TU WIEN Informatics

# Optimizing the OpenVADL compiler generator

BACHELORARBEIT

zur Erlangung des akademischen Grades

**Bachelor of Science**

im Rahmen des Studiums

**Informatik**

eingereicht von

**Johannes Gisy**
Matrikelnummer 12319605

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.i.R. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 5. Mai 2026

| Johannes Gisy | Andreas Krall |
|---|---|

TU WIEN Informatics

# Optimizing the OpenVADL compiler generator

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Science**

in

**Informatics**

by

**Johannes Gisy**
Registration Number 12319605

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof.i.R. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, May 5, 2026

| Johannes Gisy | Andreas Krall |
|---|---|

# Erklärung zur Verfassung der Arbeit

Johannes Gisy

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 5. Mai 2026

Johannes Gisy

# Danksagung

Ich bedanke mich bei Prof. Andreas Krall für sein Feedback und hervorragende Kommunikation währed der Arbeit. Außerdem bedanke ich mich bei Kevin Per, auf dessen vorrausgegangene Arbeit im OpenVADL Team diese Arbeit aufbaut. Ebenfalls bedanke ich mich bei Tobias Weiss, dessen parallele Arbeit am Compiler ich nutzen konnte.

# Acknowledgements

I would like to thank Prof. Andreas Krall for his feedback, and excellent communication throughout this thesis. I would also like to thank Kevin Per, whose previous work on the OpenVADL team served as the foundation for this project as well as Tobias Weiss, whose simultaneous work on the compiler I was able to use for this thesis.

# Kurzfassung

Einen Prozessor zu entwickeln erfordert viele Werkzeuge die zusammen funktionieren müssen. Zu diesen gehören unter anderem ein Compiler, Simulator, Assembler und Linker. Diese Werkzeuge haben eine starke Abhängigkeit zu der Prozessorentwicklung und werden daher simultan entwickelt. Die Vienna Architecture Description Language (VADL) versucht genau dieses Problem zu lösen in dem es eine *single source of truth* darstellt woraus diese Werkzeuge automatisch erzeugt werden können. In dieser Arbeit wird der auf LLVM basierte Compiler Generator mit existierenden Implementierungen für die `RV32IM` und `RV64IM` Architekturen verglichen. Des weiteren werden momentane Probleme identifiziert und mögliche Lösungen diskutiert und implementiert. Die Ergebnisse zeigen, dass kleine Details in der LLVM Target Specification große Auswirkungen auf die Leistung haben können.

# Abstract

To develop a processor one needs several tools that play together such as a compiler, simulator, assembler and linker. These need to be developed alongside the processor creating a strong dependence on each other. This is the problem that the Vienna Architecture Description Language (VADL) is trying to tackle by providing the ability to generate such tools from a *single source of truth*. In this thesis, the current LLVM based compiler generator is evaluated against existing implementations for the `RV32IM` and `RV64IM` architectures. Furthermore current issues are identified and possible fixes discussed and implemented. The results show that improving small details in the LLVM target specification can lead to quite big performance improvements.

# Contents

CHAPTER 1

# Introduction

Designing and creating a processor is a very laborious and costly task, especially in the realm of embedded computing. One needs multiple tools that all play together such as a compiler, simulator, assembler and linker. The goal of the Vienna Architecture Description Language (VADL) is to facilitate rapid design space exploration (DSE) by generating these tools from a specification which acts as the *source of truth* [FHH+25].

OpenVADL[1] is an open source implementation of VADL which aims to improve the shortcomings of the original implementation [FHH+25]. In Chapter 2 we will take a closer look at LLVM, the LLVM Compiler Backend Generator (LCB) of OpenVADL and the challenges of creating a LLVM target. Furthermore, we will present current open problems with specifying the RISC–V Instruction Set Architecture (ISA) in VADL. The next sections will cover optimizations done in OpenVADL to overcome some of these problems and discuss the performance improvements. This thesis builds on the work of Kevin Per's thesis 'LLVM Compiler Generator in OpenVADL' who implemented the LCB for OpenVADL [Per25].

[1] OpenVADL is developed at `https://github.com/OpenVADL/openvadl`

CHAPTER 2

# Background and Related Work

This section serves as an introduction to LLVM and VADL in general.

## 2.1 LLVM Compiler Backend

Modern compilers are usually split into many interchangeable components. While there are many moving parts, LLVM consists of three main modules: *frontend, optimizer, backend*.

**Frontend** Parses and validates the input code in a input language. The parsed code is then translated into an Intermediate Representation (IR) in case of LLVM into LLVM Intermediate Representation (LLVM IR).

**Optimizer** Feeds the IR through a series of input language and target machine independent analysis and optimization passes.

**Backend** Takes the optimized IR and produces assembly or machine code for a given target.

### 2.1.1 LLVM Intermediate Representation

LLVM uses its own IR called LLVM IR to represent a program. It comes in three different forms: in-memory, by a compiler, as bit code, to store on disk, and assembly like for human legibility. LLVM IR is statically typed and in Static Single Assignment (SSA) form [Prob, RT22]. To facilitate faster translations it is split into modules that act as compilation units. When a module is compiled it is first checked for well-formedness. This guarantees that it contains no type errors or SSA violations.

The type system supports a few different classes of types. These include function, single value, like integer or floating point, pointer, vector, structure, label and metadata types.

LLVM IR, like a regular assembly language, does not provide a construct for loops and uses labels with jumps instead. This creates the need for so called *phi-nodes* as LLVM IR is in SSA, where each value can only be assigned once. These *phi-nodes* take a different value depending on the label from which a *phi-node* was reached. This allows to initialize a variable when entering a loop construct and incrementing it when repeating the loop. Listing 2.1.1 shows an example with a simple function that takes a pointer `ptr %a` and the size as an 64–bit integer `i64 %n` and returns the sum of all elements in the array. On line 7 we can see the *phi-node* setting `%sum.06` to `%add` or initializing it to `0` when reached via `entry:`.

```
1  define i64 @sum(ptr %a, i64 %n) {
2  entry:
3    %cmp4 = icmp sgt i64 %n, 0
4    br i1 %cmp4, label %while.body, label %while.end
5
6  while.body:
7    %sum.06 = phi i64 [ %add, %while.body ], [ 0, %entry ]
8    %i.05 = phi i64 [ %inc, %while.body ], [ 0, %entry ]
9    %arrayidx = getelementptr inbounds nuw i64, ptr %a, i64 %i.05
10   %0 = load i64, ptr %arrayidx, align 8
11   %add = add nsw i64 %0, %sum.06
12   %inc = add nuw nsw i64 %i.05, 1
13   %exitcond.not = icmp eq i64 %inc, %n
14   br i1 %exitcond.not, label %while.end, label %while.body
15
16 while.end:
17   %sum.0.lcssa = phi i64 [ 0, %entry ], [ %add, %while.body ]
18   ret i64 %sum.0.lcssa
```

Listing 2.1: Sum of array in LLVM IR.

The LLVM instruction **getelementptr**, which computes the address of an element, in line 9 will become of interest in Section 3.1.

### 2.1.2 Target independent optimizations

Some compiler optimizations can be performed regardless of target architecture. These are so called *target independent optimizations*. In case of LLVM these operate on the LLVM IR.

#### Induction Variables

Induction Variables are variables whose values are only ever changed by some constant. Such variables usually occur in loops as some sort of index or counter [Muc98, p. 427]. LLVM has a dedicated pass to analyze and canonicalize induction variables, which can be used by subsequent passes to perform other optimizations [Proe].

