Seal5: Semi-automated LLVM Support for RISC-V ISA Extensions Including Autovectorization

Philipp van Kempen
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
philipp.van-kempen@tum.de

Mathis Salmen
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
mathis.salmen@tum.de

Daniel Mueller-Gritschneder
Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
daniel.mueller-gritschneder@tuwien.ac.at

Ulf Schlichtmann
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
ulf.schlichtmann@tum.de

***Abstract*—The RISC-V instruction set architecture (ISA) is popular for its extensibility, allowing easy integration of custom vendor-defined instructions tailored to specific applications. However, a quick exploration of instruction candidates fails due to the lack of tools to auto-generate embedded software toolchain support. In particular, exploiting SIMD instructions to accelerate typical DSP and machine learning workloads needs specialized integration. This work establishes a semi-automated flow to generate LLVM compiler support for custom instructions based on a C-style ISA description language. The implemented Seal5 tool is capable of generating support for functionalities ranging from baseline assembler-level support, over builtin functions to compiler code generation patterns for scalar as well as vector instructions, while requiring no deeper compiler know-how. This paper focuses primarily on a novel pattern generator approach for the optimized code generation for SIMD instructions, including support for autovectorization. The autogenerated LLVM toolchain reduces development times drastically while performing similarly or better compared to the existing, manually implemented Core-V reference LLVM toolchain on a wide variety of benchmarks. Seal5 further allows the addition of compiler code generation support for the Core-V SIMD instructions, which is not yet available in the reference toolchain. Additionally, Seal5 facilitates a quick exploration of custom instruction candidates as demonstrated for a cryptography extension.**

***Index Terms*—Instruction sets, Compilers, Data processing**

I. Introduction

An increase in computational efficiency for embedded workloads, such as ML, DSP or cryptographic applications, often demands specialized processing cores. The RISC-V instruction set architecture (ISA) allows processors to be tailored for such specialized workloads using so-called vendor-defined custom instruction set extensions. Exploiting and exploring such extensions is often limited by the lack of auto-generated support in simulators and embedded SW toolchains, e.g., LLVM [1] or GCC.

This work has been supported by the project MANNHEIM-FlexKI funded by the German Federal Ministry of Education and Research (BMBF) under contract no.01IS22086L.

Coding well-performing compiler patches for a set of custom instructions requires manual work and often deep compiler expertise, depending on the targeted level of support. Manually implementing assembler support for custom instructions is comparatively easy but work-intensive, while code generation support in the compiler to emit custom instructions and enable code optimizations such as autovectorization for SIMD instructions is fairly complex. Different types of SIMD instructions for RISC-V are defined in two official extensions (Packed and Vector), and OpenHW Group's custom Core-V vendor extension. One common concern for all these extensions is the amount of work required to retarget existing compiler suites such as GCC or LLVM. Implementing LLVM support for the official RISC-V vector extension was a multi-year community effort. For both of the other extensions, development is still ongoing.

To overcome this challenge, this work proposes a new approach to semi-automate the generation of LLVM compiler support for custom RISC-V extensions. The input is a textual C-style ISA description as well as optional user-defined metadata. Based on this input, our proposed Seal5 generator creates a set of LLVM compiler patches for the custom RISC-V instructions in a two-step process. The core contributions are:

- Fully-automated model-based code generation for builtins and assembly-level support for any kind of instructions.
- Semi-automated pattern generator for scalar and SIMD ALU-type instructions exploiting LLVM's frontend and optimization pipeline.
- Autovectorization support for "narrow" (sub-word) SIMD instructions with general purpose registers (GPRs).

First, we evaluate our generator approach on the Core-V instruction set extension. Our reference is the manually patched LLVM toolchain with Core-V support, which is currently being developed by the OpenHW Group community. For our generated LLVM, we see that 46 more unique Core-V instructions are being utilized over a large set of benchmarks.

Additionally, we observe improved average execution performance in profiles on the 32 bit CV32E40P processor. This is feasible, because the Seal5 LLVM exploits the compiler code generation support for Core-V SIMD instructions, which is not yet available in the community LLVM version. The applicability of Seal5 is also demonstrated using custom SHA-256 operations, which can be integrated effortlessly to achieve large performance gains for specific workloads. Most importantly, our generation approach involves minimal development effort and should be less error-prone compared to hand-written patches, as writing low-level assembly code can be avoided. We further generalized existing parts of LLVM's RISC-V backend so far solely used by the vector extension, allowing more types of SIMD instructions to profit from powerful loop level and super-word level parallelism (SLP) vectorization algorithms already available in LLVM.

II. State of the Art

A large number of official and vendor-specific RISC-V extensions have been proposed in recent years with various levels of software toolchain support. For example, new instructions for cryptography algorithms in [2] or trigonometric functions for constrained embedded systems in [3] only have assembler support and are inserted manually into hand-optimized assembly programs. In contrast, for the bitmanipulation-type instructions in [4] or the Core-V extension, which is a successor of the Xpulpnn extension [5], a full retargeting of the RISC-V GNU toolchain or LLVM tools was conducted manually.

