Synthesizing Instruction Selection Back-Ends from ISA Specifications Made Practical

Florian Drescher
Technical University of Munich
Munich, Germany
florian.drescher@tum.de

Alexis Engelke
Technical University of Munich
Munich, Germany
engelke@in.tum.de

***Abstract*—Instruction selection in compiler backends traditionally depends on huge handwritten rule libraries that map IR patterns to target instruction sequences. Porting to new architectures or extending them for new hardware features is a very error-prone process, results in a significant development effort of a dedicated expert and even requires long-term commitment to apply patches for previously introduced bugs.**

Automatic synthesis of these rule libraries from formal ISA and IR specifications eliminates the initial development effort and reduces the long-term maintenance effort, as synthesized rules are correct by construction. Prior work on synthesizing instruction selectors either requires synthesis times of multiple days or significantly falls behind the code quality of an optimizing handwritten backend – in both cases not applicable in practice.

We introduce a term canonicalization and indexing approach that accelerates finding rules on syntactically-similar bitvector terms while returning to SMT solving to ensure completeness in all other cases. Combined with search bounds derived from LLVM's existing pattern base, this reduces synthesis times from multiple days to under two hours for AArch64 and RISC-V.

We integrated the synthesized instruction selection rules for AArch64 and RISC-V into LLVM's GlobalISel backend and achieved almost on-par performance with the existing, industry-standard code generation backends in LLVM on the SPEC 2017 Integer benchmark suite (within 4% of LLVM GlobalISel).

I. Introduction

Modern Instruction Set Architectures (ISAs), including AArch64 or RISC-V, have grown increasingly complex. ISAs often feature hundreds of complex instructions (e.g., AArch64 vector instructions) or modular extension mechanisms, with RISC-V, in particular, offering a proliferation of custom configurations and vendor-specific extensions. Even mature ISAs like x86-64 and AArch64 continue to evolve, rapidly expanding their instruction sets with new functionality and instructions over time [1], [2].

A central responsibility of any compiler is to translate an intermediate representation (IR) into efficient machine code for the targeted hardware. A key component is the instruction selector, which matches IR constructs to efficient, semantically equivalent, target-specific instruction sequences. For this, instruction selectors rely on typically large rule libraries that map patterns of IR operations to corresponding instruction sequences. In most prevalent general-purpose compiler frameworks (e.g., GCC [3], LLVM [4], and Cranelift [5]), these pattern library is written by hand for all architectures, which is not only a huge effort, but also requires continuous maintenance to support new hardware features and IR operations.

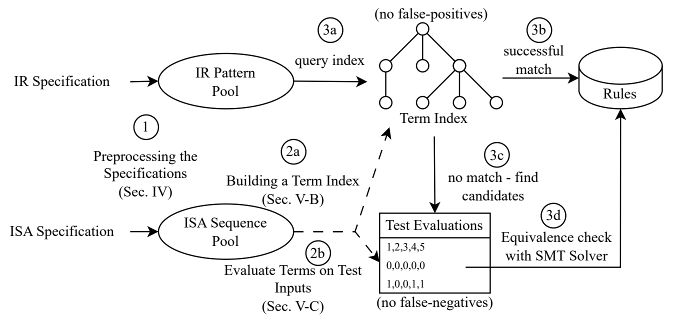
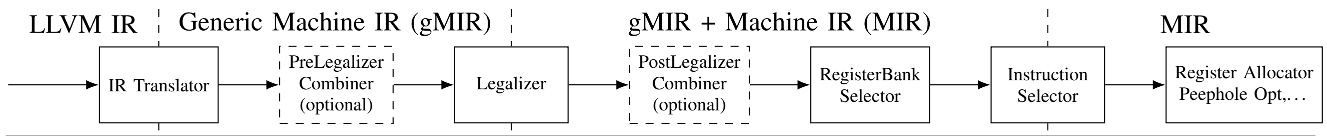


Fig. 1: Our approach to synthesizing an instruction selection rule library involves: preprocessing specifications into pattern pools (1) building a term index (2a) and evaluating the instruction sequences on test inputs (2b) and searching for rules – first try to find a match in the term index (3a) on success emitting the rule directly (3b), on failure evaluate the test inputs and probe against cached evaluations to find candidate matches (3c) and verify with an SMT solver (3d).