#### Loop Strength Reduction

*Strength reduction* in compiler design refers to replacing expensive computations by less expensive ones. Usually this is done by transforming multiplications into additions or shifts. This technique can also be applied in loops with array accesses where either the index or base may be an induction variable. Here this access can be transformed by eliminating the index and adding the constant increment to the pointer instead [Muc98, p. 435][Prof].

#### Loop Unrolling

Every time a loop iteration is executed, the loop terminating condition is checked. This leads to significant overhead, especially if the iteration count is known at compile time. Instead, the compiler can reduce or outright eliminate the loop condition check by duplicating the loop body multiple times [Muc98, p. 559].

### 2.1.3 Instruction Selection

The final aim of a compiler is to emit machine instructions. This requires transforming the IR into assembly or machine code that can be executed on a given target. While doing so, the compiler has to obey a few rules. The input and output programs have to be semantically equivalent and the compiler can only choose instructions present in the ISA of the target machine.

Instruction selection happens in the *backend* of a compiler. LLVM has three different heuristics for instruction selection. These make a tradeoff between compilation speed and optimality. The fastest selector is Fast Selection DAG Instruction Selection (FastISel), which generates poor code and does not support illegal types. FastISel is embedded in the oldest instruction selector Selection DAG Instruction Selection (SDISel) which is based on finding coverings of Directed Acyclic Graph (DAG)s. The third and newest option is Global Selection Dag Instruction Selection (GlobalISel). It allows matching an entire function body and does not require a dedicated IR which increases compilation speed [Proa] although at the time of writing the code quality is not yet on par with SDISel. The LCB of OpenVADL uses SDISel for instruction selection.

#### Selection DAG Instruction Selection

SDISel operates on DAGs of basic blocks of a function. The DAG used consists nodes specific to SDISel, so called Selection Directed Acyclic Graph Node (SDNode). These first get lowered to target specific nodes and subsequently transformed into machine instruction nodes via the help of TableGen. This is explained in more detail in the TableGen section. It performes an iterative process to legalize and optimize the DAG for the given target. Figure 2.1 shows this process and its components.

**SelectionDAG Construction** Constructs the SDISel specific DAG from LLVM IR for each basic block.

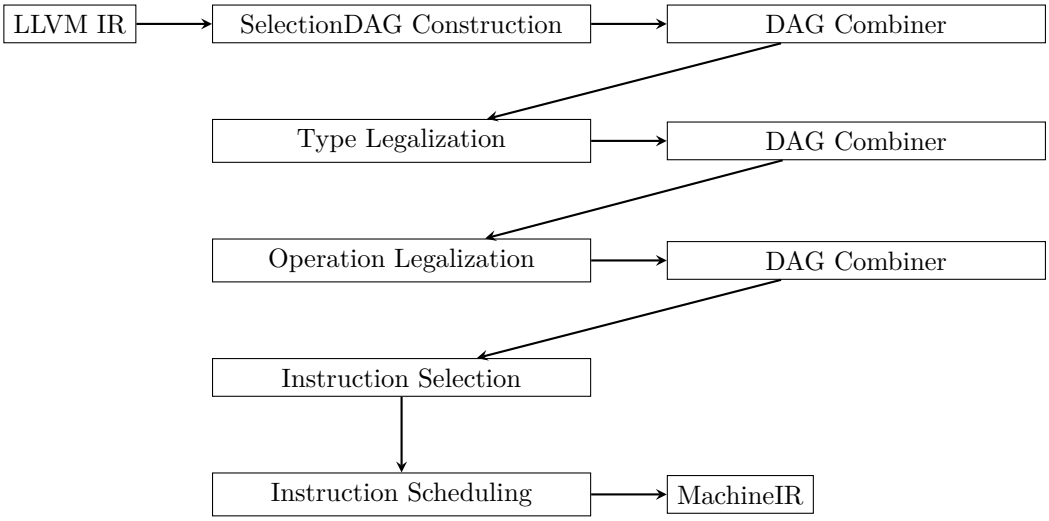


Figure 2.1: SDISel compilation flow, adapted from [FM].

| Action | Description |
|---|---|
| `Legal` | The target natively supports this operation. |
| `Promote` | This operation should be executed in a larger type. |
| `Expand` | Try to expand this to other ops, otherwise use a libcall. |
| `LibCall` | Don't try to expand this to other ops, always use a libcall. |
| `Custom` | Use the `LowerOperation` hook to implement custom lowering. |

Table 2.1: Supported operation legalization actions.

**DAG Combiner** Performs peephole optimizations to iron out any inefficiencies that the previous pass might have accidentally introduced. These transformations are generally target independent.

**Type Legalization** Lowers the types present in the DAG to legal types of the target.

**Operation Legalization** Lowers the operations with specific types to operations and types that are legal on the target. A target can choose between five different actions LLVM should perform. See Table 2.1.

**Instruction Selection** Selects the instructions based on the provided TableGen patterns and custom matching code. This transforms the SDNodes to machine IR.

**Instruction Scheduling** Schedules the selected machine instructions.

### TableGen

Each instruction of an ISA has a specific behavior. Furthermore, it matches some behavior that is semantically equivalent to some nodes in the SDISel. This requires a lot

of repetitive and error prone work to define manually, therefore LLVM uses a Domain Specific Language (DSL), TableGen, to generate the matching code for instructions.

TableGen consists of two primary items, *classes* and *records*. Classes, defined with the keyword **class**, serve as blueprints for **record**s, which are instantiations of it. Classes can define entries in a key–value format, which an record inherits and can choose to override. TableGen is statically typed and also supports `dag` and `list` among common primitive types [Prog].

Listing 2.2 shows an example for the RISC–V `ADDI` instruction used in OpenVADL.

```
// Defines a parameterized blueprint for the ADDI immS
// immediate value.
class RV3264Base_ADDI_immS<ValueType ty> : Operand<ty>
{
    // C++ method to encode the value into the
    // binary representation.
    let EncoderMethod = "RV3264Base_ADDI_wrapper";
    // C++ method to decode the value from the
    // binary representation.
    let DecoderMethod = "RV3264Base_ADDI_immS_decode_wrapper";
}

// Concrete instance of ADDI immS with a predicate function.
def RV3264Base_ADDI_immSAsInt64
: RV3264Base_ADDI_immS<i64>
, ImmLeaf<i64, [{ return RV3264Base_ADDI_immS_predicate(Imm); }]>;

// Concrete machine instruction for addition with register
// and immediate.
def ADDI : Instruction
{
// Output operands of the machine instruction.
let OutOperandList = ( outs X:$rd );
// Input operands of the machine instruction.
let InOperandList =
( ins X:$rs1, RV3264Base_ADDI_immSAsInt64:$immS );
// Encodings of the binary representation.
field bits<32> Inst;
bits<7> opcode = 0b0010011;
bits<3> funct3 = 0b000;
bits<64> immS;
bits<64> rs1;
bits<64> rd;
// Mappings of the instruction's operand into the binary
// representation.
let Inst{31-20} = immS{11-0};
let Inst{19-15} = rs1{4-0};
# ...
// Machine instruction flags that are used in the
// target-independent optimizations.
let isTerminator = 0;
```

```
42 let isBranch = 0;
43 # ...
44 // Hardware registers that are being read from by the instruction.
45 let Uses = [ ];
46 // Hardware registers that are overwritten by the instruction.
47 let Defs = [ ];
48 }
```

Listing 2.2: TableGen Instruction Definition for RV64I `ADDI`, taken from [Per25].

