Generation of a QEMU based instruction set simulator from a processor description in OpenVADL

Johannes Zottele[0009−0001−5328−9181], Matthias Raschhofer[0009−0006−0445−3738], Benedikt Huber[0009−0005−7059−3555], and Andreas Krall[0009−0002−7668−6259]

Technische Universität Wien, Vienna, Austria

Abstract. QEMU (Quick EMUlator) is a generic and open source machine emulator and virtualizer which employs Dynamic Binary Translation (DBT) to emulate a guest architecture on a host architecture. OpenVADL is an open source implementation of the Vienna Architecture Description Language (VADL), a processor description language developed for rapid design space exploration in the area of processor design. OpenVADL automatically generates various essential artifacts. One such artifact is the Instruction Set Simulator (ISS), which enables the execution of programs compiled for the described processor on different host systems. To achieve high-performance simulation with broad platform support, OpenVADL's ISS generator produces a QEMU frontend, seamlessly integrating into the QEMU system. This integration leverages QEMU's DBT, along with built-in features such as GDB debugging. Optimized generation of QEMU's TCG intermediate representation ensures competitive performance, even when compared to manually written and optimized frontends. Benchmark results show that the OpenVADL-generated QEMU based ISS achieves a speedup of up to 1.77 compared to the official handwritten QEMU frontend for the RISC-V RV64IM instruction set architecture.

Keywords: Processor description language · QEMU generator

1 Introduction

This article describes how OpenVADL automatically generates a QEMU [3] frontend from a concise Vienna Architecture Description Language (VADL) processor specification.

A primary goal of VADL is to make design space exploration in the area of processor design convenient and efficient. For meaningful experimentation and evaluation of a new processor design a functional Instruction Set Simulator (ISS) is an absolute necessity. Manually constructing an ISS from a given specification is tedious, error prone and it is difficult to guarantee consistency with the specification. To avoid these problems OpenVADL automatically generates the ISS from a VADL processor specification.

In particular, OpenVADL generates a frontend for the well established QEMU emulator and virtualizer. QEMU is designed with retargetability and extensibility in mind. So the generated frontend integrates well with the remaining QEMU system. This way OpenVADL also benefits from optimizations already implemented in the QEMU backend. QEMU uses Dynamic Binary Translation (DBT) in order to achieve efficient simulation.

OpenVADL translates a VADL specification into OpenVADL's Intermediate Representation (IR) which is common to all of its generators, the VADL Intermediate Architecture Model (VIAM). Subsequently the ISS generator translates the VIAM into QEMU's IR called TCG. This TCG representation of the specified architecture with its instructions constitutes the QEMU frontend. OpenVADL generates an efficient decode tree for instruction decoding which is integrated in the generated ISS, but can also be used by other OpenVADL artifacts. To the best of our knowledge, OpenVADL is the only system capable of generating a QEMU frontend from a high level Processor Description Language (PDL).

The evaluation of the generated ISS with the Embench benchmark suite shows competitive performance results, that outperform the handwritten QEMU frontend for the RISC-V architecture in all evaluated cases.

2 The Vienna Architecture Description Language

Listing 1.1 shows a complete Instruction Set Architecture (ISA) specification of all RISC-V instructions with immediate operands. In line 3 a constant for the register size is defined. Lines 5 to 9 declare user defined types. VADL supports bit vector types. The basic type is `Bits`. There exist two subtypes representing signed (`SInt`) and unsigned (`UInt`) two's complement integers. Line 12 demonstrates the definition of a register file. Annotations can be used to detail a definition as the specification of a zero register demonstrates (see line 11). An implicitly updated program counter is required in every ISA specification (line 14). A format definition is used to specify bitfields with named and typed member fields (line 16 to 23).

Usually, many instruction definitions are quite similar. VADL supports type safe syntactic macro templates to avoid copying and modifying specifications. A macro definition starts with the keyword `model` followed by the typed arguments and the result type of the macro. There exist syntactic types like `Id` (identifier), `BinOp` (binary operator), `Bin` (binary constant) or `IsaDefs` (ISA definitions). An instantiation of a macro or the substitution of a macro argument are indicated by the dollar sign.

