physical addresses. Interrupts are processed by RISC-V specific interrupt controllers (CLINT and PLIC in Figure 1). CLINT handles timer and SW interrupts, while PLIC deals with interrupts coming from other devices (it prioritizes and routes them to the ISS). An essential set of additional peripherals (e.g., UART, sensor) is provided as well. They are accessed through memory mapped I/O and can also be bus masters (like a DMA controller). Finally, a special system call handler is provided to (optionally) directly execute C/C++ library system calls. Our VP supports several operating systems including Zephyr, FreeRTOS, RIOT and Linux. Executable RISC-V binaries (ELF files) are loaded by means of an ELF loader component which is responsible to load the memory image and setup the program counter of the ISS.

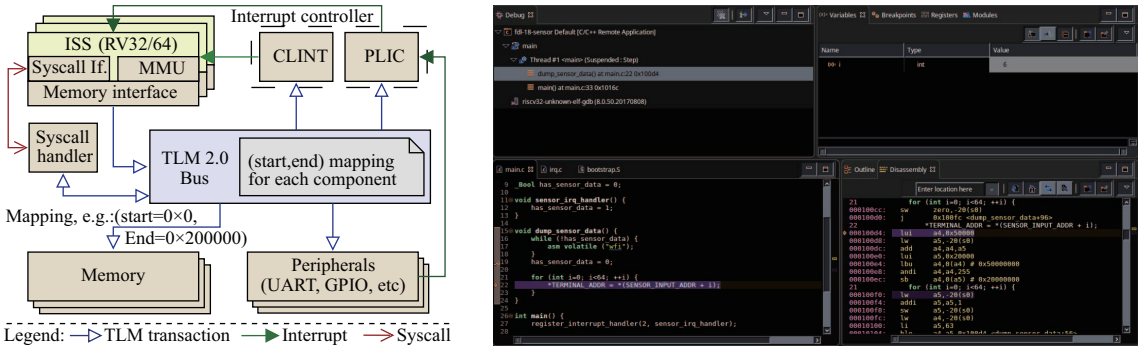

Figure 1 (Color online) (a) Overview on the RISC-V VP central architecture; (b) debugging RISC-V SW that is executed on the RISC-V VP via Eclipse by leveraging the GDB RSP interface.

4.2 Extension and configuration

Peripheral is attached to the central TLM 2.0 bus based on an address range mapping. This mapping is easily configurable. At the same time, additional peripherals can be easily integrated by attaching them to the bus system. This is very important to build different application specific platforms. A major use-case for RISC-V is integration of custom instruction set extensions to boost the overall efficiency. For this reason, the ISS can be extended accordingly with custom decode and execute functions for the instruction set.

4.3 HiFive1 configuration

One example that demonstrates the extensibility of our VP is the HiFive1 configuration. It resembles the HiFive1 board from SiFive and in particular is binary compatible with this board, i.e., the same RISC-V binary can be executed on the VP and the real board without modifications. The HiFive1 board integrates the FE310-G000 SoC[22]. It has an RV32IMAC core, code and data memories as well as numerous peripherals. This includes CLINT and PLIC-based interrupt controllers as well as UARTs and GPIOs for environment interaction. UART output is redirected to the console and for the GPIOs we provide an interprocess communication interface to enable access to an external environment model. As an example we build a Qt-based framework that supports graphical input and output components (such as buttons and LEDs) that can be attached to the GPIO interface.

4.4 Performance optimization

Performance optimizations are very important to boost the simulation speed and thus facilitate early SW development and testing. Therefore, we integrated two common optimizations designed for SystemC-based simulations.

(1) Direct memory interface (DMI) is utilized to speed-up memory access operations by bypassing the bus system. This includes fetch as well as load and store operations.

(2) Temporal decoupling enables to postpone context switches to the SystemC simulation kernel by utilizing a local time quantum to run ahead of the global simulation time. This is particularly useful inside of the ISS.

4.5 Eclipse-based SW debugging

Strong SW debugging capabilities are among the key features of a VP as they are paramount to investigate intricate errors. In addition, debugging at the VP-level provides reproducible result due to the deterministic simulation environment, thus making it even more valuable. We implemented the GDB remote serial protocol (RSP) interface to provide comprehensive SW debugging support in our VP. Graphical interfaces that support RSP can be directly attached, this includes the Eclipse IDE (see Figure 1(b)). Features include stepping through the SW (on the binary and source code level), setting breakpoints (w/o

22) HiFive1. https://www.sifive.com/boards/hifive1.

conditions) and accessing variables (reading and writing). In addition, support for debugging multi-core SW applications is provided as well.

4.6 Timing model

To support evaluation of extra-functional properties, corresponding models are integrated with the VP-based simulation. In the ISS, we provide an instruction-based timing model that allows to annotate fixed (though configurable) delays for each instruction type. This is a very generic timing model that has negligible performance impact and can be leveraged to obtain first approximate estimation on the SW execution times. In addition, more complex timing models can be integrated through specific interface mechanisms, which we demonstrated by providing a timing model designed to match the RISC-V 32 bit E31 core from SiFive [73]. These interface mechanisms enable to cover pipeline, branch prediction and caching effects which are important to obtain more accurate execution time estimations. However, this more accurate estimation comes at two costs: first, the simulation performance is noticeably affected by modeling these additional microarchitectural effects and second, such a timing model is no longer generic but needs to be provided specifically for the microarchitecture at hand.

As a complementary technique we investigated runtime adaptive simulations that enable to switch the accuracy setting at runtime through user defined configurations. This enables for example to skip the boot process of an operating system by using a fast simulation technique and focus on the actual application using a more accurate technique. To boost the simulation performance we investigated just-in-time compilation techniques in this runtime adaptive setting [74].

In addition to the ISS, the bus system and peripherals play a very important role as well to obtain accurate timing estimation results, for example for the purpose of an early design space exploration. Currently, we employ the so called loosely-timed modeling style to integrate our peripherals on top of a generic TLM-2.0 bus system. For a timing estimation, TLM transactions can be annotated with optional delays to support more accurate timings of SW that interacts with peripherals. By using a generic bus system, the VP can also serve as a foundation to integrate the more accurate approximately-timed modeling style. This enables to obtain more accurate timing results at the cost of a lower simulation performance. A good compromise between simulation performance and timing estimation accuracy may be obtained by selectively refining specific communication protocols with certain peripherals. Moreover, for different architectures, such as ARM, hybrid solutions that integrate FPGA-based emulation with VP-based models, to obtain fast and accurate simulations, have been leveraged[23]. Such solutions are also applicable for RISC-V in general.