Furthermore, it is a major burden for custom, domain-specific code generators focusing on optimized code generation. Only a few big industry players can support such ventures for the JVM [6], [7], .Net [8], JavaScript and WebAssembly [9], [10], [11]. And even here, the adoption of new architectures is limited. JavaScriptCore applies code generation for x86-64 and AArch64, and SpiderMonkey and V8 just announced additional RISC-V support [12], [13].

Recent trends offer a potential way forward: both ARM and RISC-V published formal specifications of their ISAs [14], [15] and others are likely to follow [16]. These structured semantics enable *automatic* synthesis of instruction selection rules – rules that are *correct by construction*, derived from the formal semantics of both the IR of a compiler and the ISA. While replacing handcrafted rule libraries in optimizing compilers entirely may not be realistic, our goal is to provide a foundation that requires only minimal adjustments or customized rules to reach the performance of optimized backends.

There are two prevalent approaches in prior work on automatically synthesizing instruction selection rules: Ideas based on counter-example guided inductive synthesis [17] try to find all minimal IR patterns for each ISA instruction by repeatedly generating candidate matches and verifying them via an SMT solver [18], [19], [20]. Rule-based ideas apply


LLVM IR
Generic Machine IR (gMIR)
gMIR + Machine IR (MIR)
MIR
IR Translator
PreLegalizer Combiner (optional)
Legalizer
PostLegalizer Combiner (optional)
RegisterBank Selector
Instruction Selector
Register Allocator Peephole Opt,...

(I)

```
define i32 @example(i32 %0, i32 %1){
  %2 = shl i32 %1, 4
  %3 = add i32 %2, %0
  ret i32 %3
}
```

(II)

```
%0:_(s32) = COPY $w0
%1:_(s32) = COPY $w1
%2:_(s32) = G_CONSTANT i64 4
%3:_(s32) = G_SHL %1, %2
%4:_(s32) = G_ADD %0, %3
$w0 = COPY %4(s32)
RET_ReallyLR implicit $w0
```

(III)

```
%0:gpr32 = COPY $w0
%1:gpr32 = COPY $w1
%2:gpr32 = ADDWrs %0, %1, 4
$w0 = COPY %2
RET_ReallyLR implicit $w0
```

Fig. 2: Major stages of the LLVM GlobalISel backend pipeline, which is responsible for the transformation from hardware-agnostic LLVM IR to target-specific machine instructions. Example of an LLVM IR function (I) progressively lowered to gMIR (II) and MIR (III). The generic instructions, `G_ADD`, `G_SHL` and `G_CONSTANT`, are combined into a target-dependent instruction — `ADDWrs` (addition with operand shifted by immediate).

algebraic transformations to find matching IR patterns and instruction sequences, applying several heuristics to bound the search space [21]. However, neither approach is practical: one falls significantly behind in performance due to missing rules [21], while the other requires days of synthesis even for small x86_32 subsets [18] and does not scale due to exponential growth with IR instructions. Moreover, by enumerating all minimal patterns for single ISA instructions, they spend most of the time on patterns that rarely occur and are restricted to single-instruction rules.

We combine ideas of both worlds (refer Figure 1): Given a formal ISA and IR specification, we first preprocess these into SMT bitvector terms and derive a set of considered IR patterns and ISA instruction sequences (Section IV). Next, we build an index on a canonicalized form of the bitvector terms for the ISA instruction sequences (Section V-B) and additionally evaluate them on a set of random test inputs and store the result (Section V-C). When searching for rules for an IR pattern, we first query the index for structurally similar bitvector terms and emit the most beneficial rule if there are matching ones. As a fallback, we probe the table of test evaluations to find potential candidate matches and verify them via an SMT solver.