To create patterns one creates a **record** extending the **Pat** class. This matching occurs on the now lowered and legalized SDNodes. It is possible that machine instructions cover more than one node in the DAG or a node needs to be expanded to multiple machine instructions. A pattern consists of the input in the current DAG, and an output, the machine instructions that *cover* the input. Listing 2.3 shows an example pattern for the `ADDI` instruction.

```
1 def : Pat<(add X:$rs1, RV3264Base_ADDI_immSAsInt64:$immS),
2         (ADDI X:$rs1, RV3264Base_ADDI_immSAsInt64:$immS)>;
```

Listing 2.3: TableGen Instruction selection pattern for RV64I `ADDI`

The **Pat** takes as first argument a DAG to match on. In this case this is the add operation with a register and an immediate. If this pattern is selected, the DAG will get replaced by the second argument (another DAG), in this case by the `ADDI` instruction node.

## 2.2 Introduction to VADL and OpenVADL

### 2.2.1 Vienna Architecture Description Language

VADL is a language to describe processor architectures. The specification is split into multiple sections like the ISA, Application Binary Interface (ABI), Micro Architecture (MIA) and Assembly Definition (AD) [FHH+25]. An implementation of VADL can use these sections to generate *artifacts* such as a compiler or an Instruciton Set Simulator (ISS). See Figure 2.2 for an overview. For this thesis we will mainly focus on the ISA section of VADL. It contains the encoding and behavior of instructions of an architecture.

#### Instruction Set Architecture Section

The ISA section of a VADL specification is one of the main sections. It starts with the keyword **instruction set architecture** followed by a name. An ISA contains the **instruction**s of the processor which are laid out in a **format**. An instruction consists of *behavior*, **encoding** and **assembly**. Registers are specified with **register**, which takes as type a function from Bits<Index> -> Bits<BitWidth>. The keyword **using** defines a type alias, for better

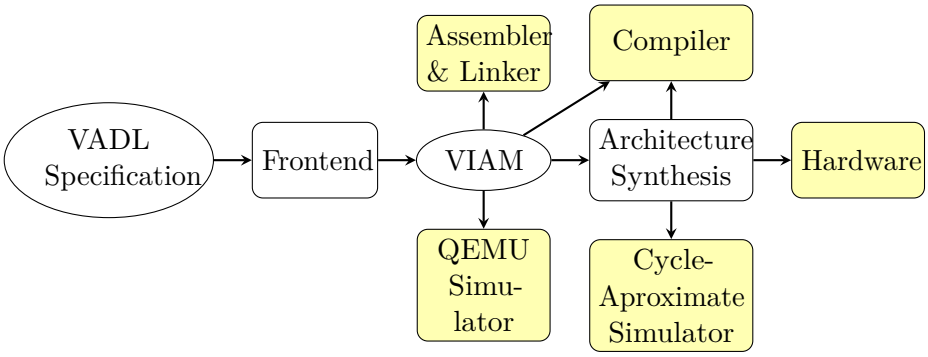


Figure 2.2: Overview of the OpenVADL architecture [FHH+26]. Generated artifacts are in yellow.

readability. Listing 2.4 shows a minimal example defining the `ADDI` instruction of RISC–V which is using the `Itype` **format**.

```
1  instruction set architecture RV32IM = {
2
3    constant Size = 32
4
5    using Byte    = Bits<8>
6    using Inst    = Bits<32>
7    using Regs    = Bits<Size>
8    using Index   = Bits<5>
9
10   [zero:X(0)]
11   register X : Index -> Regs
12
13   format Itype     : Inst =
14     { imm          : Bits<12>
15     ,rs1           : Index
16     ,funct3        : Bits<3>
17     ,rd            : Index
18     ,opcode        : Bits<7>
19     ,immS          = imm as SInt<Size>
20     }
21
22   instruction ADDI : Itype =
23     X(rd) := X(rs1) + immS
24   encoding ADDI = { opcode = 0b001'0011, funct3 = 0b000 }
25   assembly ADDI = ( mnemonic," ", register(rd),",",
26     register(rs1),",",sdec(imm))
27 }
```

Listing 2.4: Minimal example of a VADL ISA specification for `ADDI` in RISC–V .

In the listing from lines 13 to 18 we declare the different fields in the **format** with their

corresponding name and type. But sometimes the Bits<?> type of the field an a format is not the type of the value that is read. This usually happens when a value encoded in an immediate field is sign extended. Therefore VADL allows for defining field access functions which can do the necessary conversions. This can be seen at line 19. The annotation in line 10, denoted in [...] , sets the first register to always output `0`. In line 22 we define the instruction with its respective format. Line 23 specifies the *behavior* of the instruction. In this case the register rd gets set to the value of rs1 + immS. **encoding** sets the static fields in the format and **assembly** specifies how the instruction should be printed to be handled by the assembler.

## 2.3 Current open problems in OpenVADL

When generating a LLVM compiler backend we have to work with the information given in a VADL specification. As LLVM has its own structure for backends, some information can not be expressed in one system or the other or needs to be interpolated. Kevin Per calls this the *semantic gap* [Per25]. This is the main source of performance loss between a generated RISC–V compiler and an *upstream* one. This section covers some of the currently known open problems and discusses possible solutions.

### 2.3.1 Base address computation for arrays in loops

As RISC–V has a fixed instruction width of 32–bit one can only fit a limited number of bits into an immediate field. This means the address of globals that are stored in a place in memory that exceed this bit width, need to be loaded with more than one instruction. RISC–V has an instruction named `LUI rd, imm20` exactly designed to do this. `LUI` places the 32–bit U-immediate value into the destination register `rd`, filling in the lowest 12 bits with zeros [Int]. In the 64–bit variant of RISC–V the result is sign extended. Currently this computation is done for every array access in a loop, although it could be moved out of the loop as the array base address does not change between iterations.

### 2.3.2 Stack frame setup

The ABI of a platform specifies which registers a function can use as scratch space and which one need to be preserved and restored before use. This is typically done by setting up a stack frame as the scratch space for local variables. But this operation is not always necessary. When a function can manage its variables with only caller preserved registers there is no need for setting up and destroying a stack frame. Currently a function always sets up a stack frame regardless of register usage leading to quite some overhead for function calls.

### 2.3.3 Type mismatch between VADL and LLVM

Listing 2.5 shows the RV32I `SLLI` instruction in VADL.

```
format Ftype : Inst =          // Ftype instruction format
  { funct2 : Bits<2>           // [31..30] 2 bit function code
  , zero   : Bits<5>           // [29..26/25] 4 or 5 bit zero field
  , sft    : Bits<5>           // [24/25..20] 5 or 6 bit shift ammount
  , rs1    : Index             // [19..15] 1st source register index
  , funct3 : Bits<3>           // [14..12] 3 bit function code
  , rd     : Index             // [11..7]  destination register index
  , opcode : Bits<7>           // [6..0]   7 bit operation code
  , shamt  = sft as UInt       // 5/6 bit unsigned shift ammount
  }

instruction SLLI : Ftype =
  X(rd) := ((X(rs1) as UInt) << shamt) as Regs
encoding SLLI =
    {opcode = 0b001'0011, zero = 0, funct3 = 0b001, funct2 = 0b00}
assembly SLLI =
    (mnemonic, " ", register(rd), ",", register(rs1), ",", udec(sft))
```

Listing 2.5: RV32I `SLLI` instruction in VADL.