An `instruction` defines the behavior of an instruction (line 27). The `encoding` sets the fields in an instruction word which are constant for the given instruction (line 29). The `assembly` specifies the assembly language syntax for the instruction with a string expression (line 30). By packing these three definitions into a macro, we can specify an instruction with immediate operands in one single line. This macro is invoked six times for all RISC-V instructions with immediate operands (lines 33 to 38).

```
instruction set architecture RV32I = {

  constant Size = 32                  // architecture size is 32 bits

  using Byte    = Bits<  8 >          // 8 bit Byte
  using Inst    = Bits< 32 >          // instruction word type
  using Regs    = Bits<Size>          // register word type
  using Index   = Bits<  5 >          // 5 bit register index type for 32 registers
  using Addr    = Regs                // address type is equal to the register type

  [zero : X(0)]                       // register with index 0 always is zero
  register         X : Index -> Regs  // integer register file with 32 registers
  memory         MEM : Addr  -> Byte  // byte addressed memory
  program counter PC : Addr           // PC points to the start of an instruction

  format Itype  : Inst =              // immediate instruction format
    { imm       : Bits<12>            // [31..20] 12 bit immediate value
    , rs1       : Index               // [19..15] 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
    , immS      = imm as SInt<Size>   // sign extended immediate value
    }

  // macro for immediate instructions with name, operator, type and function code
  model ItypeInstr (name : Id, op : BinOp, type: Id, funct3 : Bin) : IsaDefs = {
    instruction $name : Itype =
       X(rd) := (X(rs1) as $type $op immS as $type) as Regs
    encoding $name = {opcode = 0b001'0011, funct3 = $funct3}
    assembly $name = (mnemonic," ",register(rd),",",register(rs1),",",decimal(imm))
    }

  $ItypeInstr (ADDI ; + ; SInt ; 0b000) // add immediate
  $ItypeInstr (ANDI ; & ; SInt ; 0b111) // and immediate
  $ItypeInstr (ORI  ; | ; SInt ; 0b110) // or immediate
  $ItypeInstr (XORI ; ^ ; SInt ; 0b100) // exclusive or immediate
  $ItypeInstr (SLTI ; < ; SInt ; 0b010) // set less than immediate
  $ItypeInstr (SLTIU; < ; UInt ; 0b011) // set less than immediate unsigned
  }
```

Listing 1.1. ISA specification of RISC-V instructions with immediate operands.

This example only touches the surface of the rich and complex language. A detailed description of VADL and its original implementation is contained in [5]. An overview of the open source implementation OpenVADL is presented in [6].

3 OpenVADL Overview

Figure 1 gives an overview of the OpenVADL architecture. The frontend translates a VADL processor specification into the VIAM. The architecture synthesis maps all parts of an instruction's behavior to the correct Microarchitecture (MiA) elements.

Architecture synthesis, assembler and linker generation, compiler generation, hardware generation and the generation of a cycle approximate instruction set simulator are described in detail in [5]. The current article focuses on the details regarding the generation of the QEMU based ISS.

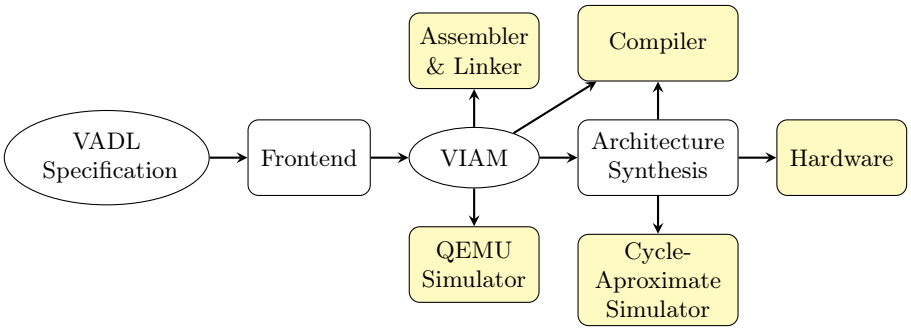

Fig. 1. Overview of the OpenVADL architecture. Generated artifacts are in yellow.

3.1 VADL Intermediate Architecture Model (VIAM)

The VIAM is OpenVADL's IR, divided into two parts. The first is an easily accessible hierarchical data structure containing all declarative definitions of a VADL specification, such as `format`, `instruction`, and `encoding`. The second is a multigraph combining a Control Flow Graph (CFG) and a dependency graph, representing the behavioral aspects of VADL, such as instruction semantics.

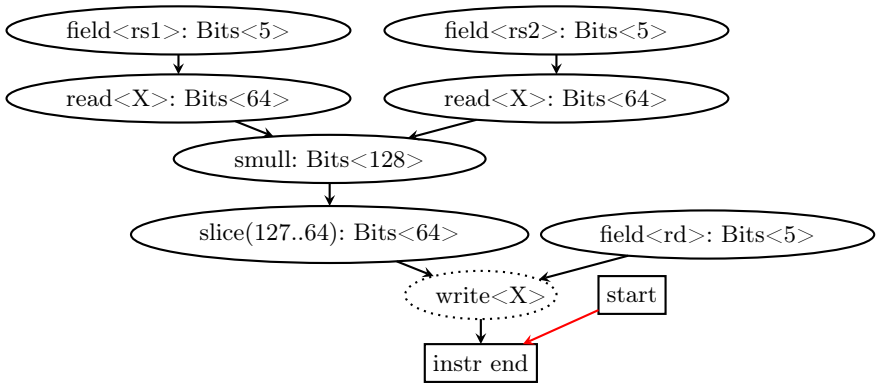

Fig. 2. VIAM behavior graph of the RISC-V `MULH` instruction with a minimal CFG. Red arrows represent control flow edges, while black arrows indicate dependencies.