We synthesized an instruction selection rule library for the RISCVg profile (in less than 40 min) and AArch64, including Neon vector instructions (in less than 2 hours) – focusing on integer instructions, leaving out floating point operations. The rule libraries are integrated into the LLVM GlobalISel instruction selection backend and evaluated on the SPEC Integer benchmark. The generated code is almost on par with the optimized code generation in LLVM – on average only 4% slower compared to the current GlobalISel backend – an industry-standard code generation framework handcrafted and tuned over multiple years.

This paper makes the following contributions:

- We formulate the synthesis of an instruction selector as a memoization of the most relevant patterns and their corresponding instruction sequences. This allows focusing on relevant patterns and lifts the restriction of single ISA instructions patterns to also support sequences of ISA instructions.
- We derive general heuristics on what parts of the potentially infinite search space of IR and ISA instructions are actually relevant by analyzing LLVM's production rule library. Based on those observations, the synthesis times are cut from multiple days to few hours without sacrificing code quality of the synthesized rule library – as confirmed by our benchmarks.
- We introduce a canonical representation and term index structure for bitvector terms, accelerating the discovery of equivalent terms and equivalence checking – avoiding expensive SMT queries.
- We synthesize a rule library for RISC-V and AArch64, integrate it into LLVM's GlobalISel infrastructure and show that the synthesized backend achieves performance comparable to LLVM's handwritten AArch64 and RISC-V backends (within 4% on average on SPEC CPU 2017 integer benchmarks compared to current GlobalISel).

Throughout the paper, we will use patterns to refer to IR instruction patterns and instructions or sequences to refer to machine instruction sequences.

II. Background

A. *Satisfiability Modulo Theories*

Satisfiability Modulo Theories (SMT) [22], [23] extends boolean satisfiability problems by arithmetic, uninterpreted functions, and other theories. In particular, the fixed-width bitvector theory allows expressing operations on bit sequences, including a fully defined set of integer operations. SMT solvers can also be used to prove the equivalence between different formulas. A popular SMT solver is Z3 [22], which accepts inputs following the SMT-LIB standard [23]. All formulas in this paper are written in a simplified syntax of SMT-LIB.

B. *LLVM's Global Instruction Selector*

This section outlines the structure of the code generation pipeline in LLVM using the GlobalISel framework – visualized in Figure 2. GlobalISel represents LLVM's most recent instruction selection infrastructure, designed to replace legacy FastISel and SelectionDAG backends.

The GlobalISel pipeline begins by translating LLVM IR, the IR used during the LLVM middle-end optimizations, into Generic Machine IR (gMIR). It is a typed, register-aware representation based on the same data structures as the final Machine IR (MIR). It mainly consists of target-independent instructions, although target-specific MIR instructions are also legal. Fig. 2 shows an example gMIR function (targeting AArch64), which takes two integer parameters, shifts the second one left by 4, adds the first one, and returns the result. It consists of target-independent `G_CONSTANT`, `G_SHL` and `G_ADD` instructions, the target-dependent `RET_ReallyLR` instruction, and `COPY` pseudo-instructions. Values are typed but not yet assigned to a register bank.

Once in gMIR, multiple transformations and progressive lowerings are applied. The target-dependent *legalizer* transforms unsupported operations into target-legal equivalents (e.g., 8-bit arithmetic on AArch64 is rewritten by inserting extension and truncation instructions).

After register banks were selected for each value, the instruction selector replaces generic instructions with concrete machine instructions by applying tree-pattern-matching and rewriting, producing the MIR program shown in Figure 2). Beyond simple one-to-one rewrites, folding multiple gMIR instructions into a single MIR is possible and vice versa. The GlobalISel framework allows declarative specification of rewrite rules in the TableGen language. During instruction selection, it performs a bottom-up traversal, greedily matching the largest tree pattern. Matched gMIR instructions are replaced by their MIR equivalents. For patterns that can not be expressed in the TableGen dialect (e.g., patterns with multiple outputs), selection can fall back to C++ code. However, declarative rules are preferred as they are easier to understand, maintain, and integrate with the rest of the system.