Numerous description languages for instruction set extensions have been proposed by industry and academia. The Tensilica Instruction Extension (TIE) maintained by Cadence [6] as well as ARC Processor Extension (APEX) by Synopsys are architectural description languages (ADL) based on the Verilog syntax. Synopsys' nML [7] language aims to offer a higher abstraction level but still requires deep knowledge about the processor's microarchitecture. Codasip's CodAL [8] and the open CoreDSL [9] language use more simple C-like syntax for behavioral descriptions.

In the industry, a set of generation tools for application-specific instruction set processors (ASIPs) seems to have established automatic compiler generation infrastructure. For example, Codasip Studio [10] provides a flexible software development kit (SDK) including an LLVM implementation based on CodAL, which was evaluated on Embench-IoT [11] in terms of performance and code size. According to [12], an extraction of the semantics of the instruction extensions is performed on LLVM's high-level intermediate representation (IR), to generate instruction selection patterns. Synopsys' ASIP Designer suite features a retargetable C/C++ compiler for several ASIP families, including `Trv32p5x` based on the RISC-V ISA extended with SIMD vector processing [13]. For both tools, it is unknown whether support for autovectorization of arbitrary custom SIMD instructions is available and direct comparison is impossible due to their black-box nature and profiling performed on custom-generated HW cores.

Recent publications explore the generation of LLVM patches either using high-level DSLs (CoreDSL) or extensions to LLVM's intermediate representation [14], [15]. To our knowledge, both proposed tools lack support for pattern generation and autovectorization. In [16], an open-source ASIP co-design environment was proposed for a different architecture, including a retargetable LLVM toolchain. Custom operations can be implemented using macro-based C++ files but must be invoked manually using language-specific intrinsics added by the software developer. To enable the automatic selection of custom instructions, a directed acyclic graph (DAG) representation further needs to be provided, which requires deep compiler expertise. [17] discusses the importance of automatic instruction selection and mapping for the automated exploration of ISA extensions. However, The introduced approach for inserting newly added instructions in the targeted program can not be applied for custom operations defined manually or by a third-party application. The performance of autovectorization for RISC-V's vector extension was evaluated in [18], concluding that high-level DSLs may be used to avoid intrinsic-based programming in cases where LLVM's vectorization passes are not successful due to the complex vector-length-agnostic design of the vector ISA. This work does not focus on improving the autovectorizer in LLVM but instead intends to enable the use of existing autovectorization support for a different SIMD extension.

Overall, to our knowledge there is no open, semi-automated compiler retargeting flow for RISC-V available, which also supports compiler code generation and autovectorization, as is addressed by the contributions in this work.

III. Prerequisites

A. Core-V ISA Extensions

TABLE I
Core-V Instructions

Name	Description	Count	Supported	
			Ref.	Ours
XCVMac	16 bit × 16 bit Mul.	8	Y	Y
	16 bit × 16 bit Mul.-Acc.	8	Y	Y
	32 bit × 32 bit Mul.-Acc.	2	Y	Y
XCVMem	Reg.-Imm. & Post-Inc. Ld./St.	8	Y	Y
	Reg.-Reg. & Post-Inc. Ld./St.	8	Y	Y
	Reg.-Reg. Ld./St.	8	Y	Y
XCVAlu	General ALU	32	Y	Y
XCVBitmanip	Bit Manipulation	16	Y[a]	Y[a]
XCVSimd	SIMD ALU	86	Y[a]	Y
	SIMD Bit Manipulation	6	Y[a]	Y
	SIMD Dot Product	36	Y[a]	Y
	SIMD Comparison	59	Y[a]	Y
	SIMD Shuffle and Pack	13	Y[a]	Y[c]
	SIMD Complex-number	19	Y[a]	Y[a]
XCVBi	Immediate Branching	2	Y[b]	N
XCVHwlp	Hardware Loop Setup	8	Y[b]	N
XCVElw	Event Load	1	Y[a]	N

[a] no code-gen. [b] broken or outdated. [c] manual code-gen.

We demonstrate our approach with the Core-V extension consisting of over 300 custom instructions, suitable for various

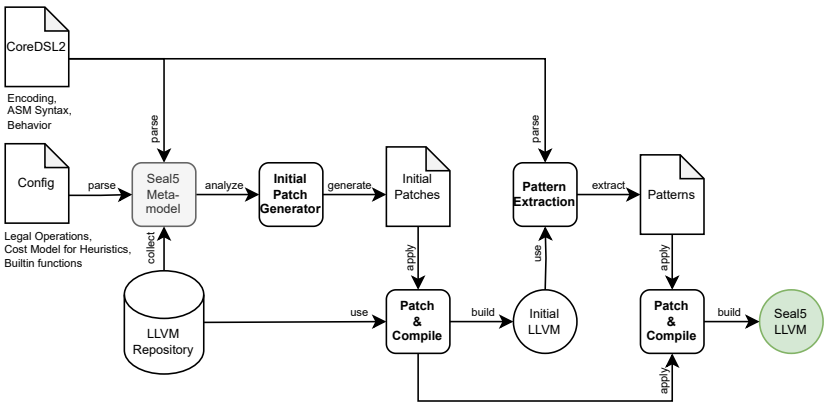


Fig. 1. Complete Seal5 Flow.

embedded applications and developed by the OpenHW Group, given in Table I. Manually implemented Core-V support exists in a reference LLVM toolchain which can be used for direct comparison to our approach. Since some subsets are either unsuitable for automation or not working properly in the reference toolchain, we focus on the XCVMac, XCVMem, XCVAlu and XCVSimd instructions in this paper.