5 Verifying the VP

Extensive verification of the VP is crucial, because the VP serves as reference model for subsequent development steps and as platform for SW development. First, we describe the basic RISC-V test infrastructure and how we tested our VP (Subsection 5.1), and then we present a summary of our formal verification techniques tailored for SystemC TLM designs (Subsection 5.2) that we plan to utilize to formally verify our VP in the next steps.

5.1 Testing

To test our VP we leveraged existing test-suites, in particular the RISC-V unit tests and the modern compliance tests (we provide more details on compliance testing in Section 7). These test-suites essentially cover important basic functionality as well as several corner-case scenarios. However, their thoroughness is necessarily limited (because the test-suite has a fixed size). Therefore, we also utilized RISC-V test generation approaches, in particular:

(1) the Scala-based Torture test generator that randomizes predefined instruction templates into new test sequences; and

(2) a fuzzing-based test generation engine that creates instruction sequences in a coverage-driven way [22].

23) ARM. The power of virtual prototyping: from soc design to software development. https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/v%irtual-prototyping-soc-design.pdf. 2019.

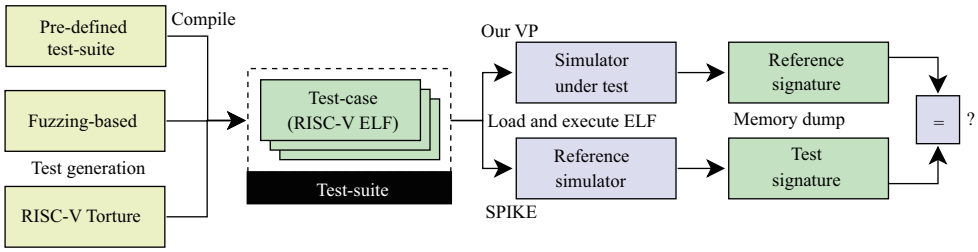

Figure 2 (Color online) Overview on the RISC-V Torture and CGF approach for VP testing.

While Torture focuses on positive testing, our fuzzing-based approach complements it with negative testing capabilities. Thus, in combination, a very solid verification of RISC-V based systems is achieved.

The test generation approaches (and the compliance tests) follow a signature-based test infrastructure. Figure 2 shows an overview. A test-suite is constructed that ultimately is compiled into executable RISC-V binaries (ELF files) that represent the test-cases. Each binary is loaded and executed on the simulator under test (our VP in this case) and a reference simulator (we used the official RISC-V reference simulator SPIKE), which both produce signatures. A signature is a memory dump that provides the test result. A difference in the signatures indicates a bug in the simulator under test.

In addition to test generation approaches, we developed and executed several complex SW applications on our VP. They are based on the Zephyr and FreeRTOS as well as Linux operating systems. Beside core components such as threads, interrupts, timers, message queues and semaphores, also several libraries such as FAT, UDP, SLIP and TinyCrypt are leveraged. Moreover, these applications utilize the full-platform (including peripherals like a UART or interrupt controller) and not just the CPU core and memory. In this testing process, we observed that the applications behaved as expected.

5.2 Formal verification of SystemC designs

Testing is widely adopted because it is ease of use and scalability but at the same time inherently suffers from incompleteness of the testing process. Therefore, to prove correctness, formal verification techniques are indispensable. However, formal verification is very challenging, in particular for SystemC-based designs [24, 75]. We identified three main challenges that need to be addressed.

(1) It has to deal with very large state spaces. The reason is that it has to consider all possible inputs and different process execution orders of the SystemC design in order to be complete.

(2) A typical SystemC design contains unbounded process loops that are triggered by events. Thus, a cycle detection mechanism is required to complete the verification process.

(3) Since SystemC is a C++ library, the verification engine requires to handle the full complexity of C++ in order to obtain a formal model for verification.

The first two challenges relate to the verifier back-end while the third challenge is a front-end issue. Therefore, to make the challenges more manageable, we introduced an intermediate verification language (IVL) (see [76]) to separate between both issues and focus on the back-end challenges henceforth (the IVL has been further extended in [77] to support additional language features tailored for TLM peripheral models). In short, the IVL is an open and compact language designed as formal intermediate representation for SystemC.

5.2.1 *Stateful symbolic simulation*

Based on the IVL, we developed a stateful symbolic simulation approach that combines several techniques to enhance the scalability in the back-end and thus address the remaining two challenges [23, 78]. The foundation are three techniques that are integrated under the simulation semantics of SystemC.

(1) Symbolic execution (SymEx) to reason about a large number of different inputs and exploration paths very efficiently.

(2) POR to prune redundant scheduling sequences of SystemC processes.

(3) State subsumption reduction (SSR) to efficiently detect exploration cycles in symbolic explorations.

While these techniques drastically boost the verification efficiency and provide a strong framework for SystemC verification, complex symbolic reasoning is still subject to scalability issues and therefore further optimizations remain to be highly important.

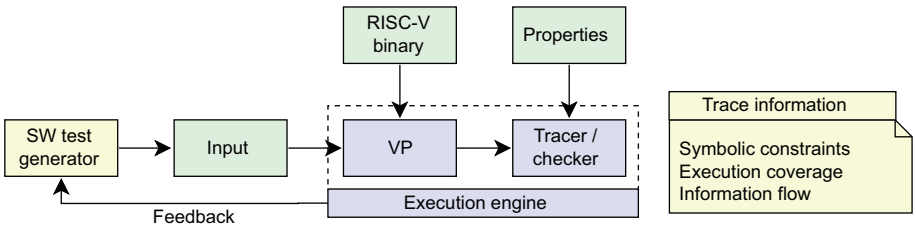

Figure 3 (Color online) VP-based verification of RISC-V embedded SW.