III. Related Work

Previous work on program synthesis can be roughly grouped into two categories: Rule-based approaches apply algebraic transformations and rewrite rules to find and prove the equivalence of ISA to IR terms, while solver-based ones formulate search queries to an SMT solver. The following will only discuss related work on generating instruction selectors.

The predominant approach for program synthesis is counter-example guided inductive synthesis as introduced by [17]. It repeatedly generates candidate programs for a given set of test inputs, verifies the equivalence with a target specification via an SMT solver, and either refines the test inputs by adding a produced counter-example or terminates with a successful match. [18] refined a previous used representation for superoptimization [24], and applied it to sythesize an instruction selector for x86-32. They generate all minimal patterns for each ISA instruction within four days and show comparable performance to the handwritten instruction selection in their compiler. However, they only considered single ISA instructions. More recently, [19], [20] target the generation of a RISC-V instruction selector, explicitly also considering instruction sequences with more than a single instruction. They generate rules for a subset of RISC-V instructions but did not evaluate the runtime of a generated instruction selector.

[21] apply algebraic rewrite rules to iteratively expand a pool of candidate formulas (starting from ISA instructions) until each required IR pattern is matched. They use a cost-based heuristic to prune formulas with high evaluation complexity. Although faster than SMT-based approaches and applicable to x86, ARM, and PowerPC, their method lags behind GCC `-O1` by up to 2× and is inherently limited by the inability to derive certain rules algebraically. VADL [25], [26] introduces an ecosystem for automatically deriving various toolchain components, including instruction selection rules for LLVM. Their approach appears to use simple rule matching to generate a RISC-V rule library, which was only evaluated on microbenchmarks.

To our knowledge, no approach uses the official ISA specifications from the vendors as a starting point; instead, they use custom handcrafted ISA specifications.

Program synthesis is also used for generation of peephole optimization rules [27], [28], [29], searching for rewrite rules within the domain of ISA instructions and superoptimizations [24], [30], [31], synthesizing the optimal instruction sequence for one concrete input program.

There is also some related work in the field of formalizing hardware specifications [32], [33], [34], [35], [36], [37]. While most of them are handwritten and only cover small parts of the microarchitectures, the SAIL ecosystem provides authoritative models based on officially published vendor specifications [38], [39], [40].

IV. Preprocessing Formal Specifications

A. Deriving An Instruction Representation

The formal behavior of ISA instructions was historically provided in the form of manuals and had to be manually extracted into formal models [33], [34], [35], [36]. However, in recent years, vendors have published official specifications of the behavior of their architectures – ARM uses a custom specification language (ASL) [14], and RISC-V provides an official specification in the SAIL language [15], [41].

A widely adopted framework to formally specify, emulate, and verify hardware specifications is SAIL [38]. While the RISC-V specification is already provided in the framework, there are automatic translation tools for ASL to SAIL [38], making it a common hub for formal ISA specifications. Besides other tools, it offers a symbolic execution engine, ISLA [39], based on the formal specifications of an ISA. Although we leverage SAIL for RISC-V and AArch64, the formal specification could also come from other formats that allow transforming the semantics of an instruction into an SMT-LIB bitvector formula.