The CFG in a VIAM behavior graph consists of at least a start and an end node (see Figure 2), and may include if-else nodes to represent diverging control flow. Each branch is enclosed by a branch start and a branch end node.

In VADL's instruction behavior, all reads semantically occur before all writes. This means a value written to a resource (register or memory) during an instruction cannot be read within the same instruction. Consequently, the order of side effects (i.e., writes) does not need to be captured in the behavior graph and is

not part of the CFG. Instead, side effects are modeled as dependencies of branch end nodes. For instance, in Figure 2, the write to register X is a dependency of the `instr end` node. All expressions—including register and memory reads—are part of the dependency graph and appear only once in the graph.

4 QEMU Background

QEMU uses DBT to accelerate guest program execution. This involves splitting guest code into Translation Blocks (TBs), which are instruction sequences that execute as a unit and contain no external jumps, interrupts or state changes. As shown in Figure 3, each instruction in a TB is translated by the guest frontend into QEMU's IR of Tiny Code Generator (TCG) operations. These operations are then compiled into native host instructions by the host backend. The resulting native code is executed directly until the TB ends.

The separation between frontend and backend makes QEMU highly extensible: supporting a new guest architecture only requires implementing the corresponding frontend.

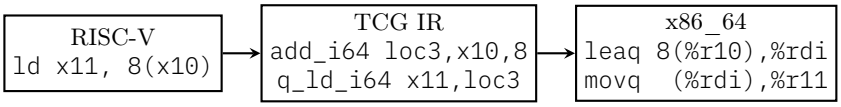

Fig. 3. The TCG translation process involves the guest frontend (e.g., RISC-V) converting instructions into TCG IR, which is then translated by the host backend (e.g., x86_64) into native host instructions for execution.

TCG operations form a quadruple code IR that operates on strongly-typed TCG variables representing input and output operands. These variables are categorized as constant, temporary, or global: constant variables hold read-only immediate values, temporary variables exist only within a TB and are discarded afterward, and global variables persist across the simulation, representing CPU state like registers.

The main QEMU emulation flow is driven by the main execution loop. Given the current Program Counter (PC) and CPU TB state, it performs a TB lookup in the TB cache to check if a translated TB for this combination already exists. A TB can only be reused if it was translated with the same CPU TB state, which is computed at the start of translation and must remain unchanged throughout the TB. This state typically includes properties like the privilege mode active during translation.

If the TB is found in the cache, it can be entered directly; otherwise, it must first be translated. When a TB finishes execution, control returns to the main loop, which performs another TB lookup.

Since entering, exiting, and looking up TBs is costly, QEMU uses direct block chaining to improve performance. This allows the current TB to jump directly

to the next TB without returning to the main loop. Internally, a target TB is assigned to a jump slot; if the instruction later jumps to the same slot, control is transferred directly. This optimization is only possible when the target PC is predictable and consistent across executions—e.g., for direct jumps based on the current PC and a fixed offset. If the new PC depends on values that are not TB-constant, chaining cannot be applied.

5 QEMU Generator

The OpenVADL ISS generator produces a QEMU frontend from a VADL specification, which is directly integrated into the QEMU system. This integration enables the use of guest-agnostic features already present in QEMU, simplifying the extension of the generated ISS's feature set. Figure 4 provides an overview of the QEMU ISS generation process.

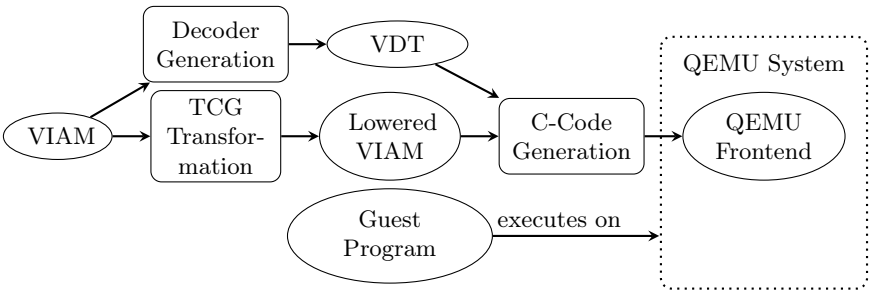

Fig. 4. Overview of the QEMU frontend generation.

5.1 QEMU Guest Generation

The entry point for QEMU frontend generation is the VADL `processor` definition, which specifies the simulated ISA, as shown in Listing 1.2. The `start` address property defines the reset vector where ISS execution begins.

```
[htif]
processor Spike implements RV64IM {
    start = 0x80000000
}
```

Listing 1.2. Processor definition with HTIF support.

A QEMU frontend defines the CPU state, machine definition, and instruction translation.