5.2.2 *Optimization and application*

One such optimization is compiled symbolic simulation (CSS) [79]. The idea is to leverage native execution to boost the exploration performance. To achieve this, the engine for symbolic execution is integrated with the process scheduler (that employs a POR-based optimization) into the SystemC design under test. Then, all components are compiled into a single binary, which enables native execution of the scheduled process orders. Performance can be further boosted by leveraging a parallelized exploration algorithm on top of CSS [80]. As a case-study, we reported verification results for a VP-based interrupt controller [77, 80]. Please refer to our related work in Subsection 2.3 for an overview on other formal approaches for SystemC verification.

In the next steps, we plan to leverage (our) formal verification methods for SystemC to verify components of our RISC-V VP and ultimately the whole VP.

6 Verification of embedded software binaries

Figure 3 shows a conceptual overview on (our) RISC-V VP-based verification methods for embedded SW binaries. Essentially, two components are involved: (1) the SW test generator (left side of Figure 3), and (2) the execution engine (center bottom of Figure 3). The test generator continuously provides new inputs to the execution engine. At the heart of the execution engine is the VP which is leveraged for executing the SW binary with each input. A VP-based execution enables to obtain very accurate verification results, because complex HW/SW interactions such as interrupts and peripheral interactions are modeled accurately and the SW is processed at the binary level. A tracer and checker component is attached to the VP to collect and process execution information and pass that feedback back to the test generator to guide the test generation process. Specified properties are checked alongside the VP-based execution.

Following this conceptual flow, we implemented three different approaches that leverage concolic testing (CT) [14, 15], coverage-guided fuzzing (CGF) [16] and dynamic information flow tracking (DIFT) [17]. They trace symbolic constraints, execution coverage and information flow during the VP-based execution, respectively (right side of Figure 3). CT and CGF enable automated test generation specifically tailored to increase coverage for functional verification. CT leverages formal methods based on solving symbolic constraints while CGF employs scalable fuzzing techniques. Properties are specified in form of SW assertions. DIFT focuses on checking of security related properties, such as integrity and confidentiality, alongside the execution and thus complements test generation techniques.

All these techniques have been shown to be very effective in the SW domain. However, using them on embedded SW binaries is challenging, because it requires to deal with architecture specific details and extensive interaction with HW peripherals. These challenges can be addressed by leveraging VPs. In the following, we briefly summarize the main idea of these three approaches in our VP-based context.

6.1 Concolic testing

CT works by tracking symbolic constraints on top of the concrete execution and solving these constraints to generate new inputs that represent new tests. Each test explores a different path through the SW program thus ultimately maximizing the SW path coverage. Symbolic constraints are also leveraged to check assertions and other potential error conditions alongside the execution.

Our VP-based approach [14, 15] enables concolic testing for RISC-V binaries that extensively interact with peripherals. On the technical side, the RISC-V ISS and memory are instrumented to track symbolic constraints alongside the native execution and a specialized interface is provided to integrate SW models

of peripheral. The interface is tailored for the requirements of SystemC-based peripherals. Our approach has been very effective in finding buffer overflows in the TCP/IP stack of FreeRTOS and a bug in the RISC-V specific memcpy function of the newlib C library.

6.2 Coverage-guided fuzzing

CT is a powerful verification technique but may be susceptible to scalability issues due to path explosion and complex symbolic constraints. Therefore, it is important to provide scalable simulation-based methods to complement CT. CGF is such a technique. Based on the principles of classical fuzzing [81], modern CGF continuously mutates randomly created data to produce new inputs and is guided by code coverage. Notable CGF representatives in the SW domain are the LLVM-based libFuzzer[24] and AFL[25].

We combined SystemC-based VPs with CGF for verification of embedded RISC-V SW binaries [16]. The fuzzing process is guided by two coverage metrics to be more efficient: (1) from the embedded SW, and (2) from the SystemC-based peripherals of the VP. This allows to bridge the gap between SW and HW in the fuzzing engine. The VP tracks both coverage information while executing test-cases. Our approach has been very effective in analyzing real-world RISC-V embedded SW binaries (based on bare-metal systems and using Zephyr).

6.3 Dynamic information flow tracking

Protecting SW against security related exploits is more important than ever in today's highly interconnected devices. A powerful technique to enable such protection is DIFT [82, 83]. DIFT tracks the flow of information alongside the SW execution from the inputs to the outputs of the system [84]. It allows to check that secret data is not leaked (confidentiality) and untrusted input does not influence sensitive data (integrity).

Our approach combines DIFT with VPs to enable early and accurate information flow analysis of embedded RISC-V binaries [17]. A major benefit of our approach is the virtually non-intrusive integration of the DIFT engine with the VP-based execution. In particular, we leverage C++ templates and operator overloading to achieve such a transparent integration. Our approach has been very effective in revealing security related exploits based on a car engine immobilizer case-study and in detecting code injections based on a standard benchmark set.

7 Cross-level compliance testing and verification

Compliance testing and verification are very important problems for RISC-V. Figure 4 shows an overview on our cross-level VP/RTL compliance testing (top) and verification (bottom) approaches. The idea is that the ISS of the VP serves as reference model for the RTL implementation, which is a common setting in a VP-based design flow. We will discuss our approaches in Subsections 7.2 and 7.3, respectively. We review related work in Subsection 2.2. In the following, we first start with a brief motivation on the role and importance of compliance testing for RISC-V and delimit it from design verification (Subsection 7.1).

7.1 Motivation on compliance testing for RISC-V

As stated in the introduction, RISC-V is a highly modular ISA that offers extensive configuration options and can be extended with custom instruction sets to build very application specific processors. However, as a direct implication it becomes very challenging to ensure that the RISC-V ecosystem as a whole is compatible with all the different processors. Too much customization can lead to the point, that SW incompatibilities between RISC-V implementations are introduced which in turn cause fragmentation of the ecosystem. This crucial problem has been recognized by the RISC-V foundation and therefore the compliance task group has been formed to develop efficient methods for compliance testing[26]. In contrast to design verification, which attempts to find bugs in the processor and ultimately prove correctness of the full functional behavior, compliance testing focuses on checking the relevant parts for the HW/SW interface to ensure compatibility of the processor with the RISC-V SW ecosystem. Thus, compliance testing for example checks if some registers are missing or have incorrect widths, it also checks available

24) libFuzzer—a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
25) American fuzzy lop. http://lcamtuf.coredump.cx/afl/.
26) The challenge of RISC-V compliance. https://semiengineering.com/toward-risc-v-compliance/. 2019.

