Cross-Level Processor Verification via Endless Randomized Instruction Stream Generation with Coverage-guided Aging

Niklas Bruns[1] Vladimir Herdt[1,2] Eyck Jentzsch[3] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[3]MINRES Technologies GmbH, 85579 Neubiberg, Germany
{nbruns,vherdt,drechsler}@uni-bremen.de eyck@minres.com

***Abstract*—We propose a novel cross-level verification approach for processor verification at the *Register-Transfer Level* (RTL). The foundation is a randomized coverage-guided instruction stream generator that produces one endless and unrestricted instruction stream that evolves dynamically at runtime. We leverage an *Instruction Set Simulator* (ISS) as a reference model in a tight co-simulation setting. Coverage information is continuously updated based on the execution state of the ISS and we employ Coverage-guided Aging to smooth out the coverage distribution of the randomized instruction stream over the time. In combination, this enables a broad and deep coverage to find intricate corner-case bugs in the RTL processor. Our case study with an industrial pipelined 32 bit RISC-V processor demonstrate the effectiveness of our approach.**

I. Introduction

Extensive processor verification at the *Register-Transfer Level* (RTL) is essential to detect intricate bugs, which could lead to enormous follow-up costs and additional design iterations. Simulation-based methods that rely on continuous processor-level stimuli generation are still prevalent and form the backbone of the verification effort due to their ease of use and scalability. In this paper we consider RISC-V [1], [2] as a representative *Instruction Set Architecture* (ISA) which serves as foundation for modern processor architectures, in particular in the embedded application domain. RISC-V is a free and open-source ISA that enables a royalty-free processor design and implementation. It is designed in a very modular way with optional standard instruction set extensions around a mandatory base integer instruction set and the ability to integrate additional custom instruction sets to build highly application-specific processors. These properties made RISC-V very popular in industry and academia. From the verification perspective, however, the extensive modularity adds additional complexity. Besides the modern features provided by RISC-V and any micro-architectural specific optimizations of the processor, such as pipelining and branch prediction, the verification tools also need to be able to deal with the large configuration space offered by RISC-V. Promising approaches in this context leverage methods based on co-simulation that employ an *Instruction Set Simulator* (ISS) (i.e. an executable abstract model of the processor core, typically implemented in C++) as a functional reference model for the RTL processor under test. Such a method is used by Google's open-source RISC-V *Design Verification* (DV) framework. It applies constraint-based specification techniques in SystemVerilog to generate RISC-V assembly tests one after another. Different RISC-V instruction sets are supported by selecting and combining the respective constraint-based specifications. Execution results between the ISS and RTL processor core are compared through execution log files. While this feature set makes RISC-V DV very powerful in general, it also has some major weaknesses. In order to keep the framework generic, the generated tests use a restricted instruction set to avoid problems with infinite loops and platform-dependent memory access operations. Moreover, by generating tests one by one, only comparatively short instruction sequences are considered, and the state of the processor under test is regularly reset for each new test execution. Furthermore, the co-simulation has an inherent performance overhead due to the extensive filesystem communication, since each RISC-V assembly test needs to be compiled, loaded onto the respective simulator, and produce a log file for comparison. Finally, the test generator is not designed to be dynamically guided by coverage information obtained from the test execution progress. Many of these issues have been addressed by a recent academic work [3]. It generates endless instruction streams and integrates the ISS with the RTL core in a very efficient co-simulation compiled into a single binary with in-memory communication. The setup allows to generate instructions without any restrictions, i.e., arbitrary combinations of load/store and *Control and Status Registers* (CSRs)[1] instructions, as well as infinite loops, are supported, which enables a very comprehensive test approach. However, the approach is still limited as it does not collect or employ runtime coverage information to assess and guide the test generation process. Instead, the instruction stream generators are based on a simple randomized test strategy which makes it very difficult to continuously achieve a broad and deep test

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

[1] In the CSRs, the processor stores additional instruction results to enable sophisticated hardware/software interactions.

coverage in endless instruction streams.