The XCVMac set implements instructions for $16/32\,\text{bit}$ signed and unsigned multiplications with optional accumulation, used a lot by ML/DSP applications, while the XCVMem set extends the RISC-V load and store instructions with support for implicit pointer increments and passing offsets via register-operands. The remaining scalar instructions are either part of XCVAlu or XCVBitmanip. Finally, XCVSimd implements most of the ALU instructions for `int8x4` or `int16x2` vectors using the same 32 general purpose registers (GPRs) used by scalar integer instructions, thus saving hardware resources and costs for data movement between different types of registers. Shuffle and (un-)packing instructions further allow efficient transformations of vectors.

B. LLVM

LLVM is a toolbox of compiler methodologies, capable of compiling almost any programming language to any target architecture. Support for additional languages can be added by creating LLVM frontends, while target architectures are implemented as LLVM backends. In between, an intermediate representation called LLVM-IR is used and optimized using LLVM's target independent optimizer [1]. LLVM was chosen as the target embedded SW compilation toolchain for various reasons, including the availability of the powerful TableGen language, which allows backend developers to extend LLVM without writing complex C++ code, and advanced support for autovectorization compared to the RISC-V GCC toolchain.

IV. Methodology

Figure 1 shows the overall flow of Seal5, which can be summarized as follows:

- First, all available inputs, such as ISA description and user-specified configurations, are collected and processed. The baseline LLVM code base is analyzed and prepared for applying patches. The collected information is stored in the Seal5 metamodel, explained in Subsection IV-B.
- The generation of patches is split into two stages. In the first step, all collected information is transformed into a set of initial C++ patches and TableGen files. The extraction of patterns is handled in a second step due to its dependence on an intermediate LLVM build with the initial patches applied.
- Between the stages, LLVM's build infrastructure is utilized to update LLVM using the generated TableGen files and C++ patches.

A running example of a $16\,\text{bit}$ SIMD instruction `cv.add.h` is used in the following to show the intermediate artifacts being generated by the Seal5 flow.

A. Inputs

We use the open CoreDSL2 [9] ISA description language as the input for our tool. CoreDSL2 is especially suited for our pattern generation approach because it uses C-like code with extensions from the Verilog syntax for behavioral descriptions of instructions, as shown in Listing 1, which we can read using a custom LLVM frontend. Any other ISA description language could also be used easily by model-based conversion to CoreDSL2. The CoreDSL2 files are parsed and converted to the Seal5 metamodel for further processing. Additional configuration for legalization, builtin functions and costs for compiler heuristics can be passed by the user via YAML syntax or CoreDSL attributes to adjust the generation of patches.

Listing 1. $16\,\text{bit}$ SIMD Addition in CoreDSL2

```
CV_ADD_H {
  encoding: 7'b0000000 :: rs2[4:0] :: rs1[4:0] ::
            3'b000 :: rd[4:0] :: 7'b1111011;
  assembly: {"cv.add.h",
             "{name(rd)},{name(rs1)},{name(rs2)}"};
  behavior: if (rd != 0) {
    X[rd][15: 0] = (X[rs1][15: 0] + X[rs2][15: 0])
        [15:0];
    X[rd][31:16] = (X[rs1][31:16] + X[rs2][31:16])
        [15:0];
  }
}
```

B. Overview of the Seal5 Metamodel

The Seal5 tool adapts the metamodel of the M2-ISA-R tool introduced in [19], which already provides a Python-based CodeDSL2 parser and code-generation framework for instruction set simulators. The metamodel has an architectural part, which describes the instruction set structurally (available registers, encodings,...) as well as a behavioral part, which represents the operation of each instruction in a tree-like fashion. Extensions have been made to the original metamodel to incorporate compiler-specific metadata, such as the legalization rules discussed later in Section IV-D.

C. Initial Patch Generation for Assembly and Builtin Support

As a first step, the information on the encoding, operands and assembly syntax of every single instruction stored in the Seal5 metamodel is transformed into TableGen records. Generating these definitions is relatively straightforward as the metamodel is structured to emit this data. To emit valid programs, LLVM's RISC-V backend further needs to know whether an instruction has side effects or involves a load/store as also indicated in [20]. Fortunately, Seal5 can detect most of these properties by traversing the behavioral model of each instruction. While the extensible compiler proposed in [14] generates the same kind of patches, it does not perform any semantic analysis and, therefore, relies on user inputs manually specified alongside the CoreDSL files.

Additionally, our implementation is capable of automatically adding intrinsic functions that are exposed to LLVM's IR as well as the Clang frontend via builtin functions. This allows SW developers to emit custom instructions without using inline-assembly syntax.

D. Initial Patch Generation for Legalization

Before the extraction of patterns can take place, we generate code used during a process called legalization. Legalization allows LLVM to handle incompatible types and operations by removing and replacing them with sequences of supported operations on supported types. This is used for operations such as `ABS`, `MIN` and `MAX`, which do not have a direct counterpart in the RISC-V base instruction set. Further, legalization is especially important for the vector types `v2i16` and `v4i8` utilized by the SIMD instructions. Seal5 determines the legalizer settings in a semi-automated fashion by running basic analysis passes on the instruction behavior and accepting user-defined legalization rules.