The CPU state primarily consists of registers defined in the ISA and can be directly extracted from the VIAM during C code generation. It may also include explicitly declared properties, such as privilege mode, which form the CPU TB state needed for TCG operation translation.

The machine model specifies the emulated hardware platform, including memory regions, peripherals, and initialization routines. Currently, the frontend generates RAM starting at the `start` address and omits peripheral devices. By declaring the processor definition with the `[htif]` annotation, the machine supports the Berkeley Host-Target Interface (HTIF), a simple protocol from the RISC-V Spike simulator for host-simulation communication. HTIF maps memory addresses to callbacks that interpret commands, enabling features like controlled exits from full-system emulation—useful for self-verifying tests on the ISS.

The core of ISS generation lies in creating instruction translation functions. Each ISA instruction requires a corresponding function that translates it into a sequence of TCG operations. By transforming and optimizing the instruction's VIAM graph, it is lowered into a form suitable for generating TCG translation code in C. The key steps of this process are outlined in the following sections.

To distinguish between different execution phases, the following terms are used:

- **Generation time**: the phase in which the QEMU frontend is generated by the OpenVADL compiler
- **Translation time**: the phase during QEMU execution where the frontend generates TCG operations
- **Runtime/Execution time**: the phase in which the translated native instructions are executed

Operation Decomposition In VADL, types are unconstrained in size, while QEMU operations and variables are restricted to 32 or 64 bits. When an ISA specification defines types exceeding 64 bits, they cannot be directly translated into TCG IR. Instead, these operations must be decomposed into smaller sub-operations operating on 64 bits or less. This decomposition is required even on 64-bit architectures like RISC-V, as shown in Figure 2, which depicts the VIAM graph for the `MULH` instruction. The instruction performs a signed 64-bit multiplication yielding a 128-bit result. To handle this, the multiplication and the slice extracting the upper 64 bits are replaced with a custom `IssMulh` node, eliminating the oversized type.

Normalization At this stage, all operations are restricted to the QEMU target size (32 or 64 bits) or smaller. To ensure correctness when executing VADL operations on narrower types, these operations must be normalized to the target size. This involves normalizing operands and results by inserting `IssExtract`

nodes that extend or truncate values—using either sign- or zero-extension of the lower bits to match the target size.

For example, to execute `VADL::asr(Bits<50>, UInt<5>)` (arithmetic shift right) using a 64-bit TCG operation, the first operand must be sign-extended from 50 to 64 bits, while the second operand must be truncated to 5 bits.

While this normalization guarantees correctness, it may introduce overhead in cases where operand widths do not affect the result. For instance, `VADL::add(Bits<50>, Bits<50>)` requires only the result to be truncated to 50 bits; the operand widths are irrelevant. A subsequent optimization pass removes such redundant `IssExtract` nodes and merges chained extracts into a single one to reduce overhead.

Side Effect Scheduling Side effects in the VIAM behavior graph are modeled as dependencies of the branch's end node. This means their execution order is unspecified, but all must be executed for a given execution path. Although PC modifications are regular side effects in the VIAM, they are emitted as jump instruction sequences in TCG IR. Thus, other side effects must be scheduled before PC changes to ensure execution during simulation.

PC modifications are added as dependencies to a new `InstrExit` node, which is scheduled just before the branch's end node in the control flow. All other side effects are inserted directly after the corresponding branch start node of the branch end node they depend on.

Safe Resource Read In VADL instruction behavior, all reads semantically occur before all writes. To enforce this, all potentially conflicting reads must be scheduled before the corresponding writes. As register indices and memory addresses are not known at generation time, reads to these resources must be conservatively assumed to conflict with all writes to the same resource.

TCG Expression Scheduling It is necessary to distinguish between expressions evaluated at TCG translation time and those at instruction execution time. Runtime expressions—such as those involving CPU state or memory (e.g., register reads)—must be lowered to TCG operations and scheduled. In contrast, expressions relying only on immediate values (e.g., format fields) or constant TB state (e.g., the current instruction's PC) are resolved at translation time. Their dependency trees are directly translatable to C expressions and require no scheduling. After this step, all TCG-relevant dependency nodes are correctly scheduled.

TCG Branch Lowering In the VIAM behavior graph, diverging control flow is represented with if-else control nodes. In TCG, it is modeled using jump-like operations targeting labels in the emitted operation sequence. If the condition was previously scheduled and marked as TCG operations, the if-else control flow is translated into a linear sequence of jumps and labels.

If the condition wasn't scheduled, the if-else remains and models a C-level if-else. Only the branch selected at translation time is emitted as TCG operations, minimizing execution time of the instruction.

TCG Operation Lowering All scheduled dependency nodes in the instruction behavior are lowered to control nodes representing TCG operations. Each TCG operation node has one predecessor and one successor. TCG variables used as inputs or outputs are modeled as dependency nodes of these operations, as illustrated in Figure 5.