We can see that format fields can have arbitrary bit widths. This results in `shamt` having the type `UInt<5>` in VADL. This presents a few challenges for the LCB. The language used by TableGen, C++, does only support types valid in C++. These are powers of two starting at 8. Currently the next bigger fitting C++ type is chosen as a replacement for the VADL type. This conflicts with the way LLVM handles constants during instruction selection which mostly have the type of the architecture bit width. This results in the immediate first being loaded into a register and then used in a register only instruction resulting in one extra instruction being selected for every shift causing significant overhead. The problem is worsened by the fact that shifts by a power of two are often selected for a multiplication or division with said power, as they are cheaper to perform.

### Shift operations in VADL

Shift operations have a peculiar property. The second operand of a shift operation only needs to have $\log_2(Bitwidth)$ many bits to achieve every possible shift. This leaves a choice for language designers and processor manufacturers how to handle cases where this is not given, e.g. bits higher than the lowest $\log_2(64) = 6$ for a 64–Bit architecture are set. In the `C` programming language, such shifts are *undefined*. VADL takes a different approach, more in line with most processor implementations. It first computes the second operand by taking it modulo the bit width of the first operand. This ensures that the second operand is of a size that can not create undefined behavior as in the `C` specification.

CHAPTER 3

# Implementation

This chapter covers implementation details for the problems discussed earlier. All listings shown in this chapter are always compiled for the `RV64IM` ISA with optimization level `-O3` unless stated otherwise.

## 3.1 Array address calculation in loops

As discussed in Section 2.3.1, array base address calculations are not moved outside of loops. Currently the generated LCB compiles the loop of the C–code in Listing 3.1 into the suboptimal assembly shown in Listing 3.2.

```
long a[200];
long* func(long n) {
    long i = 0;
    while (i < n) {
        a[i] = 200;
        ++i;
    }
    return a;
}
```

Listing 3.1: Example of a loop with address computation

```
...
.LBB0_2:
        ADDI a2,zero,3
        SLL a2,a1,a2
        LUI a3,%hi(a)
        ADDI a3,a3,%lo(a)
        ADD a2,a2,a3
```

```
8           ADDI a3,zero,200
9           SD a3,0(a2)
10          ADDI a1,a1,1
11          BLT a1,a0,.LBB0_2
12  ...
```

Listing 3.2: Old assembly for address computation in loops

The lines 4 and 5 of Listing 3.2 show the inefficiencies of the current compilation. These instructions computing the array base address, which always produces the same result, get executed for every loop iteration. Listing 3.3 shows the improved compilation, where these operations are now in lines 1 and 2, and the loops only spans lines 5 to 8 using the previously computed value in `a0`.

```
1           LUI a0,%hi(a)
2           ADDI a0,a0,%lo(a)
3           ADD a1,a1,a0
4   .LBB0_2:
5           ADDI a2,zero,200
6           SD a2,0(a0)
7           ADDI a0,a0,8
8           BNE a0,a1,.LBB0_2
```

Listing 3.3: New assembly for address computation in loops

To enable this optimization the compiler has to ensure that the base address of the loop does not change. Furthermore, it has to use Loop Strength Reduction (LSR) to transform the incremental indexed array accesses into an Induction Variable (IV). To check if such transformations are beneficial or legal a target in LLVM implements a Target Transform Info (TTI) which contains hooks describing the target that SDISel can use. Per default LLVM makes very conservative assumptions about a target, therefore correctly implementing some of these hooks can improve compilation results a lot.

One such hook concerns legal *addressing modes* of the target. LLVM uses an attribute in its IR to describe the *data layout* of the target [Proc]. It specifies, among other things, the *native register width* of the target. Previously the LCB did not define the *native register width* which resulted in LLVM rejecting valid addressing modes when trying to perform LSR. Furthermore the generated implementation of the `TargetLowering::isLegalAddressingMode` hook was incorrectly rejecting valid *addressing modes*. This resulted in LLVM being unable to transform the incremental array access into an IV.

| Value | Description |
|---|---|
| `"none"` (default) | the frame pointer can be eliminated, and its register can be used for other purposes. |
| `"reserved"` | the frame pointer register must either be updated to point to a valid frame record for the current function, or not be modified. |
| `"non-leaf"` | the frame pointer should be kept if the function calls other functions. |
| `"all"` | the frame pointer should be kept. |

Table 3.1: Supported values for the `"frame-pointer"` attribute [Prod].

## 3.2 Stack frames

When not using optimizations LLVM sets up a stack frame for every function. This ensures a function call follows the calling convention of the ABI. Functions which use just a hand full of registers typically do not need to overwrite *callee saved* registers and therefore creating a stackframe can be omitted.

LLVM uses so called *attributes* to describe a module or function. One of this attributes for a function is the `"frame-pointer"` attribute. Table 3.1 shows all possible values and their behavior. Per default LLVM defaults to tagging each function with the `"all"` attribute for a target which inserts a stack frame for every function. This was also the case previously in the LCB.

These *attributes* get added by the *driver* `clang`, which also handles other command line flags, to a translation unit (module). This requires providing default behavior information about the target, otherwise, again, a conservative approach is assumed. The new behavior, like for most architectures, is to omit the frame pointer if any optimizations are enabled. This reduces the static instruction count of especially small and leaf functions significantly as these generally never run out of *caller saved* registers and therefore do not need a stack frame.

## 3.3 Type mismatch between VADL and LLVM

As stated in Section 2.3.3 VADL can have arbitrary width bit types while TableGen does not. Furthermore constant nodes in SDISel usually are of type `Bits<MLen>` where `MLen` is the bit width of the architecture unless a target implements a custom lowering. Listing 3.4 shows the current pattern for the `SLLI` instruction.

```
1 def RV3264Base_SLLI_shamtAsInt8
2     : RV3264Base_SLLI_shamt<i8>
3     , ImmLeaf<i8, [{ return RV3264Base_SLLI_shamt_predicate(Imm); }]>;
4
```

```
def : Pat<(shl X:$rs1, RV3264Base_SLLI_shamtAsInt8:$shamt),
          (SLLI X:$rs1, RV3264Base_SLLI_shamtAsInt8:$shamt)>;
```

Listing 3.4: TableGen Instruction selection pattern for RV64I `SLLI`

This pattern will never be selected as the `shamt` operand would have to be of type `i8` but as stated above is of type `i64` for a 64–bit architecture. A temporary approach was chosen to fix this issue by introducing an *annotation* to VADL, that changes the type of the field access. A more robust approach is discussed in Section 3.3.