In this paper, we propose a novel cross-level verification approach that conceptually builds upon the previous academic work [3] and addresses the aforementioned limitations. The foundation is a randomized coverage-guided instruction stream generator that produces an endless and unrestricted instruction stream that evolves dynamically at runtime based on observed coverage information. We also leverage an ISS as a reference model in a tight co-simulation setting. Coverage information is continuously updated based on the execution state of the ISS and we employ the novel concept of *Coverage-guided Aging* to smooth out the coverage distribution of the randomized instruction stream over time. In combination, this enables a broad and deep coverage to find intricate corner-case bugs in the RTL core. Our experiments with the 32-bit pipelined RISC-V core of the MINRES *The Good Core* (TGC) series demonstrate the effectiveness of our approach. We achieve a much more regular coverage distribution of the randomized instruction stream via Coverage-guided Aging, and we found another intricate micro-architecture related bug in the interplay between the already heavily tested industrial processor with the accompanied test bench infrastructure.

II. Related Work

Several approaches have been proposed to generate tests for the purpose of processor verification. One prominent direction is to employ model-based test generators that leverage a constraint-based specification format to guide the test generation process [4], [5]. In this context, optimization techniques for constraint propagation [6], execution path coverage models [7] and mining techniques for processor manuals [8] have been considered. Alternative approaches integrate coverage-guided test generation based on bayesian networks [9] and other machine learning techniques [10] as well as fuzzing [11] and symbolic execution [12]. However, these approaches are either not designed for RTL verification or impose restrictions on the generated instruction streams. In addition, they do not target the modern RISC-V ISA.

Recently, verification approaches tailored for RISC-V have emerged. In the introduction, we already covered the modern co-simulation based approaches that are tailored for RTL and are closest to our proposed approach. Other simulation-based approaches for RISC-V generate instruction sequences by combining pre-defined randomized patterns [13] and by utilizing constraint-based specifications [14] as well as coverage-guided fuzzing techniques [15]. However, they suffer from the same limitations as the traditional processor-level stimuli generation approaches in imposing restrictions or operating at a different abstraction level than RTL. Finally, a set of directed test-suites that cover different RISC-V instruction sets [16]–[18] are available that form a baseline for testing and looking beyond simulation-based techniques. A few formal approaches that are based on model checking techniques [19], [20] have been proposed as well. Nevertheless, these formal techniques are possibly susceptible to scalability issues.

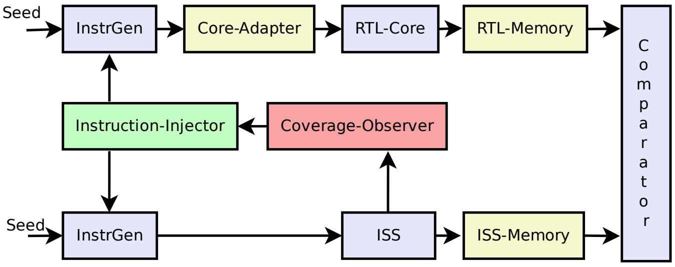

Fig. 1. Overview on core verification

III. Background on RISC-V

RISC-V, is a free and open *Instruction Set Architecture* (ISA) that was developed at UC Berkeley and is available under the open-source license: Creative Commons Attribution 4.0 International License [1], [2]. RISC-V provides three different integer base ISAs that differ primarily in the used word width: RV32I is the 32-bit version of the architecture, RV64I is the 64-bit, and RV128I is the 128-bit version. These base ISAs define integer calculations, program control, load and store operations, and debugging instructions. In addition to this base ISAs, many instruction set extensions are defined. The used extensions are appended to the name of the integer base ISA to name the capabilities of a core implementation. A 32-bit RISC-V processor with a multiplication unit, CSR instructions, Fence, and support for compressed instructions is called RV32IMCZicsrZifencei.

IV. Cross-Level Processor Verification with Coverage-guided Aging

In this section, we present our cross-level processor verification approach that is based on endless randomized instruction stream generation using Coverage-guided Aging. We start with an overview.