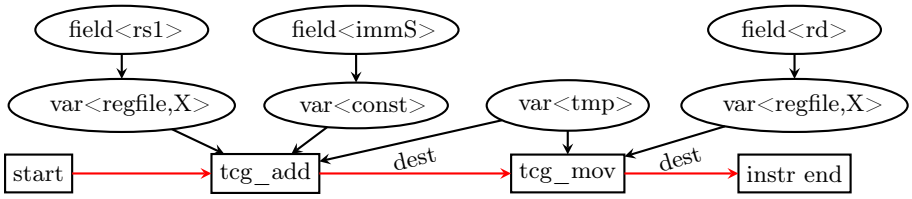

Fig. 5. Lowered VIAM behavior graph of the RISC-V `ADDI` instruction.

The resulting CFG is in Static Single Assignment (SSA) form. This means that every intermediate result is stored in separate variables and no variable is reassigned.

Lowering of `InstrExit` nodes is optimized for efficient jumps between TBs. If the PC update depends on values unknown at translation time, a TCG TB lookup is emitted. This exits the current TB and returns to the main execution loop, where the CPU TB state is evaluated and a new TB is looked up—necessary because the update PC may vary between executions of the instruction.

If the new PC is fully known at translation time (i.e., depends only on immediates or the constant TB state), it is guaranteed to be the same across executions. In this case, QEMU's jump slot optimization is applied to directly chain TB jumps, avoiding entering the main loop.

TCG Variable Allocation TCG variables used by previously lowered operations must be obtained before use. Temporaries and constants are allocated via TCG functions. Registers are accessed through generated getters that enforce register file constraints—e.g., zero registers (Figure 1.1, line 12). If a source register matches a constraint, a constant TCG variable is returned instead of accessing the actual register. For destination registers, a temporary variable is returned to prevent direct writes to constrained registers.

C Code Generation Each instruction's behavior graph is translated into a C function named `trans_<mnemonic>`, invoked by the decoder during the TCG translation loop. C code generation is straightforward, as the lowered VIAM

graph maps closely to the function body: if-else nodes become C if-else statements, expression trees become C expressions, and TCG control nodes are translated to TCG operation function calls. Figure 1.3 shows the generated translation function of the RISC-V 64 `ADDI` instruction. Since TCG and host-side optimizations handle temporary variable elimination, no extra pass to remove them is needed.

```
static bool trans_addi(DisasContext *ctx, arg_addi *a) {
    TCGv_i64 regfile_x_rd_dest = dest_x(ctx, a->rd);
    TCGv_i64 regfile_x_rs1     = get_x(ctx, a->rs1);
    TCGv_i64 const_immS_n3     = tcg_constant_i64(a->immS);
    TCGv_i64 tmp_n4_0          = tcg_temp_new_i64();
    tcg_gen_add_i64(tmp_n4_0, regfile_x_rs1, const_immS_n3);
    tcg_gen_mov_i64(regfile_x_rd_dest, tmp_n4_0);
    return true;
}
```

Listing 1.3. Generated translation function of the RISC-V 64 ADDI instruction

When configuring QEMU, a new guest is available that corresponds to the used specification. By building QEMU with this new guest, the user obtains a working QEMU executable called `qemu-system-<isa-name>`. This binary can be used to run bare-metal programs compiled for the ISA of the VADL specified processor by executing `qemu-system-<isa-name> -bios <prog>`.

5.2 GDB Stub Generation

QEMU includes a GDB stub for debugging programs running on QEMU with GDB. While most features are guest-agnostic (e.g., single-step execution), frontends must provide register information—such as name, size, and type—and implement read/write access functions for those registers.

The ISS generator produces register information using names from the Application Binary Interface (ABI) definition if attached to the processor definition; otherwise, it falls back to the register names in the ISA definition. It also generates register access functions that read and modify the CPU state as needed.

5.3 Decoder Generation

To be able to provide an efficient instruction decoder to all of its backend generators, not only the QEMU based ISS, OpenVADL offers a custom implementation for decode tree generation. It constructs an abstraction of the decode procedure, coined the VADL Decode Tree (VDT), as part of its intermediate representation. The responsibility of the VDT is to model the decode decision procedure, and

to provide all necessary information to be able to directly translate it to C-code (in the case of the ISS) to decode binary instruction words, extract the format fields specified in VADL to a structured representation, and hand it to the TCG translation operations.

In addition to the advantages provided by DBT, fast instruction set simulation requires rapid decision procedures for decoding instructions. Commonly, decoders are modeled as decision trees (sometimes also Directed Acyclic Graphs (DAGs)), where the inner nodes of the tree represent a decision based on certain bits in the input instruction word. The leafs of the decode tree hold the instructions to be matched. The cost of decoding a single instruction consists of the length of the decision path, the cost of the decision operations along the path and some constant overhead for field extraction. It is thus desirable to construct a shallow tree (while keeping the overall node count manageable) and restrict the decision functions to primitive operations.