### Annotations in VADL

Annotations in VADL are a way to extend the language without introducing new syntax. The annotation chosen for this feature is shown in Listing 3.5. The user provides the field `<ident>` and the new bit `<ex>` for the specified field on an instruction. This allows reusing the same **format** and only enabling the annotations for some instructions.

```
constant MLen = $ArchSize()      // MLen = 32 or 64 depending on ArchSize


model FtypeModel (zero45 : Id, shift : Id) : IsaDefs = {
format Ftype : Inst =            // Ftype shift immediate instruction format
  { funct2 : Bits<2>             // [31..30] 2 bit function code
  , zero   : $zero45             // [29..26/25] 4 or 5 bit zero field
  , sft    : $shift              // [24/25..20] 5 or 6 bit shift ammount
  , rs1    : Index               // [19..15] 1st source register index
  , funct3 : Bits3               // [14..12] 3 bit function code
  , rd     : Index               // [11..7]  destination register index
  , opcode : Bits7               // [6..0]   7 bit operation code
  , shamt  = sft as UInt         // 5/6 bit unsigned shift ammount
  }
}


model IShftInstr
    (name : Id, op : BinOp, funct3 : Bin, funct2 : Bin, lhsTy : Id)
        : IsaDefs = {
[ upcast access to : shamt, MLen ]      // introduced annotation
instruction $name : Ftype =             // shift immediate instructions
  X(rd) := ((X(rs1) as $lhsTy) $op shamt) as Regs
encoding $name =
    {opcode = 0b001'0011, zero = 0, funct3 = $funct3, funct2 = $funct2}
assembly $name =
    (mnemonic, " ", register(rd), ",", register(rs1), ",", udec(sft))
}
```

Listing 3.5: `upcast access to` annotation in VADL

Internally annotations are processed and type-checked just like any other code and then attached to a VIAM node. The LCB can check field accesses for this node and generate TableGen patterns with the requested type. If the type is not native to C++, the next bigger type is chosen. Negative values for the bit width of the type are not allowed.

With this annotation we can now define the LLVM type to be `i64`, which results in a pattern that will match.

#### Automatically generating correct immediate types

Although the prior approach works, it goes against the spirit of VADL, which should not expose such an implementation detail to the specification writer. It is therefore necessary to develop a *heuristic* that will produce the same pattern as the annotation.

Although out of the scope for this work, this type of issue manifests itself in more places than previously discussed. This is especially prevalent on some more complex architectures like `AArch64`. The new *heuristic* is to change the LLVM type to match the smallest sub-register of an architecture. As at the time of writing this thesis `RV64IM` is specified without sub-registers this changes the LLVM type to the desired 64–Bits while being flexible enough for more involved cases in the `AArch64` ISA. Full credit goes to Tobias Weiss who implemented this heuristic for OpenVADL.

CHAPTER 4

# Performance Evaluation

The following chapter contains performance evaluations for `RV64IM` using the *Embench* [FC] suite and using number of executed instructions as a metric.[1] Furthermore it contains evaluations regarding static instruction count.

## 4.1 Array address calculation in loops

As one can see in Figure 4.2 optimization had a big performance impact on specific benchmarks. Overall it improved relative performance from `0.704` to `0.780` for `RV64IM`, with especially noticeable improvements in some specific benchmarks. These benchmarks, like `edn`, use lookup tables to do some computation. There, static addresses, previously computed every loop iteration, are now moved outside the loop massively reducing executed instructions.

Static instruction count reduced slightly in most benchmarks (see Figure 4.1). This change also enables the possibility of *loop unrolling* which could increase static instruction count significantly, shadowing the reduction by the change. The compiler generated by the LCB will aggressively do this, even more aggressive than the upstream compiler, as it does not yet care about the increased pressure on the *instruction cache* this might produce. This phenomenon can be seen in the 5% increase in static instruction count for the `edn` benchmark.

## 4.2 Stack frames

Typically the prologue and epilogue for a function are 7 instructions, which can now be omitted if the function body allows for it. This results in the improvements shown in

[1] The current performance of the LCB can also be inspected at `https://openvadl.github.io/resselpark/`.

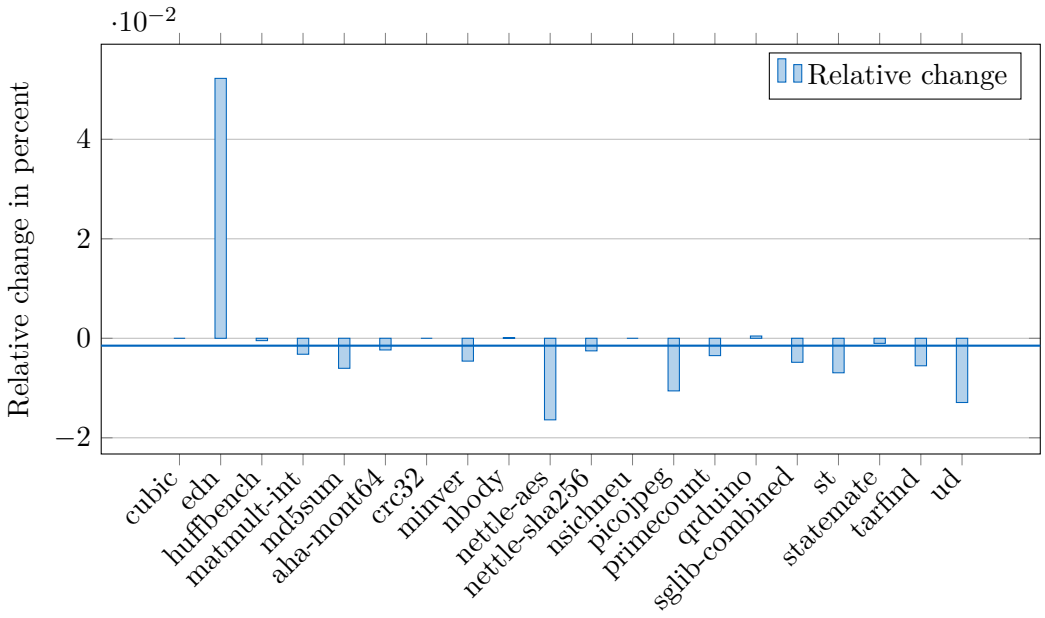


Figure 4.1: Static instruction count changes for loop invariant address calculation (lower is better)

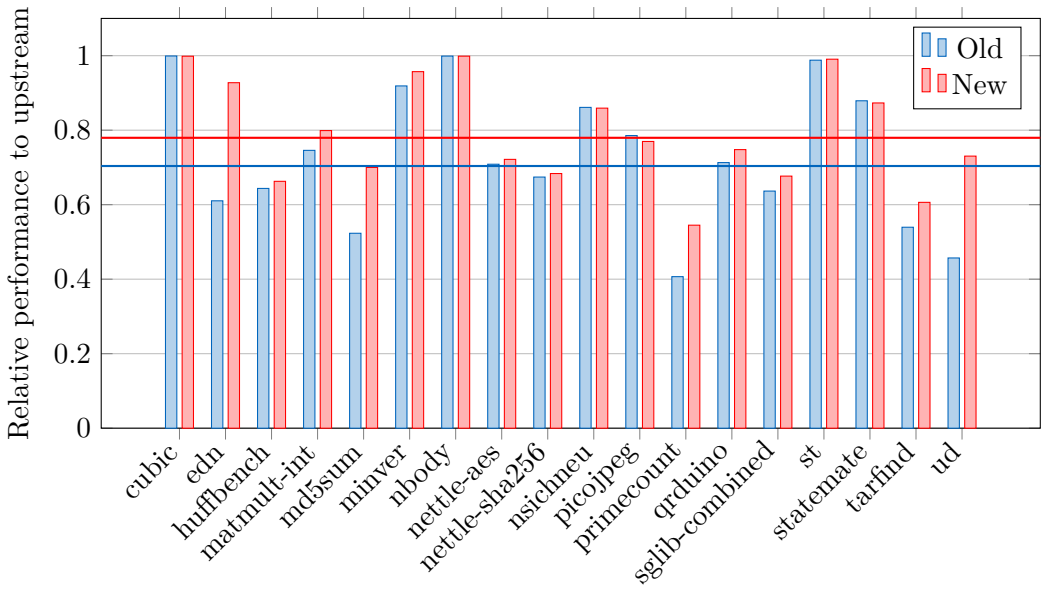


Figure 4.2: Improvements for loop invariant address calculation (higher is better)