A. Overview

Fig. 1 shows the overview of our approach. It starts with initializing the random instruction generators (InstrGen). Each core has its separate instruction generator which are initialized with the same cryptographic seeds. As a consequence, the generators provide the same endless randomized instruction stream. At first, some instructions of the endless instruction stream are generated and executed by the ISS. After this, the RTL processor fetches its instruction stream. However, for the fetching of the RTL core, micro-architectural details such as pipelining, pre-fetching, and fetch-buffering have to be considered. For this purpose, a core adapter is used, which checks for addresses that were not fetched by the ISS, fills them with randomized values (not generated by InstrGen), and forwards them to the RTL-Core. After the execution of the instructions, the core and ISS write the results to the separated memories. Next, the Coverage-Observer measures the functional coverage based on the ISS execution state, does the coverage-aging, and gives hints to the Instruction-Injector if functionality must be covered (again. In principle, the functional coverage can be specified arbitrarily complex and is used to guide the test generation over time. We will present more details on the Coverage-Observer in Section IV-B. Next, the Instruction-Injector evaluates the hints and injects instructions to cover the requested functionality. The injector must consider that the cores have different fetch behaviors and execution timings that result in individual random instruction generator states. The functional principle of the Instruction-Injector is described in Section IV-C. The purpose of the Comparator is to find functional differences between the RTL-Core and the ISS. To achieve this, it compares the register values of the ISS and the RTL-Core. The matching is not straightforward because the cores do not have the same timing behavior. The Comparator logs the value changes and constantly compares the two changes at the same position to solve this problem. If the Comparator finds any differences, then it quits the simulation. In the following, we provide more details on the Coverage-Observer (Section IV-B) and Instruction-Injector (Section IV-C), which are the two most important components to implement coverage-guided aging.

B. Coverage-Observer

The main functionality of the Coverage-Observer is to monitor the internal state of the ISS to measure the coverage. It samples the executed instructions and looks up the matching coverage points. In this work, we define the cross-product of instruction groups as coverage points. The instruction groups are defined by a verification engineer to lay the verification focus at the to-be-tested functionality. An instruction group covers a set of instructions like arithmetic or load/store instructions. Consequently, our approach guarantees to verify each functionality in combination with every function. The Coverage-Observer watches the executed instructions at run-time and is the heart of our coverage aging extension. After an instruction sequence covers an coverage point, the Coverage-Observer sets the corresponding Coverage-guided Aging counters to a defined maximal value. Periodically, the Coverage-Observer decreases the Coverage-guided Aging counter until the minimum limit is reached. In this case, it gives a hint to the Instruction-Injector. This hint consists of a random instruction sequence that is needed to cover the coverage point. The instructions are randomly selected instructions that were sampled in this run dynamically. The Coverage-Observer will reset the Coverage-guided Aging counter if the groups are covered again. Next we describe the Instruction-Injector.

C. Instruction-Injector

The purpose of the Instruction-Injector is to inject instruction sequences into the random test generators in compliance with their internal state. When the instruction injection ignores the internal states, then the generators provide differing instruction streams that may lead to a false result of the Comparator. To achieve a legal injection, the Instruction-Injector measures how many instructions have been executed before the current state of the random generator was reached. Then, it schedules the injection to the same near-future instruction count for all instruction generators. This approach is valid because deterministic random sources, that are initiated with the same cryptographic seed value, provide the same random sequences. In this way, we have ensured that the behaviors of the random instruction generators are equal.

V. Evaluation

In this section, we present our case study and discuss the evaluation results. The goal of our case study is to evaluate the applicability of Coverage-guided Aging for cross-level processor verification. We start with the test setup.