Generating the VDT is based on constructing encoding patterns from the instructions specified in VADL. Each instruction in VADL references a `format`, which defines the fields in the instruction encoding. An additional encoding definition specifies constant bits (usually the opcode), which are used to identify the instruction during the decoding step. From this specification, OpenVADL derives a bit pattern, indicating which bits in the encoding are fixed and which are unknown ahead of time. The pattern and the task of pattern matching can thus be efficiently implemented using bitmasks and primitive bitwise operations.

The current VDT generation algorithm 1 is largely based on the procedure initially described by Henrik Theiling [11]. It starts with a set of bit patterns and their associated instructions and keeps track of the bits that have yet to be tested using the mask parameter. Initially this is a fully set bit mask, but upon recursion the already tested bits are removed to propagate the state of the decoding path.

First, the algorithm selects the bits to partition the entry set, by collecting bits that are fixed in all entries. By selecting the maximum set of (possibly disjoint) constant parts of the instruction word to split the input set, the algorithm ensures that the function is decisive, while making sure that the matching function can be performed efficiently.

After possibly terminating, or computing a fallback node in the presence of subsumed instructions, the algorithm partitions the entry set by the selected significant bits. Each of these partitions is then handled recursively and will be transformed to its own decision sub-tree.

Finally, the partitioned child nodes and the optional fallback node are then combined to a new inner decision node of the decode tree.

After constructing the VDT, it can be used to materialize the decoder for several of OpenVADL's backends. For the ISS this entails generating equivalent C-code to model the decode procedure. The decision nodes are compiled to nested switch statements, selecting the bits to test at each inner decision node and switching over the possible partitions in the instruction set. Once a leaf node has been reached, OpenVADL additionally calls a previously generated C-

Algorithm 1 Decode tree generation in OpenVADL

```
1: Input: mask ← ALL_SIGNIFICANT()
2: Input: insns ← Set of tuples (BitPattern, Instruction)
3: Output: VDTNode
4: function GENERATE_VDT(mask, insns)
5:     m ← mask
6:     for all (bitPattern, _) ∈ insns do
7:         m ← m ∧ SIGNIFICANT(bitPattern)
8:     end for
9:     fallback ← None
10:    if m = 0 then
11:        if LEN(insns) = 1 then
12:            return VDT_LEAF(insns[0])
13:        else
14:            (fallback, subsumed, new_mask) ← FIND_FALLBACK(mask, insns)
15:            insns ← subsumed
16:            m ← new_mask
17:        end if
18:    end if
19:    partitions ← PARTITION(m, insns)
20:    children ← {}
21:    for all (bitPattern, p_insns) ∈ partitions do
22:        new_mask ← mask ∧ ¬m
23:        children[bitPattern] ← GENERATE_VDT(new_mask, p_insns)
24:    end for
25:    return VDT_DECISION(mask, children, fallback)
26: end function
```

function to extract the instruction's format fields to a structured model and calls the appropriate TCG translation procedure.

6 Evaluation

To evaluate the runtime performance of the generated QEMU frontend, a comparison was made between the QEMU generated from the RISC-V RV64IM VADL specification and the upstream QEMU `riscv64` frontend. The Embench [2] benchmark suite was used, as it provides a diverse set of bare-metal benchmarks. All benchmarks were executed on both an AArch64 Apple Silicon system (Figure 7) and an x86-64 system (Figure 6).

All results are shown as speedup relative to the upstream `riscv64` frontend (baseline at 1). To assess the impact of specific optimizations, all benchmarks were also executed using generated frontends with selected optimizations disabled. The two most significant optimizations are included in the results.

Figures 6 and 7 show that the generated QEMU is at least as fast as the upstream version. This is because, aside from additional TCG `mov` op-

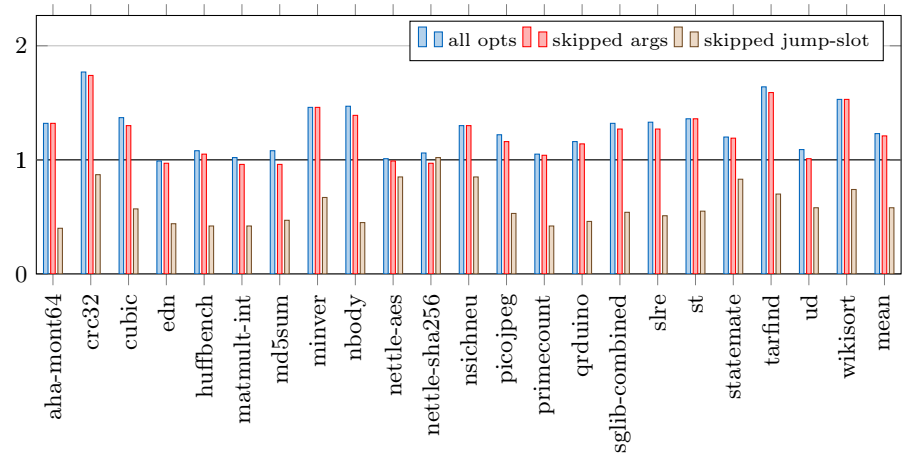

Fig. 6. Speedup of generated QEMU relative to upstream on x86-64 (higher is better)