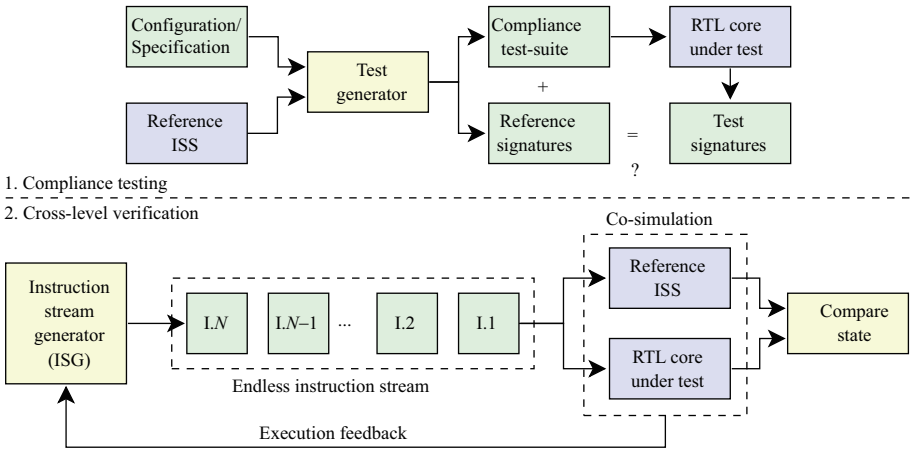

Figure 4 (Color online) Cross-level co-simulation for verification purposes.

modes and access types, as well as basic sanity checks on the instructions (including if they are absent). This is very important to ensure that highly customized RISC-V processors from different vendors can benefit from the extensive RISC-V ecosystem. The official approach for compliance testing that is pursued by the compliance task group is to build a designated compliance test-suite for each RISC-V standard instruction set. While recent reports indicate that a high quality has been reached by the compliance test-suite for the base integer instruction set, the compliance testing problem for RISC-V is still far from being solved. Efficient compliance test generation methods are required to obtain comprehensive results. Finally, compliance testing complements design verification but does not replace it. Thorough verification has still to be performed in the subsequent design flow steps.

7.2 Compliance testing

Ultimately we follow the official approach on compliance testing but generate alternative test-suites that enable more comprehensive compliance testing. Essentially, two steps are involved: (1) generating a compliance test-suite together with a set of reference signatures (i.e., selected result register and memory values), and (2) using the generated test-suite to check compliance of an RISC-V simulator or RTL core under test as shown in Figure 4 (top). Following this methodology, we developed three complementary approaches for test-case generation that target compliance testing from the positive and negative testing perspective:

(1) Coverage requirements for the final test-suite are specified in combination with instruction constraints in a domain specific language. All requirements are solved in combination with the constraints using an SMT solver to generate a compliance test-suite [21]. This offers a strong foundation for positive testing.

(2) Complements our first approach by utilizing mutation-based testing. In a first step, a set of mutation classes, tailored for the RISC-V ISA, is defined. In the second step, symbolic execution techniques are leveraged to generate a test-suite that kills all mutations in the ISS (that implements the RISC-V ISA) [85].

(3) Here the focus is on negative testing to further complement the positive testing approaches. Negative testing attempts to reveal bugs by focusing on illegal instructions and other inputs that can cause exceptions. The rational is to ensure that no additional behavior is accidentally added in an RISC-V implementation. We leverage fuzzing-based techniques that are guided by different coverage metrics [20].

In combination, our approaches offer a strong compliance testing framework and significantly enhance the existing official test-suites. Based on our approaches, we found new bugs in several RISC-V simulators.

7.3 Cross-level verification

Thorough verification is crucial to detect bugs and prove their absence. Formal verification techniques at processor level still suffer from a high complexity and potentially limited scalability. For this reason simulation-based methods that leverage randomized testing techniques still form the backbone of the verification effort.

In this context, we proposed an efficient approach [19] for RISC-V processor verification in an RTL/ISS cross-level setting. At the heart of our approach is an instruction stream generator (ISG) that generates an endless and dynamically evolving stream of instructions. This instruction stream is fed in a cross-level co-simulation setup to the reference ISS and RTL core under test, as shown in Figure 4 (bottom). After each instruction the resulting architectural state is compared between both models. Our approach offers three major benefits compared to traditional test generation approaches.

(1) No control flow related restrictions is placed on the instruction stream. In particular, infinite loops (or trap loops) are no problem, because the instruction stream evolves dynamically at runtime (hence a different instruction can be returned for the same program counter).

(2) Restarts, that reset the execution state before executing a new test, are not necessary. By using a single endless instruction stream, very long test sequences, which can be very important to find intricate bugs, are possible.

(3) A very high performance is achieved by placing the ISS and RTL core in a tight co-simulation within a single process. Furthermore, instructions are directly injected into the processor fetch interface making the processing step very efficient. We observed more the 200 million processed instruction per hour on a standard laptop.

Our approach has been very effective in finding several serious bugs in an industrial pipelined 32 bit RISC-V core. Most bugs were related to the RISC-V privileged architecture, particularly CSRs. Testing CSRs is a challenging task which has been mostly neglected by existing RISC-V verification frameworks. In Subsection 2.2, we discuss related work on RISC-V verification in more detail.

8 Application in the IoT/CPS domain

Modern IoT and CPS devices have certain properties that make their design flow very challenging: They integrate complex hardware and software components; they are often resource constrained devices that need to operate efficiently under tight energy consumption requirements; they operate and extensively interact with the physical environment; they provide intelligent functions and a high degree of connectivity; they have stringent requirements on robustness as well as safety and security.

Therefore, it is very important to provide methods for design space exploration, parallelize the HW and SW development and start early with integration and verification efforts. For this reason, VPs play an essential role in the design flow for such embedded systems by providing a platform for early SW development and serving as an executable reference model for the subsequent design flow steps.

