Chair of Electronic Design Automation
Department of Computer Engineering
School of Computation, Information and Technology
Technical University of Munich

Towards Automated LLVM Support and Autovectorization for RISC-V ISA Extensions

Philipp van Kempen, Mathis Salmen, Daniel Mueller-Gritschneder, Ulf Schlichtmann

Problem Statement

Motivation

- Exploring new RISC-V instruction for emerging workloads (automatically)
- Extending RISC-V with custom instructions needs many steps and can be time-consuming and error prone
- Eliminate manual efforts in Compiler Retargeting to enable ISA DSE

Compiler Retargeting

- The process of supporting new computing platforms in software development tools
- Support Levels:
  1. Assembler: Minimal support for custom instructions (ASM only)
  2. Intrinsics/Builtins: Manual insertion of custom instruction in high-level languages
  3. CodeGen: Pattern based instruction selection based on DAG
  4. Autovectorization: Automatic SIMD support (Loop-level & Basic-block)

Contributions

- Fully-automated model-based code generation for builtins and assembly-level
- Semi-automated pattern generator for scalar and SIMD ALU-type instructions
- Autovectorization support for “narrow” (sub-word) SIMD instructions GPRs

Inputs

A. CoreDSL Code [1]

- High-level Language to describe RISC-V Cores & their ISA
- Encoding, Assembly Syntax & Semantics

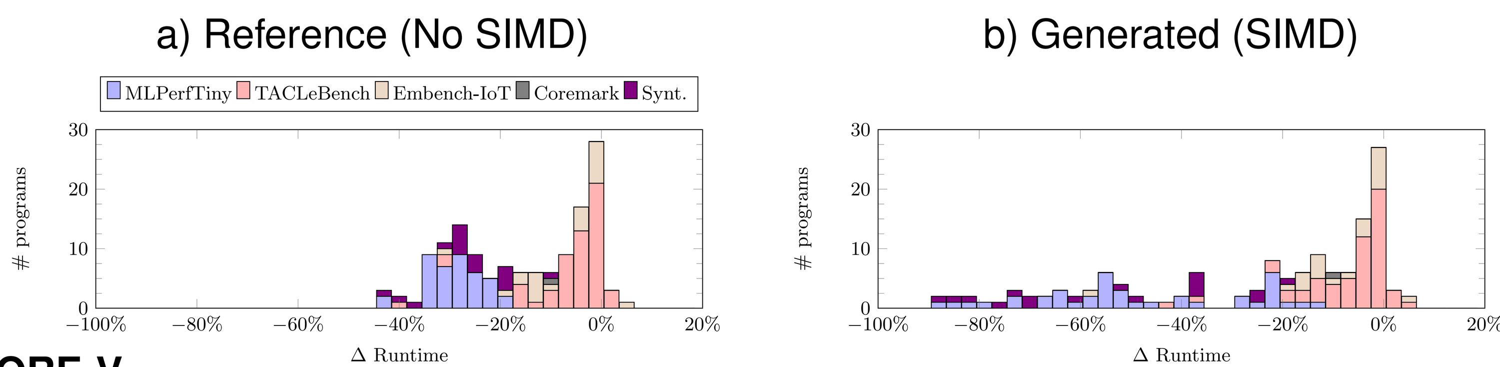
B. YAML Settings

- Configure Seal5 tools, passes, filters, logging,...

C. Test Sources

- Hand-written Assembly/Codegen tests

Evaluation

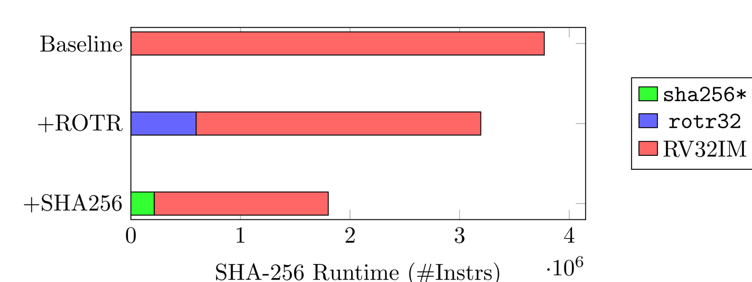


CORE-V

- Core-V Extension (implemented on CV32E40P [3])
  - 300+ MAC/Mem/ALU/Bitmanip/SIMD/ instructions
- Configurations
  - Baseline LLVM: Upstream LLVM 17 (`RV32IM`)
  - Core-V Reference LLVM: Developed by OpenHWGroup community (`RV32IM_XCVMac_XCVMem_XCVAlu`)
  - Seal5-generated LLVM (`RV32IM_XCVMac_XCVMem_XCVAlu[_XCVSimd]`)
- Benchmarks: 100+ programs (MLPerfTiny, Embench, TACLeBench, Coremark)
- Results: Without SIMD, Seal5 performs similar to Reference LLVM. With SIMD-support Seal5 outperforms Reference LLVM drastically.