E. Pattern Extraction

LLVM usually relies on patterns manually provided by compiler experts to insert custom instructions automatically into arbitrary programs [21]. Instruction selection can be done manually via C++ code or using simple TableGen syntax in a similar fashion to the instructions' definition. During the compilation of LLVM, data structures required for the instruction selection process are generated implicitly using the TableGen utility `llvm-tblgen --gen-dag-isel`.

Seal5 extracts suitable patterns for compiler-level code generation by using a modified LLVM project. The approach is depicted on the left side of Figure 2.

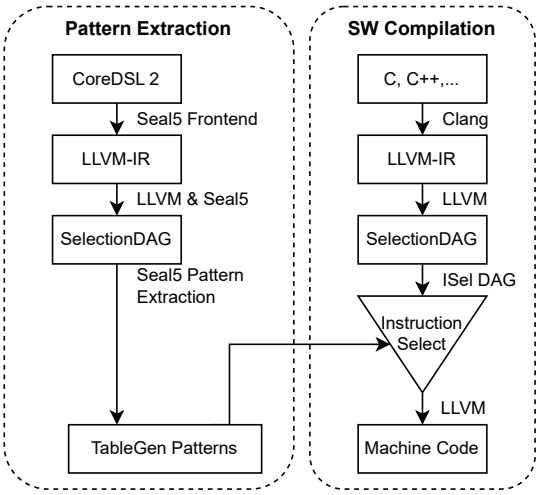


Fig. 2. Pattern extraction approach based on modified LLVM. The left side shows the flow of the custom instruction's behavior, while the right side shows where the patterns are later used during SW compilation in the Seal5 LLVM.

Our approach exploits the C-style syntax used for the instruction's behavioral descriptions in CoreDSL2. We developed a custom frontend (derived from Clang) to transform the C-style behavioral description of each instruction into an LLVM-IR function. Pointers are passed to the function for accessing the source and destination registers as well as immediate operands. This allows for trivial sub-word access via regular load/store operations. LLVM then applies numerous optimizations to simplify the instruction behavior functions.

Listing 2. Optimized LLVM-IR of 16 bit SIMD Addition

```
define void @implCV_ADD_H(ptr nocapture readonly %
    rs1, ptr nocapture readonly %rs2, ptr nocapture
    writeonly %rd) local_unnamed_addr #0 {
  %1 = load <2 x i16>, ptr %rs1, align 2
  %2 = load <2 x i16>, ptr %rs2, align 2
  %3 = add <2 x i16> %2, %1
  store <2 x i16> %3, ptr %rd, align 2
  ret void
}
```

Listing 2 shows the LLVM-IR module after optimization. Next, LLVM legalizes the instruction behavior functions, which must follow the same rules as in the final LLVM build for compatible patterns to be generated. To achieve this, we use an intermediate version of LLVM for this step, which includes all patches except the instruction selection patterns (which are yet to be generated). During compilation, LLVM generates directed acyclic graph (DAG) representations of each function's dataflow. In normal usage of LLVM, instructions are selected by matching their patterns to these DAGs. However, instead of emitting instructions, we intend to generate patterns. For this purpose, we output the DAGs in TableGen syntax for use as patterns. The DAGs can be used as patterns with minimal intervention because we are compiling instruction behavior functions.

To extract instruction selection patterns, Seal5 utilizes an added hook to access SelectionDAGs immediately before ISel would usually occur. Before this hook executes, instruction behavior functions are subject to the exact same code generation pipeline as any other code compiled with Seal5 LLVM. In contrast to [10], which directly maps high-level LLVM-IR nodes to patterns, and [13], operating on a so-called instruction-set graph (ISG), our approach based on SelectionDAG yields patterns that are highly similar to the DAGs LLVM generates during regular usage, thus increasing the likelihood of matches.

After pattern generation completes, generated patterns are added to the intermediate LLVM version, creating the final Seal5 LLVM. The patterns are then used during the compilation of regular code, as shown on the right side of Figure 2. The TableGen pattern for the discussed example is given in Listing 3.

Listing 3. ISel Pattern for 16 bit SIMD Addition

```
def : Pat<
  (add GPR32V2:$rs2, GPR32V2:$rs1),
  (CV_ADD_H GPR32V2:$rs1, GPR32V2:$rs2)>;
```

Due to limitations of SelectionDAG and TableGen, the following conditions must be met by behavior functions (after optimization) for pattern generation to succeed:

- Single basic block only
- Sequence ends with exactly one write to a register (pattern tree may have a single root)

This limits the types of custom instructions for automatic pattern extraction to a subset consisting of mainly memory, ALU and SIMD instructions. However, more complex instruction behavior, for example for custom branch instructions, is supported as well by falling back to a visitor-pattern based approach using the behavioral part of the metamodel.

F. Vectorization

Another aspect of generating SIMD patterns is vectorization. CoreDSL2 behavior has no notion of vectors. Hence, the code for SIMD instructions uses (unrolled) loops performing multiple scalar operations in a linear fashion. This is problematic since instruction selection patterns cannot have more than one result and need to make explicit use of LLVM's vector types. Fortunately, LLVM's autovectorizer is capable of detecting and combining these scalar operations. This allows us to simplify the instruction behavior by using vector types and eliminates multiple outputs by using vectorized memory accesses.