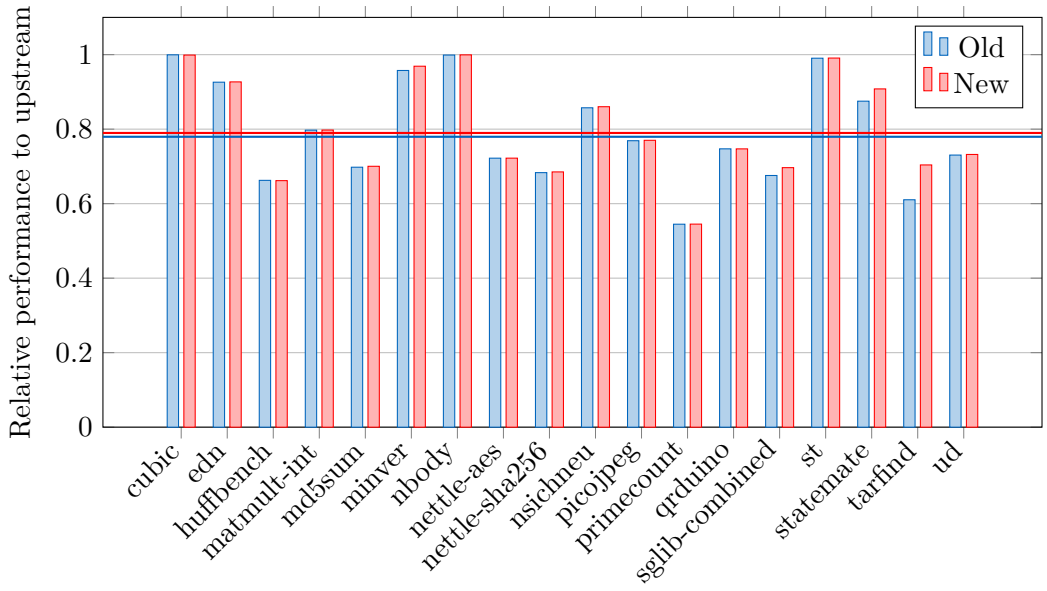


Figure 4.3: Improvements for stack frame elimination (higher is better)

Listings 4.2, 4.2 and 4.2. Figure 4.3 shows the relative performance improvements.

```
long square(long a)
{
    return a * a;
}
```

Listing 4.1: Simple function which only uses one register

```
square:
    ADDI sp,sp,-16
    SD ra,8(sp)
    SD fp,0(sp)
    ADDI fp,sp,16
    MUL a0,a0,a0
    LD fp,0(sp)
    LD ra,8(sp)
    ADDI sp,sp,16
    RET
```

Listing 4.2: Previously generated assembly

```
square:
    MUL a0,a0,a0
    RET
```

Listing 4.3: New assembly without creating a stackframe

This optimization in contrast to the previous one had a pretty small constant improvement across each benchmark by 2-3 percentage points. Overall it improved relative performance from `0.779` to `0.790`.

It also hat quite some impact on the static instruction count.

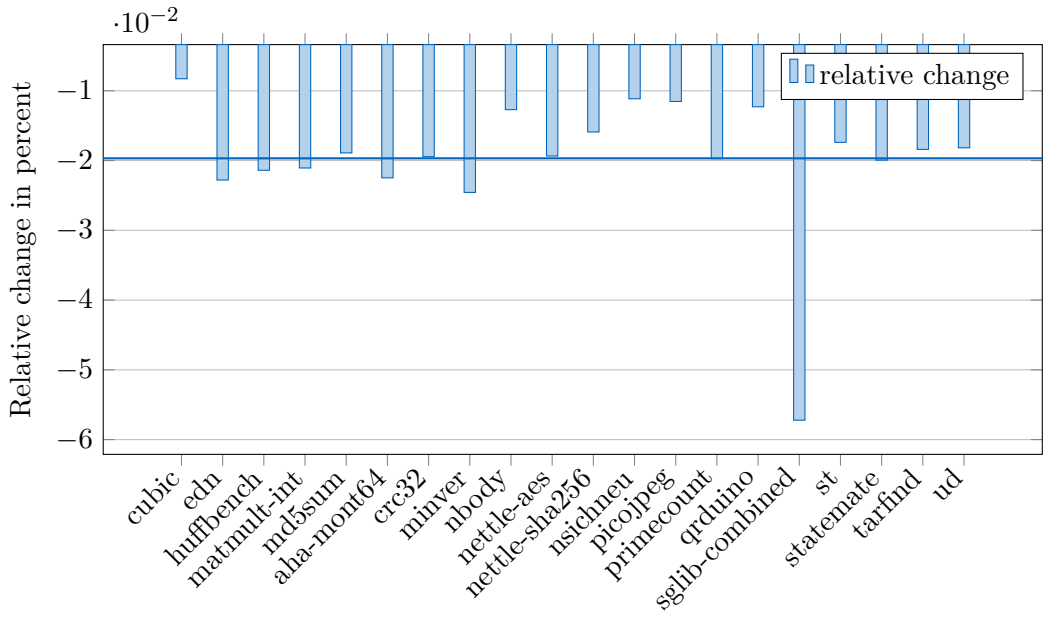


Figure 4.4: Static instruction count changes for stack frame omission (lower is better)

## 4.3 Type mismatch between VADL and LLVM

This simple looking change has some serious performance impact as already speculated in the evaluation by Kevin Per [Per25]. It reduces the instruction count for every shift and multiplication and division by a power of two by one. Furthermore this reduces register pressure as the extra register to load the constant is no longer needed potentially leading to less spills. Listings 4.4, 4.5 and 4.6 show the improved instruction selection on a simple function. This currently only works if the source and result bit width match the *native register width* of 64 bits. See Section 5.1 for more details on why this does not yet work with 32 bit types for `RV64IM`.

```
1 long shift(long a)
2 {
3     return 2 * a;
4 }
```

Listing 4.4: Simple shift operation

```
1 shift:
2     ADDI a1,zero,1
3     SLL a0,a0,a1
4     RET
```

Listing 4.5: Previously generated assembly

```
1 shift:
2     SLLI a0,a0,1
3     RET
```