A VP provides a single unified view which covers the SW, HW and environment in a single execution environment. By being an SW model written in the C++ based SystemC language, a VP provides very extensive introspection capabilities that cover the SW, HW and environment levels. A very important feature in particular for CPS is the ability to configure and access the execution state on a very fine granular basis during the execution. This strongly facilitates testing and debugging of CPS. On one hand the execution can be stopped, including the physics simulation of the environment (which is not possible in a physical setting), on the other hand very specific configurations can be setup and tested much more easily and quickly (which can be very cumbersome and expensive to setup in a physical setting). Moreover, to cover robustness aspects, which are crucial for IoT and CPS, error effect simulations can be conducted at the VP level in a straightforward way. Essentially, it works by injecting errors such as single bit flips in registers and observing their effects on the simulation outcome. Such an error effect simulation is very fast, as it is performed on a C++ model, but also very accurate, as the VP provides the complete HW interface in a way it is relevant to the SW execution and environment interaction. By using snapshots, testing and debugging can be further improved on the VP level. Moreover, components at the VP level can be replaced by refined RTL implementations or even connected with physical devices to enable hybrid simulations which can be very important for CPS/IoT devices that strongly interact with the physical environment.

Based on our open source RISC-V VP, such a development flow for CPS/IoT devices is enabled for the modern RISC-V architecture. Moreover, our VP can also serve as a platform for further research and education in these areas. As mentioned in Section 4 our RISC-V VP in particular also covers the SW, HW, and environment levels. On the SW side, we do support complex stacks with libraries and operating systems such as FreeRTOS, RIOT, Zephyr and Linux. On the HW side, the VP is a configurable and extensible platform that scales from single- to multi-core devices. We provide an example configuration

for the HiFive1 board from SiFive including a virtual environment for an example scenario that enables to provide inputs and visualize outputs. An Eclipse-based debugging environment enables fine granular introspection of the execution state and control of the simulation.

9 Challenges and future work

Our RISC-V based VP is implemented in SystemC TLM-2.0 and already provides a significant set of features, which makes our VP a strong foundation for several application areas. Among them are early SW development and the ability to analyze complex HW/SW interactions for RISC-V based systems. In addition, our verification approaches tailored for RISC-V to verify the VP, the embedded SW and the RTL in a VP cross-level setting, show very promising results and integrate seamlessly with the VP-based design flow. Nonetheless, our VP and verification approaches can still be improved and extended. In the following, we discuss our plans for future work and sketch the main challenges to further enhance virtual prototyping solutions for RISC-V.

• Verification. Formal verification methods are crucial to obtain correctness proofs but they are still highly susceptible to scalability problems. Thus, boosting their scalability is a very important research direction. Therefore, domain specific optimizations and efficient combination of different verification techniques such as symbolic execution with state space abstraction and reduction techniques are investigated (Subsection 2.3). The major goal is to avoid state space explosion as much as possible and thus pave the way for applying formal verification methods to the whole VP. We believe that our proposed stateful symbolic simulation for SystemC (Subsection 5.2) provides a very solid foundation to seek this goal. Another way to improve scalability is to integrate existing simulation-based techniques with formal methods (e.g., coverage-guided fuzzing with symbolic simulation) to create a single unified verification approach that combines the benefits of both worlds. Such a unified combination enables to cover the state space very efficiently by providing a deep and broad cover. The idea is to switch seamlessly and intelligently at runtime between formal and simulation-based verification techniques on demand to achieve the best possible utilization of available resources. Advanced exploration strategies can be devised to speed-up bug hunting further and increase coverage more efficiently. An important point in this direction is also the development of stronger coverage metrics, e.g., [86], that combine code coverage with functional coverage and take complex features such as interrupts or threads into account. The challenges in formal SW verification are conceptually similar. Therefore, these solutions apply as well in the SW context to improve handling of large state spaces. Existing SW verification methods mainly leverage symbolic execution techniques (Subsection 2.4).

• Fast and accurate simulation. VPs should provide a high simulation performance (to deal with complex SW) and at the same time yield accurate timing results (to do performance evaluations and optimizations), which are two conflicting requirements in general. Different research directions are available to tackle this problem. Source level timing simulations (SLTS) attempt to instrument precise timing information into the SW, which is then compiled and natively executed on the host system (e.g., [87,88]). An alternative is to leverage DBT-based techniques to speed-up the VP-based simulation via native execution (e.g., [89, 90]). However, sophisticated analysis techniques are required to instrument precise timing information and accurate modeling of complex HW/SW interactions such as interrupts becomes very challenging. Another research direction is to leverage runtime adaptive simulations that can dynamically (and ultimately intelligently) switch between fast and accurate execution modes at runtime, e.g., [91, 92]. A use-case would be to perform a fast boot of an operating system and then a precise analysis of an SW driver. Beside performance evaluations, these methods are also applicable for other non-functional properties such as power consumption.

• Cross-level methodology. The VP serves as a reference platform for subsequent development steps in the design flow and in addition is an executable high-level model that offers fast simulation performance combined with strong debugging and configuration capabilities. An important research direction is to investigate a cross-level methodology that brings together the VP level and RTL. This allows to re-use VP-based information at RTL and enhance VP-based methods with RTL information. For example, in [93] an RTL to TLM correspondence analysis is presented that can be used to speed-up a VP-based error effect simulation. Another approach switches between RTL and VP simulation model at runtime to achieve that [94]. Starting from the VP, the approach in [95] derives an RTL property set based on TLM properties as starting point for RTL model checking. It is a step towards building a fully automated

and scalable cross-level methodology that enables to utilize VP verification results and provide them for the RTL. Our presented cross-level compliance testing and verification approaches (Section 7) are a step in this direction as well. They can be further improved by leveraging stronger coverage metrics for the test-suite generation and application of formal methods for the VP/RTL equivalence check. Finally, high-level synthesis (HLS) techniques are an important building block to streamline the design flow for embedded systems. However, their applicability is still limited.