Our implementation is limited to "narrow" SIMD RISC-V instructions operating solely on 32 bit general purpose registers. However, we cannot just use `GPR` operands since these are only meant to hold scalar values of type `i32`. Instead, we generate patches introducing the new register classes `GPR32V2` and `GPR32V4`, as used by the generated pattern in Listing 3. These are defined to hold `v2i16` or `v4i8` vectors, being otherwise identical to regular `GPR`s. In contrast to the vector registers used by RISC-V's official vector extension, type casts between `GPR32V2`/`GPR32V4` and `GPR` are free, eliminating data movement penalties. Additionally, memory operations involving these SIMD types can be handled using the regular `i32` load and store instructions. For supporting SIMD comparisons, the legalizer settings are updated to promote types such as `v2i1` and `v4i1` to `v2i16`/`v4i8`.

G. Heuristics

Heuristic functions and cost functions are used throughout the LLVM compilation pipeline. If SIMD instructions are used, it is especially important to assist LLVM's vectorizers (loop and basic block) in making meaningful decisions to unroll or interleave vectorized loops. Default SIMD-related settings are tailored to powerful vector instruction sets such as the RISC-V vector extension or X86's AVX512. Cost functions also estimate whether a specific operation, e.g. strided or unaligned memory access, is costly on a specific target device and should be avoided. Of particular importance for XCVSimd is avoiding the vectorization of pointers, since these cannot be represented in 32 bit vectors. A significant performance loss was observed without matching heuristics, as LLVM would split vectors into redundant scalar operations late. Hence, a cost description must be provided as another input to the Seal5 generator flow. We also observed synergies when enabling the post-incrementing addressing mode for the XCVMem register-register load/store instructions, leading to less boilerplate code for pointer updates in tight vectorized loops.

H. Custom Modifications

Some very specific code changes cannot be generated automatically. For the Core-V extensions, this includes the custom instruction selection of the shuffle and pack instructions mentioned in Table I. Support for hardware loops and immediate branching instructions, which are not covered here, would require changes in the RISC-V backend, including the addition of custom passes. Adding more elaborate patterns to increase the number of emitted custom instructions for specific operations is also possible. The number of manually added lines of code (LOCs) is around 200 mostly reusing existing code to support the RISC-V vector extension in LLVM.

V. Experimental Results

A. Experimental Setup

Our Seal5 approach is evaluated using three different LLVM toolchain versions:

1) **Baseline:** Upstream LLVM project compiling for `RV32IMC` without Core-V support.
2) **Reference:** Core-V fork of LLVM project developed by the OpenHW Group community, used to compile for the supported subset of extensions.
3) **Generated Seal5 (Ours):** Our automatically generated LLVM targeting the same Core-V extensions
   a) without SIMD code generation or
   b) including autovectorization (SIMD) support.

TABLE II
USED BENCHMARKS

Group	Workloads	Configurations	Total
MLPerfTiny [22] (MicroTVM)	aww vww resnet toycar	2 Backends 1-3 Layouts AutoTVM / no tuning	40
Embench-IoT [11]	various benchmarks	-	22
Coremark	single benchmark	100 Runs	1
Synthetic (Ours)	SIMD benchmarks	-	17

For obtaining realistic performance cycle counts, a verilated RTL model of the CV32E40P core implementing all Core-V instructions was used.

An extensive set of workloads given in Table II, both from TinyML applications and general embedded SW benchmarks, is used to evaluate the different LLVM compilers. It should be noted that with few exceptions, most workloads were not conceived for optimized SIMD processing. While most of them are applicable for general ALU, multiply-accumulate and memory instructions, about 80% of them do not support vectorization due to the use of types larger than $16\,\mathrm{bit}$ or floating point operations. For all programs for which checksums are available to verify the correctness of the generated output, we successfully verified that the generated Seal5 LLVM compiler toolchain provides correct outputs.

B. Evaluation of the Utilization of Custom Core-V Instructions

A static analysis of the generated programs is performed to identify the number of unique Core-V extension instructions that were automatically inserted by the compiler. A high utilization is desirable to show that the instruction selection patterns work as expected and the compiler makes use of custom instructions.

The results of the reference LLVM toolchain are given in Table III. The relative utilization obtained by comparing the counts with the total number of unique instructions in Table I yields that, for XCVMem, 50% or more unique instructions are consistently emitted by the reference compiler. Utilization of XCVMac and XCVAlu instructions are 17% and 75%, respectively. SIMD instructions are not generated at all due to missing support in the reference LLVM.

In comparison, the utilization data for the generated Seal5 toolchain is given in Table IV. For the Seal5 generated LLVM, the values for XCVMac and XCVMem largely match the ones obtained using the reference toolchain. For XCVAlu, nine fewer instructions are emitted automatically. However, this can mainly be explained by scalar ALU instructions being replaced with their SIMD counterpart whenever possible. Finally, a larger number of 55 XCVSimd instructions are emitted, leading to a total utilization of 32% or 89 unique Core-V instructions for Seal5, compared to 16% or 43 instructions for the reference Core-V LLVM. The compilation time of the generated LLVM, as well as the reference one, is slightly higher compared to the baseline LLVM due to the higher number of available instructions and patterns.