SHA-256

- RISC-V Scalar Cryptography Extension [5] includes custom SHA256 operations
- Results: 52% reduction in number of executed instruction



Roadmap

Recent additions

- Migration from ISelDAG to GlobalISel
- Support compressed instructions, 64-bit targets and custom registers

Work in Progress

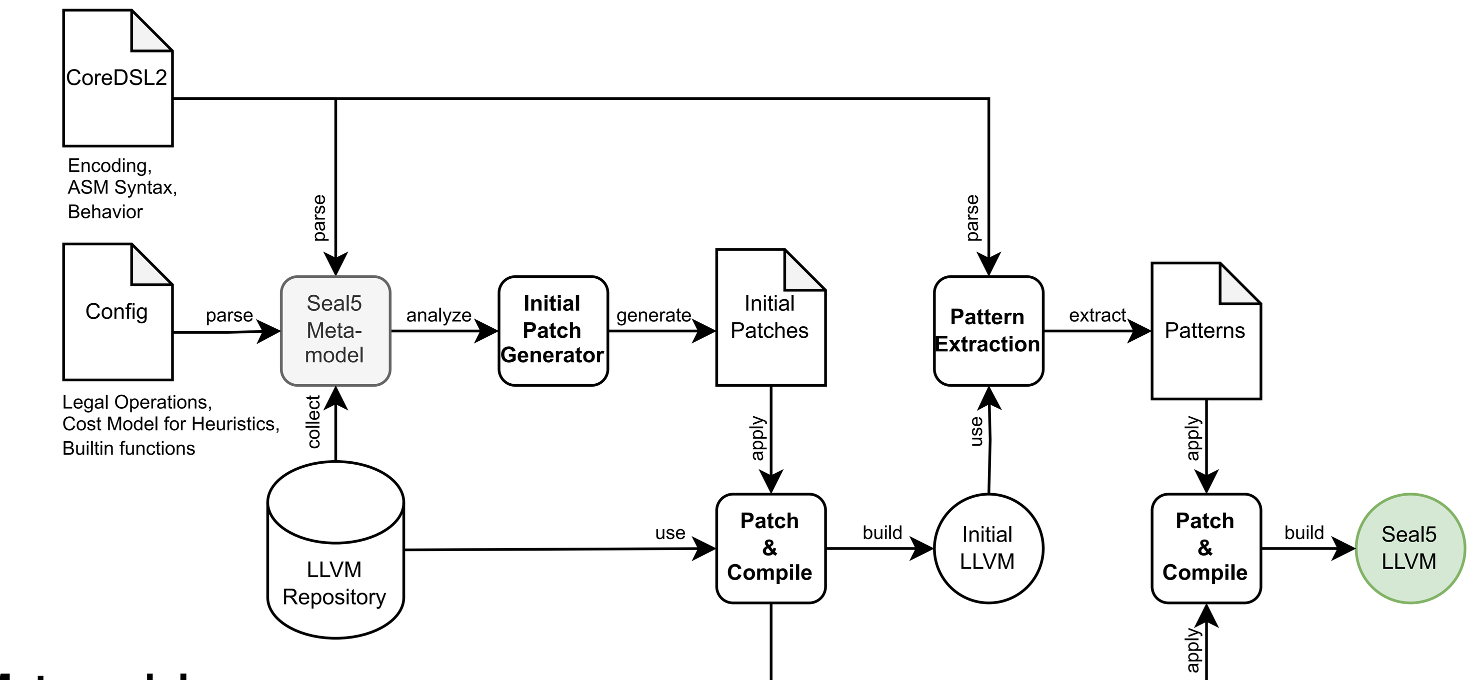
- Generation of test cases
- Support register-pairs (→ LLVM Support for RISC-V Packed Extension)

Planned Features

- Support more datatypes: `float32,float16,int4,...`
- uArch-aware scheduling, ...