• Security. Security is a crucial aspect that will become even more important in the highly connected next generation embedded systems. Therefore, the VP-based design flow should be augmented to consider security related aspects and thus enable early and accurate evaluation of security policies that reason about data integrity and confidentiality. Our VP-based DIFT is a very promising combination to achieve this goal. However, beside DIFT-based monitoring security policies it is necessary to devise efficient and scalable verification techniques and coverage metrics that are tailored for security policies and complement existing functional verification. Another interesting direction to support DIFT would be to automatically learn the data flow relations, e.g., based on neural networks [96], within a SystemC-based peripheral to avoid a manual DIFT integration into every peripheral.

• Mixed-signal. Beside security, another important aspect that should be considered early in the VP-based design flow is mixed-signal support in particular from the verification perspective (e.g., [97, 98]). The goal here would be to provide a unified and comprehensive verification environment that can reason about digital and analog components in combination. This complementary research direction would further broaden the verification scope to support highly heterogeneous systems.

• RISC-V. The RISC-V ISA adds a set of additional challenges that are inherently rooted in the design decisions made for RISC-V. Being an open and royalty-free ISA that has been specifically designed to be highly configurable and extensible, RISC-V is particularly suited to build highly efficient application specific solutions that include only the necessary features combined with custom domain specific extensions. This unrestricted design freedom has implications on the RISC-V ecosystem and adds unique challenges to verification solutions (for example, compliance testing is extremely important and challenging). Considering the VP-based design flow, in particular the integration of custom instruction extensions will be very important and should be supported at all levels ranging from specification to generation of simulation models (functional and non-functional) as well as verification.

Finally, in the best case, ultimately all these aspects should be integrated into a single unified VP-based framework that cross-integrates and connects all approaches. For example, the integration of AMS models should work together with the DIFT-based security approaches and be compatible with the HLS as well as be supported by the formal verification technique. This amplifies the existing challenges even further. A compositional approach that efficiently combines all separate building blocks might be a viable solution to tackle this problem.

10 Conclusion

In this paper, we presented advanced virtual prototyping techniques tailored for RISC-V in a unified overview. The foundation is our open-source RISC-V VP that is implemented in SystemC TLM and designed as a configurable and extensible platform. It scales from small bare-metal systems to large multi-core systems that run applications on top of the Linux operating system. Based on our VP, we discussed and reviewed advanced verification techniques that are designed to verify the VP itself, the embedded RISC-V SW and the RTL implementation in a VP-based cross-level setting. Finally, we sketched promising directions for future work and open challenges in the context of virtual prototyping for RISC-V.

Acknowledgements This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys (Grant No. 01IW19001), within the project Scale4Edge (Grant No. 16ME0127), and within the project SATiSFy (Grant No. 16KIS0821K), and the German Research Foundation (DFG), as part of Collaborative Research Center (Sonderforschungsbereich) 1320 EASE – Everyday Activity Science and Engineering, University of Bremen (http://www.ease-crc.org/; the research was conducted in subproject P04). Finally, we would like to thank Daniel Große for extensive helpful discussions and Sören Tempel as well as Pascal Pieper for their help in implementing extensions to our RISC-V VP platform.

References

1 Waterman A, Asanović K. The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA. 2019
2 Waterman A, Asanović K. The RISC-V Instruction Set Manual; Volume II: Privileged Architecture. 2019
3 Herdt V, Große D, Drechsler R. Enhanced Virtual Prototyping: Featuring RISC-V Case Studies. Cham: Springer, 2020
4 de Schutter T. Better Software. Faster!: Best Practices in Virtual Prototyping. Mountain View: Synopsys Press, 2014
5 IEEE Std. 1666. IEEE Standard for Standard SystemC Language Reference Manual, 2012
6 Große D, Drechsler R. Quality-Driven SystemC Design. Berlin: Springer, 2010
7 Streubühr M, Rosales R, Hasholzner R, et al. ESL power and performance estimation for heterogeneous MPSOCS using SystemC. In: Proceedings of Forum for Specification and Design Languages (FDL), 2011. 1–8
8 Grüttner K, Görgen R, Schreiner S, et al. CONTREX: design of embedded mixed-criticality control systems under consideration of extra-functional properties. Microprocessors Microsyst, 2017, 51: 39–55
9 Onnebrink G, Leupers R, Ascheid G, et al. Black box ESL power estimation for loosely-timed TLM models. In: Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016. 366–371
10 Herdt V, Le H M, Große D, et al. On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate VP-based error effect simulation — a case study. In: Proceedings of Forum on Specification and Design Languages (FDL), 2016. 1–8
11 Herdt V, Le H M, Große D, et al. Towards early validation of firmware-based power management using virtual prototypes: a constrained random approach. In: Proceedings of Forum on Specification and Design Languages (FDL), 2017. 1–8
12 Herdt V, Le H M, Große D, et al. Maximizing power state cross coverage in firmware-based power management. In: Proceedings of the 24th Asia and South Pacific Design Automation Conference, 2019. 335–340
13 Herdt V, Große D, Pieper P, et al. RISC-V based virtual prototype: an extensible and configurable platform for the system-level. J Syst Architecture, 2020, 109: 101756
14 Herdt V, Große D, Drechsler R. RVX — a tool for concolic testing of embedded binaries targeting RISC-V platforms. In: Proceedings of Automated Technology for Verification and Analysis, 2020
15 Herdt V, Große D, Le H M, et al. Early concolic testing of embedded binaries with virtual prototypes: a RISC-V case study. In: Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC), 2019. 1–6
16 Herdt V, Große D, Wloka J, et al. Verification of embedded binaries using coverage-guided fuzzing with SystemC-based virtual prototypes. In: Proceedings of the Great Lakes Symposium on VLSI, 2020. 101–106
17 Pieper P, Herdt V, Große D, et al. Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020
18 Herdt V, Große D, Le H M, et al. Extensible and configurable RISC-V based virtual prototype. In: Proceedings of Forum on Specification and Design Languages, 2018. 5–16
19 Herdt V, Große D, Jentzsch E, et al. Efficient cross-level testing for processor verification: a RISC-V case-study. In: Proceedings of Forum for Specification and Design Languages (FDL), 2020
20 Herdt V, Große D, Drechsler R. Closing the RISC-V compliance gap: looking from the negative testing side. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020
21 Herdt V, Große D, Drechsler R. Towards specification and testing of RISC-V ISA compliance. In: Proceedings of the 23rd Conference on Design, Automation and Test in Europe, 2020. 995–998
22 Herdt V, Große D, Le H M, et al. Verifying instruction set simulators using coverage-guided fuzzing. In: Proceedings of Design, Automation and Test in Europe Conference & Exhibition, 2019
23 Herdt V, Le H M, Grobe D, et al. Verifying SystemC using intermediate verification language and stateful symbolic simulation. IEEE Trans Comput-Aided Des Integr Circ Syst, 2019, 38: 1359–1372
24 Herdt V, Drechsler R. Efficient techniques to strongly enhance the virtual prototype based design flow. In: Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2020
25 Binkert N, Beckmann B, Black G, et al. The GEM5 simulator. SIGARCH Comput Archit News, 2011, 39: 1–7
26 Mueller-Gritschneder D, Dittrich M, Greim M, et al. The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping. In: Proceedings of International Symposium on Rapid System Prototyping (RSP), 2017. 79–84
27 Devarajegowda K, Fadiheh M R, Singh E, et al. Gap-free processor verification by S2QED and property generation. In: Proceedings of Design, Automation Test in Europe Conference & Exhibition (DATE), 2020. 526–531
28 Fadiheh M R, Müller J, Brinkmann R, et al. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020
29 Adir A, Almog E, Fournier L, et al. Genesys-pro: innovations in test program generation for functional processor verification. IEEE Des Test Comput, 2004, 21: 84–93
30 Campbell B, Stark I. Randomised testing of a microprocessor model using SMT-solver state generation. In: Proceedings of Formal Methods for Industrial Critical Systems, 2014. 185–199
31 Katz Y, Rimon M, Ziv A. Generating instruction streams using abstract CSP. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012. 15–20
32 Chupilko M, Kamkin A, Kotsynyak A, et al. MicroTESK: specification-based tool for constructing test program generators. In: Proceedings of the 13th International Haifa Verification Conference, 2017
33 Fine S, Ziv A. Coverage directed test generation for functional verification using bayesian networks. In: Proceedings of Design Automation Conference, 2003. 286–291
34 Ioannides C, Barrett G, Eder K. Feedback-based coverage directed test generation: an industrial evaluation. In: Proceedings of the 6th International Haifa Verification Conference, 2011
35 Martignoni L, Paleari R, Roglia G F, et al. Testing CPU emulators. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, 2009. 261–272
36 Bombieri N, Fummi F, Pravadelli G. Incremental ABV for functional validation of TL-to-RTL design refinement. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition, 2007. 882–887