A. Test Setup

As *Device Under Test* (DUT), we used the 32-bit pipelined RISC-V core of the MINRES *The Good Core* (TGC) series, which has already been extensively verified using simulation-based approaches and formal techniques. As reference ISS, we used the ISS of the open-source SystemC-based RISC-V VP[2]. To enable the co-simulation, we translated the industrial RTL core to C++ using the open-source tool Verilator[3] and integrated it into a SystemC test bench along with the ISS. For our evaluation, we configured the core and ISS to support the RISC-V subset RV32IMCZicsrZifencei (see: Section III). All experiments were executed on an Ubuntu 20.04 LTS machine with an AMD Ryzen 7 PRO 4750U CPU with 4.1GHz and 36GB RAM and a SystemC simulation time limit of 1 second ($\approx$ 20 million instructions). By analyzing the RISC-V specification, we identified the following six important instruction groups that act as base for the coverage points in this case study: Arithmetic, Control Flow, Memory, Special & System, Control & Status Register (CSR), and Other. The group *Arithmetic* contains all arithmetic instructions of the instruction subsets RV32I and RV32C and all instructions of RV32M. The group *Control Flow* contains the unconditional jump and the conditional branch instructions of RV32I and RV32C. The group *Memory* contains the load/store instructions of RV32I and RV32C and the memory ordering instructions of RV32I. The group *Special & System* contains the ECALL and EBREAK, the NOP and the HINT instructions of RV32I, and the illegal, NOP, breakpoint, and HINT instructions of RV32C. Additionally, it contains the FENCE instruction of ZIFENCEI. The group *CSR* is equivalent to ZICSR. The group *Other* contains all instructions of the undefined and unsupported subsets and the privileged architecture. As a consequence of the six instruction groups and the resulting 36 coverage points, we configured the Coverage-guided Aging counter to the value 100 and will be decremented after a new instruction is generated. With the value 100, there are enough random instructions, and at the same time, the coverage points are triggered frequently. In the following, we compare the results of a random test generator with and without our Coverage-guided Aging extension (Section V-B). Then we present a bug that we found during the development process (Section V-C).

B. Random vs. Coverage-guided Aging

Fig. 2 shows the result bar chart of our case study. The chart gives information about how often the coverage points (defined as cross product of the instruction groups) were executed by the