TABLE III
UTILIZATION OF UNIQUE CORE-V INSTRUCTIONS BY REFERENCE CORE-V LLVM

Group	**Utilization** (unique instructions)				
	Mac	Mem	Alu	Simd	Total
MLPerfTiny	2 (11%)	14 (58%)	15 (47%)	0 (0%)	31 (11%)
Embench-IoT	2 (11%)	14 (58%)	18 (56%)	0 (0%)	34 (12%)
TACLeBench	3 (17%)	15 (62%)	21 (66%)	0 (0%)	39 (14%)
CoreMark	1 (6%)	13 (54%)	5 (16%)	0 (0%)	19 (7%)
Synthetic	1 (6%)	12 (50%)	1 (3%)	0 (0%)	14 (5%)
Total	3 (17%)	16 (67%)	24 (75%)	0 (0%)	43 (16%)

TABLE IV
UTILIZATION OF UNIQUE CORE-V INSTRUCTIONS BY SEAL5 GENERATED LLVM (OURS)

Group	**Utilization** (unique instructions)				
	Mac	Mem	Alu	Simd	Total
MLPerfTiny	2 (11%)	12 (50%)	12 (38%)	11 (6%)	37 (14%)
Embench-IoT	2 (11%)	13 (54%)	15 (47%)	21 (10%)	51 (19%)
TACLeBench	3 (17%)	15 (62%)	15 (47%)	42 (21%)	75 (27%)
CoreMark	1 (6%)	14 (58%)	5 (16%)	5 (2%)	25 (9%)
Synthetic	1 (6%)	14 (58%)	1 (3%)	16 (8%)	32 (12%)
Total	3 (17%)	16 (67%)	15 (47%)	55 (28%)	89 (32%)

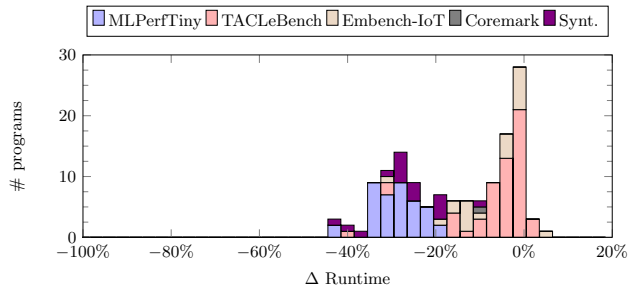
C. Evaluation of the Core-V Execution Runtime

Based on cycle counts of the RTL-level simulation of the CV32E40P core, we evaluate the runtime gain obtained from the compiler using the Core-V extension over all groups of benchmark programs. We use the baseline LLVM version without Core-V support as the baseline runtime.

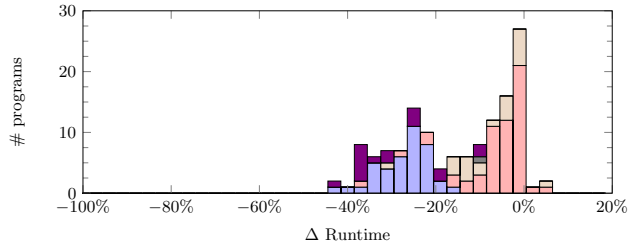
Figure 3(a) shows that the reference Core-V toolchain can reduce the runtime of many benchmark programs by up to -42% compared to the runtime without the Core-V extension. For a large set of programs, no performance gain is achieved (runtime reduction 0%), mainly due to the use of unsupported data types. For a few programs, we see slight runtime overheads. An average runtime improvement of -15% is reached over all benchmark programs.

For the generated Seal5 toolchain, the results are depicted separately in Figures 3(b) and 3(c). For the non-SIMD case, the plot looks comparable to the reference one in Figure 3(a) averaging at a runtime improvement of -14%. For the SIMD case, superior runtime reductions are obtained, up to -86% ($7\times$ speedup) for selected MLPerfTiny benchmarks. With SIMD support, an average runtime improvement of -24% is reached over all benchmarks.

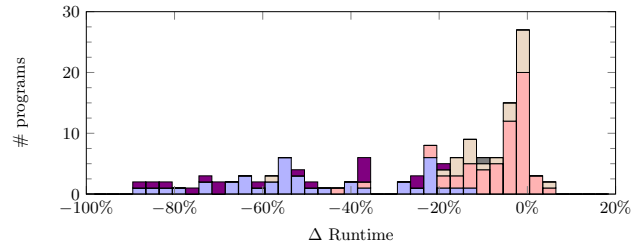
Evidently, the generated Seal5 LLVM toolchain results in competitive results compared to the available Core-V reference



(a) Reference LLVM, Without SIMD



(b) Seal5 LLVM (Ours), Without SIMD



(c) Seal5 LLVM (Ours), With SIMD

Fig. 3. Histogram of the reduction of all benchmark program's runtime in cycles measured on the CV32E40P core. Baseline is the runtime of the program compiled without any Core-V extension support. Δ Runtime smaller 0% indicates that the LLVM with Core-V extensions could improve runtime compared to a program with just standard RISC-V instructions. Programs are grouped by benchmark (colors) into runtime bins.

toolchain. With the availability of SIMD support, Seal5 can clearly outperform the reference toolchain.

In real-world usage, further runtime improvements may even be achieved by adapting programs to the available SIMD support.