Methodology

Flow



Metamodel

- Based on M2-ISA-R [2]
- Extend with Seal5-specific information (Legalization Rules, Costs, Builtins,...)
- Python based Frontend (CoreDSL parser), Transforms (Analysis) and Backends

Intial Patch Generation

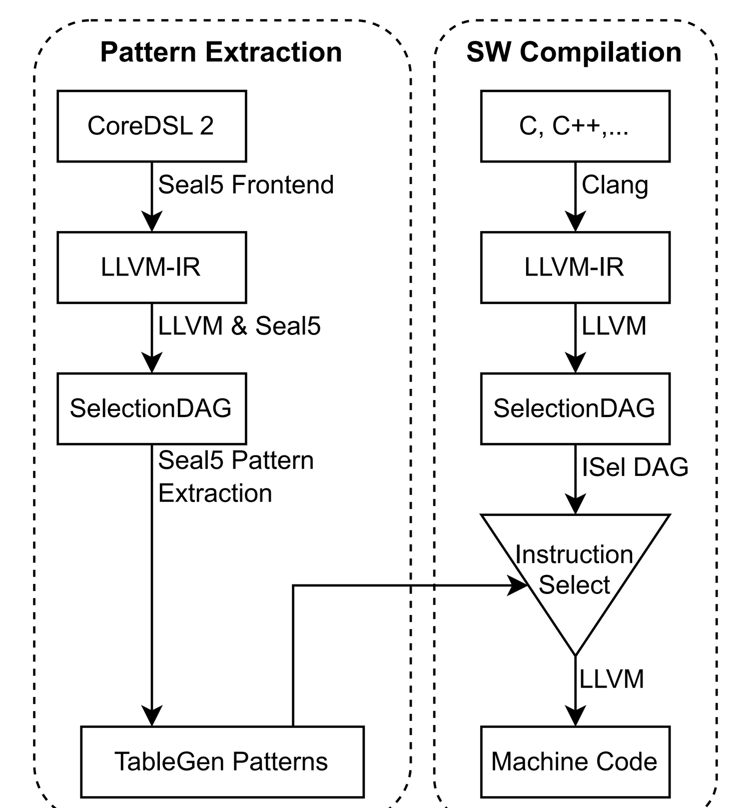
Generates Tablegen and C++ artifacts for:

- Assembly-level support
- Builtins/Intrincsics
- Legalization rules
- Heuristics and cost functions

Extraction of Code-Generation Patterns

How to avoid manual definition of ISel patterns in LLVM?

1. Convert CoreDSL behavior to LLVM-IR functions
2. Perform lowering in a similar way to target SW
3. Add hook to emit final DAG right before Instruction Selection would take place
4. Transform DAG nodes to TableGen code for patterns



Autovectorization support

- While CoreDSL has no notion of vectors, LLVM can detect SIMD instructions automatically to generate patterns using vector-types
- Challenge: Adjusting compiler heuristics and cost functions of existing autovectorizers to generate efficient code for “narrow” SIMD.

Getting Started

1. **Installation** `pip install seal5`
2. **Running Examples** `python3 examples/demo.py`
3. **Read Documentation** `https://seal5.readthedocs.io`
4. **Report Bugs and request features** `https://github.com/tum-ei-eda/seal5/issues`

Usage

Python API

```
seal5_flow = Seal5Flow("llvm-project") → seal5_flow.initialize(...)
    → seal5_flow.setup(...) → seal5_flow.load(["*.core_desc", ...])
    → seal5_flow.transform(...) → seal5_flow.generate(...) → seal5_flow.patch(...)
    → seal5_flow.build(...) → seal5_flow.test(...) → seal5_flow.deploy(...)
    → seal5_flow.export(...) → seal5_flow.cleanup(...)
```

Command-Line Interface

```
seal5 init llvm-project/ → seal5 setup ... → seal5 load *.core_desc *.yml *.test.c
    → seal5 transform ... → seal5 generate ... → seal5 patch ... → seal5 build ...
    → seal5 test ... → seal5 deploy ... → seal5 export ... → seal5 cleanup ...
```

References

[1] CoreDSL: `https://github.com/Minres/CoreDSL/wiki/CoreDSL-2-programmer's-manual`
[2] M2-ISA-R: `https://github.com/tum-ei-eda/M2-ISA-R`
[3] CV32E40P Spec: `https://cv32e40p.readthedocs.io/en/latest/instruction_set_extensions.html`
[4] Core-V LLVM Project: `https://github.com/openhwgroup/corev-llvm-project`
[5] Scalar Cryptography Extension: `https://github.com/riscv/riscv-crypto`

This work has been developed in the project MANNHEIM-FlexKI funded by the German Federal Ministry of Education and Research (BMBF) under contract no.01IS22086L. The authors are responsible for the content of this publication.

SPONSORED BY THE Federal Ministry of Education and Research

Contact:
philipp.van-kempen@tum.de
mathis.salmen@tum.de
daniel.mueller-gritschneder@tuwien.ac.at

Open Source:
https://github.com/tum-ei-eda/seal5