To obtain a semantic model of each instruction with SAIL, we provide symbolic operands as inputs and leverage the symbolic execution engine to transform each instruction into a set of SMT formulas. An instruction can have different effects on hardware components: it can modify the program counter, set some flags, write to memory, and write to vector or general-purpose registers; each is modeled with a separate bitvector term. Compared to previous representations [24], [18], we do not construct a single SMT formula to encapsulate the whole functionality of an instruction, but instead, view each *effect* of an instruction separately. Symbolic inputs are the PC modeled as a 64-bit value, the condition flags (not relevant on RISC-V) each modeled as a single bit value, and (vector) registers modeled as 64-bit or 128-bit values respectively – subregister operations have to extract from those. There are no additional instruction attributes (e.g., conditional codes for AArch64 branching instructions), but instead, each possible attribute assignment is considered a separate instruction (e.g., branch equal, branch not equal). Refer Listing 3a for a simple example of an AArch64 `add` instruction that can also shift its second register operand by an immediate (`ADDWrs` instruction from Figure 2). It has two symbolic input registers, an immediate, and the register effect describes the computation.

```
xd : (bvadd xn (bvshl xm, ((_ zext 58) imm6)))
```

(a) `add xd, xn, xm, lsl #imm6`

```
memory : (store ((_ extract 31 0) xt) #x4 xn)
```

(b) `str wt, xn`

```
xt : (load #x8 xn)
xn : (bvadd xn ((_ sext 55) imm9))
```

(c) `ldr xt, [xn],#imm9`

Fig. 3: Examples of derived instruction representations on AArch64. An instruction can have multiple effects.

To model memory interactions, we introduce two symbolic functions: load, which returns the value loaded from memory of a certain size, and store, which can only appear (but also has to) as the root operation of a memory effect and stores a value of a certain size into memory (refer Listing 3b and 3c). These memory operations model a relaxed memory model – other memory orderings require additional operations. An instruction can also have multiple effects, even of the same type (e.g., post/pre-index loads on AArch64, refer Listing 3c).

We do not model register constraints (early clobbers, SP/XZR on AArch64) as part of the formalization, as they are not relevant for instruction selection.

After having semantics for the individual instructions, we can compose them into sequences according to the following rules: (1) We only consider sequences where each instruction has a (transitive) impact on the effect of the last one. (2) We do not append to an instruction with a PC effect, as this implies a sequence that spans multiple basic blocks. This restriction simplifies the handling of the formulas. (3) At most one memory operation is allowed within a sequence. Otherwise, matching those sequences would require a sophisticated alias analysis for the operands. This could be slightly lifted to support multiple loads, but we did not see any cases where this was beneficial.

B. Deriving An IR Representation

The formal representation of our IR follows the same rules as the ISA instruction specifications – both share a common representation. In the case of LLVM, the input representation to the instruction selector are gMIR instructions. As those currently do not have a formal specification available and their semantics are primarily described in documentation or code, we manually defined symbolic specifications for a large subset of gMIR instructions. In principle, these specifications could be derived automatically – through symbolic execution or by translating existing lowering rules into a symbolic form. Automating this process is an important direction for future work, but it is not within the scope of this paper.

As in the case of ISA instructions, IR patterns can be composed into larger ones, allowing for the formation of arbitrary patterns by composing the formal representation of multiple gMIR instructions. While the number of possible patterns to consider is huge, one would ideally only focus on the most beneficial patterns – those that are preferable over combinations of their subpatterns. However, this is impractical, as their usefulness can not be determined upfront but can only be evaluated post-synthesis. As a reasonable alternative, we use most frequently occurring patterns (refer Section VII-B).

V. Matching IR Patterns and ISA Sequences

Given a pattern, we want to find a semantic equivalent instruction sequence from a pool of potential sequences. For simple cases, this comes down to checking whether two symbolic expressions are equal – a well-studied problem. However, synthesizing an effective and efficient instruction selection rule library comes with some additional pitfalls and challenges, some of which have received surprisingly little attention in the literature so far.

A. Challenges

1) Exhaustively searching: Searching all instruction sequences and checking them via SMT solvers is infeasible. Already for RISC-V's 40 base instructions and over 200 gMIR operations, exploring all combinations of IR patterns with up to 5 operations and instruction sequences of up to 2 instructions already results in $> 5 \cdot 10^{14}$ SMT queries. The complexity grows exponentially for larger ISAs like AArch64 (>150 integer instructions) and larger IR patterns. Hence, effective heuristics [21], [20] are essential to prune the search space and make equivalence checking tractable. Most prior work [18], [21] synthesized all possible IR patterns for isolated ISA instructions, not only resulting in a still rather large search space, but also yielding patterns that are unlikely to be used and unable to identify multi-ISA-instruction patterns.