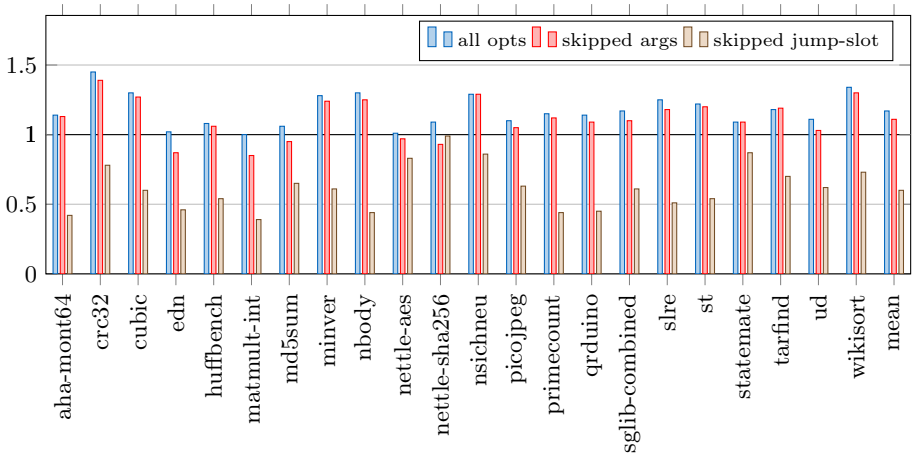

Fig. 7. Speedup of generated QEMU relative to upstream on AArch64 (higher is better)

erations—which are optimized away by the host backend—the generated instruction translation functions emit the same number of TCG operations.

Additionally, the results show that the generated QEMU achieves a speedup of up to 1.77 in some benchmarks on x86-64. This is due to the simplicity of CPU TB state calculation in the generated frontend, compared to the upstream `riscv64` implementation, which handles many more RISC-V extensions than `RV64IM`.

Performance analysis using the Linux `perf` tool [1] shows that benchmarks like `edn` spend almost no time in the TB state calculation function, whereas others like `crc32` invoke it frequently. For example, on upstream QEMU, `crc32`

spends 47% of its execution time in `cpu_get_tb_cpu_state`. This is due to tight loops executing `JALR` instructions, which update the program counter with a register value. As described in Section 5.1, such dynamic updates—unknown at translation time—cause a jump to the main loop and a TCG TB lookup, requiring recomputation of the CPU TB state.

Regarding optimizations, the most impactful is the jump slot optimization (TB chaining), labeled as `skipped jump-slot`. The optimization of built-in arguments that do not require normalization (see Section 5.1) also shows notable impact in some benchmarks, such as `mdsum`, and is labeled as `skipped args`.

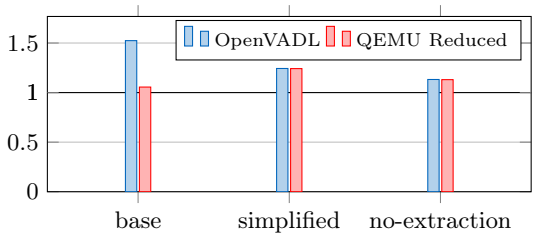

Fig. 8. Speedup of the decoder relative to upstream QEMU (higher is better)

To evaluate the performance of the OpenVADL generated decoder we compare it to the version provided by the upstream QEMU frontend. The evaluation was performed on an AArch64 Apple Silicon system using micro-benchmarks over a uniformly distributed set of RISC-V RV64IM instructions. Since the upstream decoder always includes instructions for additional RSIC-V extensions, the evaluation is additionally against a QEMU generated decoder with an instruction set reduced to the 64 bit IM extension.

The `base` version compares the unmodified decoders as generated by OpenVADL and QEMU respectively. The significant speedup of the VDT can be exclusively explained with the difference in implementations for field extraction upon successful classification of the instruction. Removing costly verification checks in the QEMU version and eliminating constant field extraction (e.g. opcodes) in the VDT implementation results in the benchmark `simplified`. Here the additional overhead for decoding a larger instruction set becomes evident, while showcasing that the structurally equivalent decode trees of the VDT vs. reduced QEMU perform equally well. Finally we compare a version without field extraction (`no-extraction`), to exclusively evaluate the decision procedure.

7 Related Work

Computer architecture simulation is a well researched area. An overview about the methodology as well as a discussion on trade-offs between performance and accuracy is presented in [12] and [13].

Many PDLs are used as input for ISS generators. A comparison of the generated simulators is presented in [5]. QEMU itself is introduced in [3].