37 Ecker W, Esen V, Hull M. Implementation of a transaction level assertion framework in SystemC. In: Proceedings of the 10th Design, Automation and Test in Europe Conference and Exhibition, 2007. 1–6
38 Ferro L, Pierre L. ISIS: runtime verification of TLM platforms. In: Proceedings of Forum on Specification & Design Languages (FDL), 2009. 1–6
39 Tabakov D, Vardi M Y. Monitoring temporal SystemC properties. In: Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), 2010. 123–132
40 Godefroid P. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Berlin: Springer, 1996
41 Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2005. 110–121
42 Kundu S, Ganai M, Gupta R. Partial order reduction for scalable testing of SystemC TLM designs. In: Proceedings of the 45th Annual Design Automation Conference, 2008. 936–941
43 Blanc N, Kroening D. Race analysis for SystemC using model checking. ACM Trans Des Autom Electron Syst, 2010, 15: 1–32
44 Moy M, Maraninchi F, Maillet-Contoz L. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. Des Autom Embed Syst, 2005, 10: 73–104
45 Karlsson D, Eles P, Peng Z. Formal verification of SystemC designs using a petri-net based representation. In: Proceedings of the Conference on Design, Automation and Test in Europe, 2006. 1228–1233
46 Traulsen C, Cornet J, Moy M, et al. A SystemC/TLM semantics in Promela and its possible applications. In: Proceedings of International SPIN Workshop on Model Checking of Software, 2007. 204–222
47 Herber P, Fellmuth J, Glesner S. Model checking SystemC designs using timed automata. In: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2008. 131–136
48 Kroening D, Sharygina N. Formal verification of SystemC by automatic hardware/software partitioning. In: Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design, 2005. 101–110
49 Cimatti A, Narasamdya I, Roveri M. Software model checking SystemC. IEEE Trans Comput-Aided Des Integr Circ Syst, 2013, 32: 774–787
50 Große D, Le H M, Drechsler R. Proving transaction and system-level properties of untimed SystemC TLM designs. In: Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2010), 2010. 113–122
51 Chou C N, Ho Y S, Hsieh C, et al. Symbolic model checking on SystemC designs. In: Proceedings of the 49th Annual Design Automation Conference, 2012. 327–333
52 Chou C N, Chu C K, Huang C Y R. Conquering the scheduling alternative explosion problem of SystemC symbolic simulation. In: Proceedings of IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, 2013. 685–690
53 Herber P, Pockrandt M, Glesner S. State — a systemc to timed automata transformation engine. In: Proceedings of the 17th International Conference on High Performance Computing and Communications (HPCC), IEEE 7th International Symposium on Cyberspace Safety and Security (CSS) and IEEE 12th International Conference on Embedded Software and Systems (ICESS), 2015. 1074–1077
54 Lin B, Cong K, Yang Z, et al. Concolic testing of systemc designs. In: Proceedings of the 19th International Symposium on Quality Electronic Design (ISQED), 2018. 1–7
55 Lin B, Xie F. A systematic investigation of state-of-the-art SystemC verification. J Circuit Syst Comp, 2020, 29: 2030013
56 Cadar C, Dunbar D, Engler D R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008. 209–224
57 Godefroid P, Levin M Y, Molnar D A. Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, 2008
58 Chipounov V, Kuznetsov V, Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. SIGARCH Comput Archit News, 2011, 39: 265–278
59 Cha S K, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code. In: Proceedings of IEEE Symposium on Security and Privacy, 2012. 380–394
60 Shoshitaishvili Y, Wang R, Salls C, et al. SOK: (state of) the art of war: offensive techniques in binary analysis. In: Proceedings of IEEE Symposium on Security and Privacy, 2016. 138–157
61 Herdt V, Le H M, Große D, et al. Combining sequentialization-based verification of multi-threaded C programs with symbolic partial order reduction. Int J Softw Tools Technol Transfer, 2019, 21: 545–565
62 Regehr J, Cooprider N. Interrupt verification via thread verification. Electron Notes Theor Comput Sci, 2007, 174: 139–150
63 Horn A, Tautschnig M, Val C G, et al. Formal co-validation of low-level hardware/software interfaces. In: Proceedings of Formal Methods in Computer-Aided Design, 2013. 121–128
64 Ahn S, Malik S. Automated firmware testing using firmware-hardware interaction patterns. In: Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2014. 1–10
65 Mukherjee R, Purandare M, Polig R, et al. Formal techniques for effective co-verification of hardware/software co-designs. In: Proceedings of the 54th Annual Design Automation Conference, 2017. 1–6
66 Davidson D, Moench B, Ristenpart T, et al. FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution. In: Proceedings of USENIX Security, 2013. 463–478
67 Zaddach J, Bruno L, Francillon A, et al. AVATAR: a framework to support dynamic security analysis of embedded systems' firmwares. In: Proceedings of Network and Distributed System Security Symposium, 2014
68 Lee H, Choi K, Chung K, et al. Fuzzing can packets into automobiles. In: Proceedings of IEEE 29th International Conference on Advanced Information Networking and Applications, 2015. 817–821
69 Alimi V, Vernois S, Rosenberger C. Analysis of embedded applications by evolutionary fuzzing. In: Proceedings of International Conference on High Performance Computing & Simulation (HPCS), 2014. 551–557
70 van den Broek F, Hond B, Torres A C. Security testing of GSM implementations. In: Proceedings of International Symposium on Engineering Secure Software and Systems, 2014. 179–195
71 Muench M, Stijohann J, Kargl F, et al. What you corrupt is not what you crash: challenges in fuzzing embedded devices. In: Proceedings of Network and Distributed System Security Symposium, 2018
72 OSCI. OSCI TLM-2.0 Language Reference Manual, 2009
73 Herdt V, Große D, Drechsler R. Fast and accurate performance evaluation for RISC-V using virtual prototypes. In: Pro-

ceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020. 618–621
74 Herdt V, Große D, Tempel S, et al. Adaptive simulation with virtual prototypes in an open-source RISC-V evaluation platform. J Syst Architecture, 2021, 116: 102135
75 Vardi M Y. Formal techniques for SystemC verification. In: Proceedings of the 44th Annual Design Automation Conference, 2007. 188–192
76 Le H M, Große D, Herdt V, et al. Verifying SystemC using an intermediate verification language and symbolic simulation. In: Proceedings of the 50th Annual Design Automation Conference, 2013. 1–6
77 Le H M, Herdt V, Große D, et al. Towards formal verification of real-world SystemC TLM peripheral models – a case study. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition, 2016. 1160–1163
78 Herdt V. Complete Symbolic Simulation of SystemC Models: Efficient Formal Verification of Finite Non-Terminating Programs. Berlin: Springer, 2016
79 Herdt V, Le H M, Große D, et al. Compiled symbolic simulation for SystemC. In: Proceedings of the 35th International Conference on Computer-Aided Design, 2016. 1–8
80 Herdt V, Le H M, Große D, et al. ParCoSS: efficient parallelized compiled symbolic simulation. In: Proceedings of International Conference on Computer Aided Verification, 2016. 177–183
81 Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Commun ACM, 1990, 33: 32–44
82 Suh G E, Lee J W, Zhang D, et al. Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004. 85–96
83 Hedin D, Sabelfeld A. A perspective on information-flow control. In: Proceedings of Software Safety and Security - Tools for Analysis and Verification, 2012. 319–347
84 Denning D E R. Cryptography and Data Security. Boston: Addison-Wesley Longman Publishing Co., Inc., 1982
85 Herdt V, Tempel S, Große D, et al. Mutation-based compliance testing for RISC-V. In: Proceedings of the 26th Asia and South Pacific Design Automation Conference, 2021. 55–60
86 Hassan M, Herdt V, Le H M, et al. Data flow testing for virtual prototypes. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition, 2017. 380–385
87 Bringmann O, Ecker W, Gerstlauer A, et al. The next generation of virtual prototyping: ultra-fast yet accurate simulation of HW/SW systems. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition, 2015. 1698–1707
88 Cornaglia A, Hasan M S, Viehl A, et al. JIT-based context-sensitive timing simulation for efficient platform exploration. In: Proceedings of the 25th Asia and South Pacific Design Automation Conference, 2020. 369–374
89 Böhm I, Franke B, Topham N. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In: Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2010. 1–10
90 Charif A, Busnot G, Mameesh R, et al. Fast virtual prototyping for embedded computing systems design and exploration. In: Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools, 2019. 1–8
91 Topham N, Franke B, Jones D, et al. Adaptive High-Speed Processor Simulation. Berlin: Springer, 2010. 145–159
92 Beltrame G, Sciuto D, Silvano C. Multi-accuracy power and performance transaction-level modeling. IEEE Trans Comput-Aided Des Integr Circ Syst, 2007, 26: 1830–1842
93 Herdt V, Le H M, Große D, et al. On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate VP-based error effect simulation — a case study. In: Proceedings of Forum on Specification and Design Languages (FDL), 2016. 1–8
94 Mueller-Gritschneder D, Sharif U, Schlichtmann U. Performance and accuracy in soft-error resilience evaluation using the multi-level processor simulator ETISS-ML. In: Proceedings of the International Conference on Computer-Aided Design, 2018. 1–8
95 Herdt V, Le H M, Große D, et al. Towards fully automated TLM-to-RTL property refinement. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition, 2018. 1508–1511
96 Clemens J. Learning device models with recurrent neural networks. In: Proceedings of International Joint Conference on Neural Networks (IJCNN), 2018. 1–8
97 Vörtler T, Einwich K, Hassan M, et al. Using constraints for SystemC AMS design and verification. In: Proceedings of Design and Verification Conference & Exhibition Europe, 2018
98 Hassan M, Große D, Vörtler T, et al. Functional coverage-driven characterization of RF amplifiers. In: Proceedings of Forum for Specification and Design Languages (FDL), 2019. 1–8