We address this problem by first looking for structurally equivalent terms in an index (Sec. V-B), avoiding expensive SMT solver invocations through evaluations on random data (Sec. V-C), and pruning the search space utilizing information about frequently occurring IR patterns (Sec. VII-B).

2) Immediate encodings: Immediates pose another challenge, especially in fixed-width ISAs. While IR operations work with abstract bit-widths (e.g., 64-bit addition), hardware instructions often impose restrictive and non-uniform immediate encodings (e.g., 9-bit signed values). Consequently, rather than testing whether a pattern in the instruction sequence candidate pair is strictly equal, instruction selection must determine whether they can be made equal under certain constraints `a,b : BitVec`$_{16}$

I	`(bvadd a (bvnot b) #x0001)`
II	`(bvadd a (bvmul #xffff b))`
III	a $+_{16}$ -1b

Fig. 4: Two syntactically-different bitvector terms for subtraction of two 16-bit values (I, II) and their common canonicalized representation, where $+_{16}$ denotes 16-bit addition.

— most notably on the values or encodings of immediates, but can also involve alignment or PC-relative addressing. This can lead to multiple instruction sequences for the same pattern with different constraints. For example, materializing a constant on AArch64 may require 1–4 `mov` instructions depending on the immediate value, while different load and store instructions must be used based on bit width, signedness, and offset alignment. To our knowledge, no prior work has identified this issue; instead, existing approaches abstract over immediates, requiring manual effort and ignoring the constraints during synthesis [18].

We handle immediates specially for index lookups (Sec. V-B3) and when querying the SMT solver, we apply heuristics to transform immediates in the instruction representation (e.g., zero-/sign-extend) (Sec. V-C).

3) Instruction costs: Often, there are multiple different ways to encode the same operation, and in some cases, shorter instruction sequences can actually be less efficient than a sequence of more instructions. Coming up with a suitable cost model that encodes these micro-architectural differences and prefers architectural canonical representations is hard.

We currently use the total number of input operands that occur in all instructions of a sequence as a very simple cost metric, which has worked sufficiently well. Using other properties (e.g., instruction latencies, instruction count) as part of the cost model may result in improved code; this is left as future work.

B. Efficiently Querying Structurally Similar Terms

Equivalent bitvector terms can be formulated in numerous syntactically different terms, making equivalence checking of two bitvector formulas NP-hard [42]. However, we observed that handwritten IR descriptions and the symbolic execution of the imperative SAIL specifications for semantically equivalent terms share a similar syntactical structure. Nonetheless, terms as simple as subtraction of two values might still be represented slightly differently between the IR specification and the ISA instruction formula (refer to Figure 4). We generally assume that strength reduction, e.g. converting multiplications to shifts, has already happened during IR optimization.

1) Canonical representation: To account for those differences, we propose a canonical representation for each bitvector term, which consequently allows efficient equivalence checking of two terms using structural unification. The following property regarding the canonical representations of two terms should hold: If the canonical representations are equivalent, the terms are equivalent – two semantically equivalent terms do not necessarily have to end up with the same representation – in contrast to [42], who use a similar canonical representation for *exact* equivalence checking on a very reduced set of operations.

TABLE I: Canonicalization transformations where $+_n$ denotes addition module 2^n, val_n a constant bitvector of width n, a_n and b_n denotes arbitrary bitvector terms of width n.