The ISS described in [4] uses the LLVM [7] just-in-time (JIT) compiler to generate the code for the host architecture. As opposed to QEMU, the dynamic translation is not restricted to basic blocks but extended to regions consisting of consecutive basic blocks. The simulator is capable of cycle accurate execution. Currently OpenVADL only supports the generation a purely functional ISS. The main drawback of their system is the high compilation time imposed by the LLVM JIT compiler.

CrossDBT is another emulator generator based on the LLVM JIT compiler [8]. An interpreter is used to decode the guest machine code. In contrast to OpenVADL, the processor description is not done in a PDL but in C++. Furthermore, CrossDBT only supports user mode emulation.

Pydgin [9] uses the RPython translation toolchain to generate an ISS from an architecture specification written as Python classes. The intent of RPython is to automatically generate a JIT compiler for dynamic programming languages. Pydgin leverages the RPython generated meta-tracing JIT compiler for its generated ISS, thus achieving similar performance to dedicated DBT frameworks. In contrast to OpenVADL the architecture description is done in a an embedded Python Domain Specific Language (DSL) allowing meta programming capabilities. In OpenVADL meta programming is supported by VADL's macro system.

OpenVADL's decode tree generator is based on [11]. The article describes a technique to generate decode decision trees for mostly regular ISA encodings. While it does have limited support for overlapping instruction patterns in case of subsumed encodings, it is not capable of handling fully irregular patterns. To improve OpenVADL's capabilities of generating efficient decode decision trees for irregular encodings, we introduced some ideas from [10]. The handling of irregular encodings is currently a work in progress.

8 Conclusion

We presented the details of the algorithms to generate the guest specific frontend for a QEMU based instruction set simulator from a concise processor specification in VADL. A detailed evaluation shows the effects of the different optimizations on two different host architectures. The benchmark results show that the OpenVADL-generated QEMU based ISS achieves a speedup of up to 1.77 compared to the official handwritten QEMU frontend for the RISC-V RV64IM instruction set architecture. Finally, OpenVADL is open source and available at openvadl.org.

References

1. perf(1) - Linux manual page, https://man7.org/linux/man-pages/man1/perf.1.html, [Online; accessed 3-March-2025]

2. Embench: A modern embedded benchmark suite (2024), https://www.embench.org/, [Online; accessed 15-January-2024]
3. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference, FREENIX Track. pp. 41–46. USENIX (2005), https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf
4. Brandner, F., Fellnhofer, A., Krall, A., Riegler, D.: Fast and Accurate Simulation using the LLVM Compiler Framework. In: Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO. vol. 9, pp. 1–6 (2009), https://www.complang.tuwien.ac.at/andi/papers/rapido_09.pdf
5. Freitag, F., Halder, L., Himmelbauer, S., Hochrainer, C., Huber, B., Kasper, B., Mischkulnig, N., Nestler, M., Paulweber, P., Per, K., Raschhofer, M., Ripar, A., Schwarzinger, T., Zottele, J., Krall, A.: The Vienna Architecture Description Language (2025). https://doi.org/10.48550/arXiv.2402.09087
6. Freitag, F., Halder, L., Huber, B., Kasper, B., Nestler, M., Per, K., Raschhofer, M., Ripar, A., Zottele, J., Krall, A.: OpenVADL: An open source implementation of the Vienna Architecture Description Language. In: Vialle, S., Suarez, E., Pionteck, T. (eds.) ARCS 2025 – 38th GI/ITG International Conference on Architecture of Computing Systems. Springer Nature Switzerland (2025)
7. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. p. 75. CGO '04, IEEE Computer Society, USA (2004). https://doi.org/10.1109/CGO.2004.1281665
8. Li, W., Luo, X., Zhang, Y., Meng, Q., Ren, F.: CrossDBT: An LLVM-based user-level dynamic binary translation emulator. In: Cano, J., Trinder, P. (eds.) Euro-Par 2022: Parallel Processing. pp. 3–18. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-12597-3_1
9. Lockhart, D., Ilbeyi, B., Batten, C.: Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 256–267. IEEE (2015). https://doi.org/10.1109/ISPASS.2015.7095811
10. Okuda, K., Takeyama, H.: Decision tree generation for decoding irregular instructions. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1592–1597. EDAA (2016), https://past.date-conference.com/proceedings-archive/2016/pdf/0066.pdf
11. Theiling, H.: Generating decision trees for decoding binaries. In: Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems. pp. 112–120. ACM (2001). https://doi.org/10.1145/384198.384213
12. Yi, J.J., Kodakara, S.V., Sendag, R., Lilja, D.J., Hawkins, D.M.: Characterizing and comparing prevailing simulation techniques. In: 11th International Symposium on High-Performance Computer Architecture. pp. 266–277 (2005). https://doi.org/10.1109/HPCA.2005.8
13. Yi, J.J., Lilja, D.J.: Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. IEEE Transactions on Computers **55**(3), 268–280 (2006). https://doi.org/10.1109/TC.2006.44