[2] https://github.com/agra-uni-bremen/riscv-vp

[3] https://www.veripool.org/verilator/

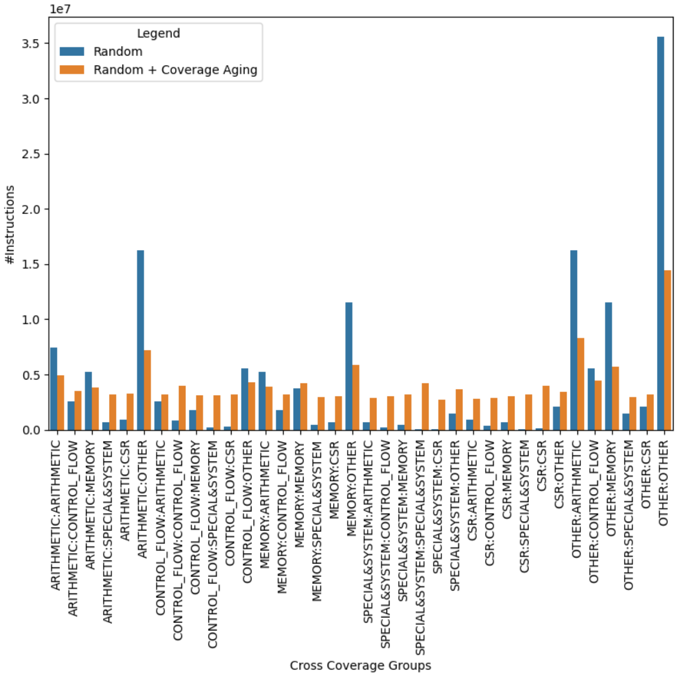

Fig. 2. Cross Coverage Groups : Sum of all runs

random test generator and the Coverage-guided Aging test generator. The random generator is a re-implementation of the test generator of [3] and has already proven its excellent bug-hunting capabilities. Unfortunately, it tends to favor specific test state spaces. It is based on a static randomized test strategy that does not change over time. However, such an adjustment is critical since we are looking at an endless instruction stream and not at individual cases where readjustment after each run is possible. As stated in the legend, the blue bars, which are always on the left side, represent the instructions generated by the random test generator and the orange bars belonging to the test generator that is enhanced with Coverage-guided Aging. The execution of the random test generator leads to substantial peaks in specific combinations of instruction groups while other combinations were almost never executed. For example, the count of *Special & System : Special & System* is so low that it almost can not be seen, and in opposite, the combination of *Other : Other* was executed very often. Thus, clear gaps can be seen. In contrast, the Coverage-guided Aging generator has much weaker peaks on certain groups. In addition, every group is executed and always reaches a clearly visible execution count. Thus, the result of the random test generator seems to degenerate. In comparison, the Coverage-guided Aging test generator provides a more balanced result, and no gaps can be seen. Unfortunately, the results could not be presented for space reasons. Thus, we have shown that Coverage-guided Aging complements to close gaps and achieves more balanced verification results.

C. Detected Pipeline Bug

During the development of the Coverage-guided Aging test generator we have discovered a micro-architectural related bug in the accompanied test bench adapter of the already well-tested industrial RTL-Core. In certain test cases, there where no free entries in the execute FIFO of the pipeline and thus the core did not receive any further instructions. This was triggered because the pipeline was only emptied by the test bench adapter when a valid instruction was executed. Therefore, a test case could trigger this error if the core ran too many invalid instructions (see: Special & System : Special & System in Fig. 2) in succession.

D. Discussion and Future Work

Our case study shows, that Coverage-guided Aging is a effective extension for cross-level processor verification. We have shown that Coverage-guided Aging complements to close gaps and achieves a much more regular coverage distribution. Furthermore we found another intricate micro-architectural bug in the already heavily tested industrial processor. For future work, we plan to design advanced micro-architecture coverage metrics to measure specific feature testing like the hazard handling of pipelines. In addition, we plan to create a processor verification benchmark based on finely detailed coverage groups.

References

[1] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, 2019.

[2] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, 2019.

[3] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient cross-level testing for processor verification: A risc- v case-study," in *FDL*, 2020, pp. 1–7.

[4] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: innovations in test program generation for functional processor verification," *D&T*, pp. 84–93, 2004.

[5] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using SMT-solver state generation," in *Formal Methods for Industrial Critical Systems*, F. Lang and F. Flammini, Eds., 2014, pp. 185–199.

[6] Y. Katz, M. Rimon, and A. Ziv, "Generating instruction streams using abstract CSP," in *DATE*, 2012, pp. 15–20.

[7] M. Chupilko, A. Kamkin, A. Kotsynyak, and A. Tatarnikov, "MicroTESK: specification-based tool for constructing test program generators," in *HVC*, 2017.

[8] W. Ma, A. Forin, and J. Liu, "Rapid prototyping and compact testing of CPU emulators," in *RSP*, 2010, pp. 1–7.

[9] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, 2003, pp. 286–291.

[10] C. Ioannides, G. Barrett, and K. Eder, "Feedback-based coverage directed test generation: An industrial evaluation," in *Hardware and Software: Verification and Testing*, S. Barner, I. Harris, D. Kroening, and O. Raz, Eds., 2011.

[11] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU emulators," in *ISSTA*, 2009, pp. 261–272.

[12] H. Wagstaff, T. Spink, and B. Franke, "Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators," in *CASES*, 2014, pp. 1–10.

[13] "RISC-V torture test generator," https://github.com/ucb-bar/riscv-torture.

[14] V. Herdt, D. Große, and R. Drechsler, "Towards specification and testing of RISC-V ISA compliance," in *DATE*, 2020, pp. 995–998.

[15] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.

[16] "RISC-V ISA tests," https://github.com/riscv/riscv-tests.

[17] "RISC-V compliance task group," https://github.com/riscv/riscv-compliance.

[18] N. Bruns, V. Herdt, D. Große, and R. Drechsler, "Toward RISC-V CSR compliance testing," *IEEE ESL*, vol. 13, no. 4, pp. 202–205, 2021.

[19] "RISC-V formal verification framework," https://github.com/SymbioticEDA/riscv-formal, 2020.

[20] "OneSpin 360 DV RISC-V Verification App," https://www.onespin.com/solutions/risc-v, 2020.