D. Autovectorization Analysis

To demonstrate this, the last set of benchmarks consists of 17 hand-written synthetic workloads chosen to be trivially vectorizable using LLVM's loop vectorizer. Hence, any missed opportunities for autovectorization are likely due to limitations of the ISA or microarchitecture itself, such as missing support for strided memory loads.

The results are shown in Table V for 8 and 16 bit sized data. Most benchmarks operating on flat vectors of length N (instead of $\sqrt{N} \times \sqrt{N}$ matrices) achieve the expected $2\times$ or $4\times$ speedup factors. For variations of (transposed) matrix-matrix multiplications, the layout of the underlying data as well as the order of loops becomes quite relevant. Only for even unstrided memory access patterns, where multiple data items can be fetched from memory in parallel, the theoretical speedups can be obtained.

TABLE V
PERFORMANCE ON SYNTHETIC SIMD WORKLOADS ($N = 1024$)

Workload	Bits		Runtime (Cycles)		%
	I	O	Baseline RV32IMC	Seal5 LLVM with SIMD	Δ
to_upper	8	8	11350	2903	−74.4%
add	8	8	13392	2141	−84.0%
$(c = x + y)$	16	16	12372	3936	−68.2%
dotp	8	32	12368	1758	−85.8%
$(c = x^T y)$	16	32	11348	3168	−72.1%
saxpy	8	8	13393	5227	−61.0%
$(s = \alpha x + y)$	16	16	12373	10324	−16.6%
matmul	8	8	369939	238866	−35.4%
$(\mathbf{C} = \mathbf{AB})$	16	16	371991	240918	−35.2%
matmul[a]	8	8	367891	196038	−46.7%
$(\mathbf{C} = \mathbf{AB})$	16	16	402711	262568	−34.8%
tr. matmul	8	8	400658	194067	−51.6%
$(\mathbf{C} = \mathbf{AB}^T)$	16	16	371991	240918	−35.2%
tr. matmul[a]	8	8	400659	302354	−24.5%
$(\mathbf{C} = \mathbf{AB}^T)$	16	16	402711	304402	−24.4%
tr. matmul[b]	8	8	368914	68075	−81.5%
$(\mathbf{C} = \mathbf{AB}^T)$	16	16	371987	117230	−68.5%

[a] reordered loop layout. [b] explicit call to dotp.

Coming up with optimal code for these kinds of operations is crucial to avoid performance degradations. LLVM's autovectorizer is capable of accelerating the workloads by exploiting Core-V's SIMD instructions using the patterns emitted by Seal5.

E. Extending Seal5 LLVM with Custom Instructions

In the previous sections, Seal5 applicability was successfully demonstrated using the Core-V instruction set. As Seal5's compatibility is not limited to the Core-V instructions, anyone may add custom instructions targeting specific types of workloads in a straightforward fashion.

$$\mathrm{rotr}^n(x) := (x >> n) \mid (x << (32 - n)) \tag{1}$$

$$\begin{aligned}
\mathrm{sha256sig}_0(x) &:= \mathrm{rotr}^7(x) \oplus \mathrm{rotr}^{18}(x) \oplus (x \gg 3) \\
\mathrm{sha256sig}_1(x) &:= \mathrm{rotr}^{17}(x) \oplus \mathrm{rotr}^{19}(x) \oplus (x \gg 10) \\
\mathrm{sha256sum}_0(x) &:= \mathrm{rotr}^2(x) \oplus \mathrm{rotr}^{13}(x) \oplus \mathrm{rotl}^{22}(x) \\
\mathrm{sha256sum}_1(x) &:= \mathrm{rotr}^6(x) \oplus \mathrm{rotr}^{11}(x) \oplus \mathrm{rotl}^{25}(x)
\end{aligned} \tag{2}$$

Equation 1 defines a simple rotation instruction, while Equation 2 shows four custom operations for the SHA-256 secure hash algorithm as proposed in [23], which both have been implemented in the CoreDSL syntax and loaded into the Seal5 tool. Without further manual inputs, the generated LLVM toolchain is capable of inserting the custom instructions thanks to the automatically generated patterns such as given in Listing 4.

Listing 4. Generated pattern for sha256sig_0 operation

```
def : Pat<
  (xor (xor (rotl GPR:$rs1, (i32 25)), (rotl GPR:
      $rs1, (i32 14))), (i32 (srl GPR:$rs1, (i32 3))
      )),
  (SHA256SIG0 GPR:$rs1)>;
```

The observed runtime improvements can be demonstrated with the unmodified `nettle-sha256` benchmark from the Embench-IoT suite. The number of instructions (simulated at instruction-level) to run the program, as given in Figure 4, can be reduced by 52% when utilizing the SHA-256 instructions, matching the observations previously discussed in [16]. In comparison, the single rotation instruction only yields a speedup of 15%.

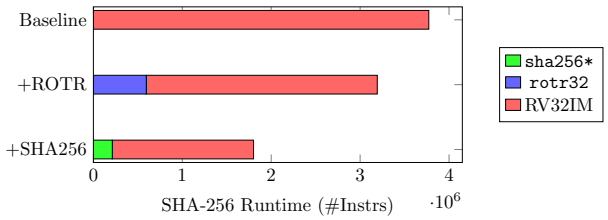


Fig. 4. Runtime of `nettle-sha256` benchmark compiled by Seal5 LLVM using either no extensions, 32 bit rotations or the custom SHA-256 instructions. Colors indicate the percentage of dynamic instruction counts grouped by instruction type.

