Bounded Load/Stores in Grammar-based Code Generation for Testing the RISC-V Vector Extension

Manfred Schlägl and Daniel Große

Institute for Complex Systems, Johannes Kepler University Linz, Austria
manfred.schlaegl@jku.at, daniel.grosse@jku.at

Abstract

In this paper, we consider a Grammar-based fuzzing framework for testing the RISC-V "V" Vector Extension. We focus on one of the major challenges, namely generating valid vector load/store instruction sequences by extending a context-free grammar with functions to create elements in a context-sensitive way.

Introduction

In recent years, the open and royalty-free Instruction Set Architecture (ISA) RISC-V [1] has gained significant traction in both academia and industry. A distinctive feature of the RISC-V ISA is its high degree of modularity, which is achieved through a wide range of extensions that can be seamlessly integrated with a minimal base ISA to tailor it to specific application requirements. One outstanding extension is the RISC-V "V" Vector Extension (RV), which was recently ratified in version 1.0 [2]. With 624 new instructions and 32 vector registers, the extension introduces extensive Single Instruction, Multiple Data (SIMD) capabilities to the RISC-V architecture. In SIMD, operations are executed not only on individual data elements but on entire vectors of elements simultaneously. With this, SIMD leverages Data-Level Parallelism (DLP) to enhance data throughput and overall performance of parallelizable algorithms like those often utilized in machine learning and multimedia applications [3].

Recently, RVV was integrated in the open-source, SystemC TLM (IEEE 1666, [4]) based Virtual Prototype (VP) RISC-V VP+ [5, 6]. VPs are high-level, executable models of the entire Hardware (HW) platforms which can run unmodified production Software (SW) [7] and therefore allow early design space exploration, and system evaluation and validation. The paper [5] also presents the verification method of the RVV integration. Here, the authors use the FORCE-RISCV Instruction Sequence Generator (ISG) provided by the OpenHW Group [8] to generate RVV programs. The generated programs are then executed in a reference simulator and in the Simulator under Test (SuT), the VP. Finally, the traces of each execution are compared for differences. With this method, the authors were able to achieve a functional coverage beyond 94% (according to riscvOVPsim), based on a generated test set consisting of 100 test cases with over 12k instructions each. Furthermore, this new approach does not depend on execution traces, but instead uses system state comparison after execution (registers, memory, ...). With this, it will be possible to perform the verification on any target for which state extraction is possible, e.g. on real HW.

Our new verification approach, including code generators and pre-generated test sets, will be released as open source in the near future.

In this paper we discuss one of the major challenges in developing our new verification approach, namely the generation of valid load/stores with a grammar-based code generator. In the next section, we will briefly introduce grammar-based code generation with focus on RVV. After that, we will discuss the complexities associated with generating valid load/store instructions and propose a solution approach, illustrated on a specific RVV load/store operation.

Grammar-based Code Generation

Grammars can be used to generate syntactically valid input. In our case, a grammar is utilized several times to create a valid assembler program step by step, which is then translated into machine code and executed.

Listing 1 shows a small excerpt of our context-free grammar for generating RVV instructions. Technically, the grammar is written in Python as a dictionary, mapping from strings to lists of strings, with each string describing a symbol of the grammar. Symbols written in pointed brackets (e.g. "<start>" in Line 2) are non-terminal symbols. Symbols without pointed brackets (e.g. ", v0.t" in Line 10) are terminal symbols.

Listing 1: Excerpt of a context-free grammar for RVV

```
RVVGrammar = {
    "<start>": ["<instr_v_config">", "<instr_v_compute>", ... ],
    "<instr_v_config>": ["<instr_v_compute>"],
    "<instr_v_compute>": ["<instr_v_vector_int>"],
    "<instr_v_vector_int>": ["vadd <.vv>", "vadd <.vx>", "vadd <.vi>", ... ],
    "<.vi>": ["<vreg>", "<vs1>", "<imm5>", "<vm>"]
}
```

Excerpt of a context-free grammar for RVV

Listing 1 shows a small excerpt of our context-free grammar for generating RVV instructions. Technically, the grammar is written in Python as a dictionary, mapping from strings to lists of strings, with each string describing a symbol of the grammar. Symbols written in pointed brackets (e.g. "<start>" in Line 2) are non-terminal symbols. Symbols without pointed brackets (e.g. ", v0.t" in Line 10) are terminal symbols.
entry in the dictionary (= each line in Listing 1) describes an expansion rule for a non-terminal symbol (left side). The list (right side) describes the expansion alternatives and can contain non-terminal or terminal symbols. Non-terminal symbols are expanded according to the expansion rules until only terminal symbols remain. All remaining terminal symbols are integrated in the finally generated instruction as string. By randomly selecting expansion candidates, the grammar shown in Listing 1 can, for example, generate instruction strings such as “vadd.vv v2, v3, v4” or “vadd.vx v0, v3, x3, v0.t”.

Generating bounded Load/Stores

As we have shown in the last section, we can generate randomized instructions from our grammar. This holds also true for RVV load/store operations. For example, our grammar is able create the RVV unit stride store instruction vse8.v v1, (x5). This instruction stores the elements in vector register v1 to a memory location starting at an address specified by the value of integer register x5 and with an increment of 8 bits, or 1 byte per element (unit stride). However, since the value in x5 depends on the instructions executed so far, chances are high, that it does not point in a valid memory region (especially on RV64 with 64 bit address range). As a consequence, most generated load/store instructions will be invalid and lead to a fault in the execution. To get valid vse8.v instructions, it must be ensured that the values in the used integer register (plus the number of elements) are addresses in a valid range. In the following, we present the generation of bounded load/stores as a solution to this problem.

Prior to each load/store instruction, we generate code that ensures that the value of the used register is within a valid address range. However, this is not efficiently expressible in a context-free grammar. For example, the creation of specific values for the boundaries alone would inflate the grammar enormously. The solution is to extend the context-free grammar with new function symbols (in addition to non-/terminal symbols), which can generate strings in a context-sensitive way. Generation is done exactly as described above, but whenever the left side of an expansion is a function symbol, the corresponding function is called and its return value is used as result of the expansion. An example of such an extended grammar is presented in Listing 2. The newly introduced function symbol gen_v_store_vse8, which generates bounded vse8.v store instruction sequences is shown in Line 7.

To illustrate the concept, Listing 3 shows a sample pseudocode implementation of the gen_v_store_vse8 generation function. RISC-V assembler instructions are highlighted in blue. Global variables defining the valid address range and maximum store length are defined in Lines 1–4. Allocation of the source vector registers and the integer registers (address and scratch) is done in Lines 8–10. The calculation of the mask for the upper bound and the offset for the lower bound is performed in Lines 12–15. Code for masking the address with the upper bound mask is generated in Lines 17–19. Code generation for adding the lower bound offset is done in Lines 21–23. Finally, the vse8.v instruction is generated in Line 26.

This concludes our presentation of the concept of grammar-based bounded load/store code generation, based on a relatively simple vse8.v instruction. Overall, RVV supports a very powerful and extensive set of different and more complex load/store instructions for dealing with arrays and other data structures (e.g. strided, indexed). In the current state of our verification framework, the presented concept is used to realize code generation for all RVV load/store instructions. Our RVV verification framework, including code generators and pre-generated test sets, will be released as open source in the near future.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References