Listing 4.6: New assembly without extra immediate load

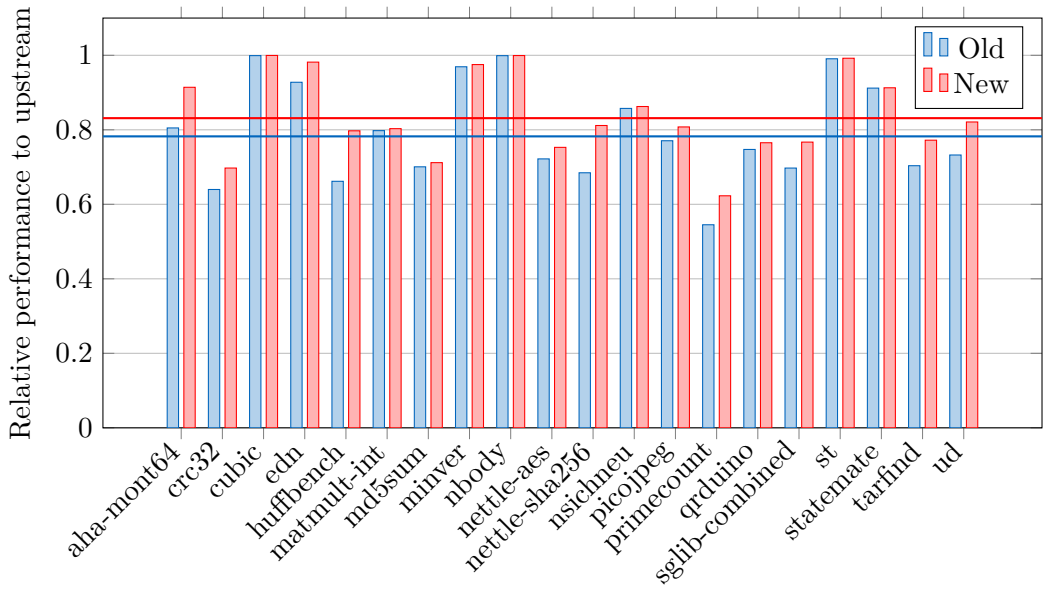


Figure 4.5: Improvements for type mismatch fix (higher is better)

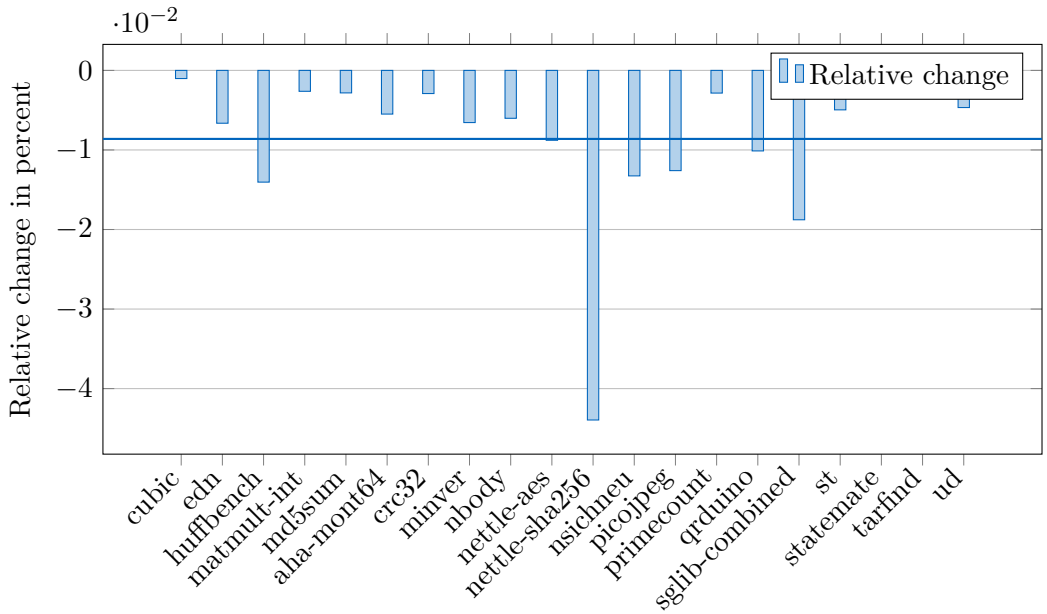


Figure 4.6: Static instruction count changes for type mismatch fix (lower is better)

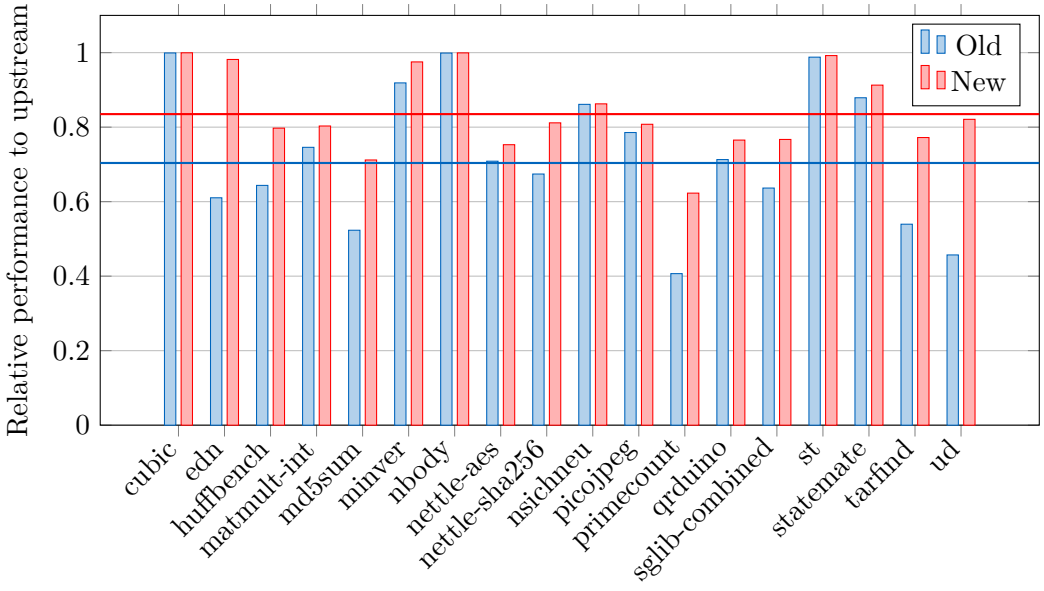


Figure 4.7: Improvements for all optimizations combined (higher is better)

## 4.4 Total improvements

The total improvements can be seen in Figure 4.7, this covers all previously discussed optimizations. This increases the performance from the original `0.704` to `0.835` for `RV64IM`. Static instruction count decreased by 3% on average, with the only increase in a benchmark caused by loop unrolling.

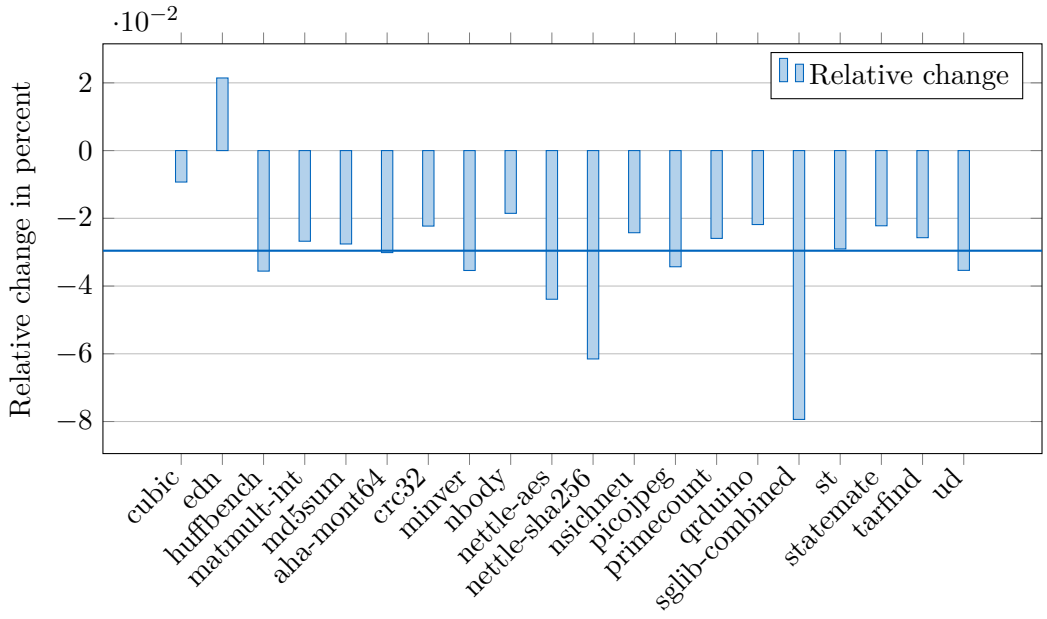


Figure 4.8: Total static instruction count changes (lower is better)