VI. Conclusion

In this work, the Seal5 flow for automatically generating LLVM toolchain patches based on high-level ISA descriptions was proposed. For the considered Core-V instruction sets, we can not only utilize more custom instructions compared to the manually implemented reference toolchain but also reduce the execution time significantly for numerous workloads. The most significant performance improvements are due to new support for "narrow" SIMD instructions. This work opens the door for efficient exploration of custom ISA extensions in the future. Moreover, using the utilization reports, one could identify custom instructions successfully utilized by the compiler. Finally, we demonstrated the usability of Seal5 in Section V-E by evaluating a set of custom instructions from literature effortlessly, requiring minimal prior knowledge of LLVM, which helps with fast prototyping in the CoreDSL2 and RISC-V ecosystem. The Seal5 tool[1], as well as the generated Core-V LLVM,[2] are open source and available on GitHub.

References

[1] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[2] H. Cheng, J. Großschädl, B. Marshall, D. Page, and T. Pham, "Risc-v instruction set extensions for lightweight symmetric cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 193–237, 2023.

[3] Z. Gao, L. Zhao, and H. Chen, "A trigonometric function instruction set extension method based on risc-v," in *2022 IEEE/ACIS 22nd International Conference on Computer and Information Science (ICIS)*. IEEE, 2022, pp. 119–126.

[4] B. Koppelmann, P. Adelt, W. Mueller, and C. Scheytt, "Risc-v extensions for bit manipulation instructions," in *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2019, pp. 41–48.

[5] A. Garofalo, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "Xpulpnn: Enabling energy efficient and flexible inference of quantized neural networks on risc-v based iot end nodes," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1489–1505, 2021.

[6] H. A. Sanghavi and N. B. Andrews, "Tie: An adl for designing application-specific instruction set extensions," in *Processor Description Languages*. Elsevier, 2008, pp. 183–216.

[7] J. Van Praet, D. Lanneer, W. Geurts, and G. Goossens, "nml: A structural processor modeling language for retargetable compilation and asip design," in *Processor Description Languages*. Elsevier, 2008, pp. 65–93.

[8] Codasip, "What is codal?" 2021. [Online]. Available: https://codasip.com/2021/02/26/what-is-codal/

[9] W. Ecker, P. Adelt, W. Mueller, R. Heckmann, M. Krstic, V. Herdt, R. Drechsler *et al.*, "The scale4edge risc-v ecosystem," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 808–813.

[10] Codasip, "Re-targetable llvm c/c++ compiler for risc-v," 2023. [Online]. Available: https://codasip.com/2023/07/25/re-targetable-llvm-c-c-plus-plus-compiler-for-riscv/

[11] D. Patterson, J. Bennett, C. G. P. Dabbelt, G. Madhusudan, and T. Mudge, "Embench: Open benchmarks for embedded platforms," 2023.

[12] Codasip, "Enhanced llvm support for risc-v," 2019. [Online]. Available: https://riscv.org/wp-content/uploads/2019/03/12.15-Zdnek-Prikryl-Enhanced-LLVM-Support-For-RISC-V.pdf

[13] Synopsys, "Asip designer: Application-specific processor design made easy." [Online]. Available: https://www.synopsys.com/dw/doc.php/ds/cc/asip-brochure.pdf

[14] J. Schlamelcher, T. Goodfellow, B. Kebianyor, and K. Grüttner, "Extending clang/llvm with custom instructions using tablegen – an experience report," in *Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2024, 27th Workshop*, 2024.

[15] M. Halkenhäuser, "Automatic compiler support for application-specific instruction set architecture extensions," Master's thesis, Technische Universität Darmstadt, 2022.

[16] K. Hepola, J. Multanen, and P. Jääskeläinen, "Openasip 2.0: co-design toolset for risc-v application-specific instruction-set processors," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2022, pp. 161–165.

[17] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, "Automatic instruction set extension and utilization for embedded processors," in *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. ASAP 2003*. IEEE, 2003, pp. 108–118.

[18] N. Adit and A. Sampson, "Performance left on the table: An evaluation of compiler autovectorization for risc-v," *IEEE Micro*, vol. 42, no. 5, pp. 41–48, 2022.

[19] J. Kappes, R. Kunzelmann, K. Emrich, C. Foik, D. Mueller-Gritschneder, and W. Ecker, "Effective processor model generation from instruction set simulator to hardware design," in *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, 2023, pp. 1–7.

[20] R. Leupers and P. Marwedel, "Instruction-set modelling for asip code generation," in *Proceedings of 9th International Conference on VLSI Design*. IEEE, 1996, pp. 77–80.

[21] G. H. Blindell, "Instruction selection," *Principles, Methods, and Applications*, 2016.

[22] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau *et al.*, "Mlperf tiny benchmark," *arXiv preprint arXiv:2106.07597*, 2021.

[23] "Risc-v scalar cryptographic extension v1.0.0," 2009. [Online]. Available: https://github.com/riscv/riscv-crypto/releases/tag/v1.0.0-scalar

[1] https://github.com/PhilippvK/corev-seal5

[2] https://github.com/PhilippvK/corev-llvm-project/tree/seal5_paper