CHAPTER 5

# Future Work

## 5.1 Operations with sign extended result types

RV64IM has instructions that operate only on the lower 32–bit of a register and then sign extends the result to 64–bit. One such instruction is `SLLIW rd, rs1, uimm5`. It shifts `rs1` by `uimm5` treats this value as 32–bit and sign extends it to 64 bits. The upstream compiler uses custom logic to select this instructions, but the LCB is currently not capable of selecting this instruction. The reason being that typically a sign_extend_inreg gets expanded in the operation legalization phase (see Figure 2.1 and Table 2.1). This results in the generated pattern of `SLLIW` not matching anymore.

This means that the generated compiler by LCB is currently incapable of selecting any of the `w-type` instructions of RV64IM. Implementing this would require a lot of work in the LCB as this likely would have to be done via a `custom LowerOperation` (like the implementation in the upstream compiler) and is currently not a focus of development.

## 5.2 Efficient loads of globals

Addresses of globals do not fit in one immediate field of an instruction in RISC–V . This creates the necessity to assemble the address first before any loads from such an address can take place. But RISC–V provides a 12–bit offset field in its load instruction making it redundant to assemble the whole address in a register and then loading with 0 offset. It is instead possible to incorporate the offset inside this immediate field. Listings 5.1 and 5.2 show an example of this optimization saving one instruction per load.

```
1 LUI a3,%hi(global_addr)
2 ADDI a3,a3,%lo(global_addr)
3 LW a2,0(a3)
```

Listing 5.1: Currently generated assembly

```
1 LUI a3,%hi(global_addr)
2 LW a2,%lo(global_addr)(a3)
```

Listing 5.2: Optimal sequence saving one instruction

CHAPTER 6

# Conclusion

In this thesis we saw that VADL is more than capable at expressing a real world ISA, like RISC–V and delivering respectable performance. But each specification comes with its own challenges that need to be handled in the compiler generator, such that a compiler backend can generate good code. In the case of LLVM these mostly manifest in missing TableGen patterns and target hooks. Correctly generating these from a VADL specification can range from trivial to extremely complex. Overall we improved relative performance from `0.704` to `0.835` compared to *upstream* for `RV64IM`.

# Overview of Generative AI Tools Used

No AI tools were used during the creation of this work.

# List of Figures

# List of Tables

# Acronyms

**ABI** Application Binary Interface. 8, 10, 15

**AD** Assembly Definition. 8

**DAG** Directed Acyclic Graph. 5, 6, 8

**DSE** design space exploration. 1

**DSL** Domain Specific Language. 7

**FastISel** Fast Selection DAG Instruction Selection. 5

**GlobalISel** Global Selection Dag Instruction Selection. 5

**IR** Intermediate Representation. 3, 5, 6, 14

**ISA** Instruction Set Architecture. 1, 5, 6, 8, 9, 13, 17, 29

**ISS** Instruciton Set Simulator. 8

**IV** Induction Variable. 14

**LCB** LLVM Compiler Backend Generator. 1, 5, 11, 13–15, 17, 19, 27

**LLVM** LLVM. xi, xiii, 1, 3–7, 10, 11, 14, 15, 17, 29

**LLVM IR** LLVM Intermediate Representation. 3–5

**LSR** Loop Strength Reduction. 14

**MIA** Micro Architecture. 8

**OpenVADL** OpenVADL. xv, 1, 5, 7, 10, 11, 17

**SDISel** Selection DAG Instruction Selection. 5, 6, 14, 15, 33

**SDNode** Selection Directed Acyclic Graph Node. 5, 6, 8

**SSA** Static Single Assignment. 3, 4

**TableGen** TableGen. 5–8, 11, 15–17, 29

**TTI** Target Transform Info. 14

**VADL** Vienna Architecture Description Language. xi, xiii, xv, 1, 3, 5, 7–11, 15–17, 29, 37

**VIAM** VADL Intermediate Architecture Model. 17

# Bibliography


[FC] Free and Open Source Silicon Foundation CIC. Embench: A modern embedded benchmark suite. https://www.embench.org/. Accessed: 2026-03-12.

[FHH+25] Florian Freitag, Linus Halder, Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Benjamin Kasper, Niklas Mischkulnig, Michael Nestler, Philipp Paulweber, Kevin Per, Matthias Raschhofer, Alexander Ripar, Tobias Schwarzinger, Johannes Zottele, and Andreas Krall. The vienna architecture description language, 2025. URL: https://arxiv.org/abs/2402.09087, arXiv:2402.09087.

[FHH+26] Florian Freitag, Linus Halder, Benedikt Huber, Benjamin Kasper, Michael Nestler, Kevin Per, Matthias Raschhofer, Alexander Ripar, Johannes Zottele, and Andreas Krall. Openvadl: An open source implementation of the vienna architecture description language. In Sven Tomforde, Christian Krupitzer, Stéphane Vialle, Estela Suarez, and Thilo Pionteck, editors, *Architecture of Computing Systems*, pages 156–171, Cham, 2026. Springer Nature Switzerland.

[FM] Justin M. Fargnoli and Alex E. MacLean. A beginner's guide to selectiondag. https://llvm.org/devmtg/2024-10/slides/tutorial/MacLean-Fargnoli-ABeginnersGuide-to-SelectionDAG.pdf. Accessed: 2026-02-17.

[Int] RISC-V International. The RISC-V Instruction Set Manual. https://docs.riscv.org/reference/isa/unpriv/rv32.html. Accessed: 2026-02-10.

[Muc98] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[Per25] Kevin Per. *LLVM Compiler Generator in OpenVADL*. reposiTUm, Wien, 2025. doi:https://repositum.tuwien.at/handle/20.500.12708/224999.

[Proa] LLVM Project. Global instruction selection. https://llvm.org/docs/GlobalISel/index.html. Accessed: 2026-02-16.

[Prob] LLVM Project. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html. Accessed: 2026-02-17.

[Proc] LLVM Project. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html#data-layout. Accessed: 2026-03-02.

[Prod] LLVM Project. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html#function-attributes. Accessed: 2026-03-03.

[Proe] LLVM Project. LLVM's analysis and transform passes. https://llvm.org/docs/Passes.html#indvars-canonicalize-induction-variables. Accessed: 2026-05-02.

[Prof] LLVM Project. LLVM's analysis and transform passes. https://llvm.org/docs/Passes.html#loop-reduce-loop-strength-reduction. Accessed: 2026-05-02.

[Prog] LLVM Project. Tablegen programmer's reference. https://llvm.org/docs/TableGen/ProgRef.html. Accessed: 2026-02-17.

[RT22] Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based compiler design*. Springer, 2022. doi:10.1007/978-3-030-80515-9.