(I)	(bvadd a_n b_n)	$\rightarrow \mathrm{a}_n +_n \mathrm{b}_n$
(II)	(bvnot a_n)	$\rightarrow -1 +_n -1\mathrm{a}_n$
(III)	(concat a_n b_m), $k = m + n$	$\rightarrow 2^m a_n +_k b_n +_k$ $-2^m(\mathrm{extract}_{k:m}\ b_n)$
(IV)	(bvmul a_n (b_n $+_n$ ... c_n))	$\rightarrow$ (bvmul a_n b_n) $+_n$ ... $+_n$ (bvmul a_n c_n))
(V)	(bvmul a_n val_n)	$\rightarrow \mathrm{val}\cdot\ \mathrm{a}_n$
(VI)	(bvshl a_n val_n)	$\rightarrow 2^{val}\mathrm{a}_n$
(VII)	(bvurem a_n val_n), $\mathrm{val} = 2^x$	$\rightarrow (\mathrm{extract}_{x-1}\ a_n)$
(VIII)	(ite a_1 0 b_n)	$\rightarrow$ (ite ($+_1$ a_1 1) b_n 0)
(IX)	(ite a_1 (... $+_n$ b_n) (... $+_n$ b_n))	$\rightarrow \mathrm{b}_n +_n$ (ite a_1 $(\ldots)$ $(\ldots)$)

Based on the idea of [42], our canonical representation consists of *terms* and *atoms*. Each term has a coefficient, an operation (e.g., bitvector addition, multiplication, ite), and operands which are themselves canonicalized terms or atoms. Atoms are symbolic variables with a coefficient and, in our case, also carry additional domain information (e.g., (vector) register or immediate). This representation is different from the SMT bitvector formulas: (1) We introduce coefficients and implicitly zero-extend terms and atoms when applied to an operation with higher bitwidth (simplifies unification later on). (2) Besides existing arithmetic and bitwise operations, we introduce additional operations based on the idea of [19] to model complex operations as functions. Those are load, store (mentioned in IV), popcount, count leading zeros and count trailing zeros. (3) Boolean values are treated like bitvectors of length 1 – modeling integer comparisons as a sequence of bitvector arithmetic and conditionals. (4) To disambiguate, we try to transform bitvector formulas into linear combinations (addition module 2^n for some fixed n) when possible (rules shown in Table I). For operations not mentioned in the table, the operation of the term remains, and we recursively canonicalize the operands. Note that due to implicit zero-extension, bitvector addition on different bitwidths is not associative and requires the introduction of overflow terms in Rule (III). In addition to the transformations, we apply simplifications with constant terms. The canonical representation of a subtraction, shown in Figure 4, is the same for both initial formulas. The first formula is transformed with rules (II) and (I); the second one with rules (V) and (I).

By imposing some total order on the operands of commutative operations (e.g., ordered by subterm type: constants, variables, and within the subterm type by their term id, which is discussed in the following), the canonical representation allows structural unification and equivalence checking.

2) Efficient Querying: For efficient querying in a set of canonicalized terms and deduplication, we store them in a trie (refer Figure 5). Each path from the root to a node corresponds to a modulo-n linear combination, with the individual operands on the edges – each node represents one term. The terms along the edges could be atoms or (sub)terms themselves. In the

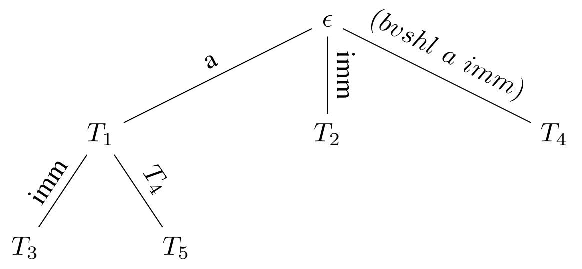


Fig. 5: The AArch64 term trie for the terms: $a + (bvshl\ a\ imm)$ (add with shifted operand), $a + imm$ (add with immediate), and their respective subterms. Nodes correspond to canonicalized terms. Edges correspond to atoms or (sub)terms, which are recursively stored in the tree. Coefficients along the edges are left out for simplicity (all set to 1).

case of a subterm, it is recursively stored in the trie as well – as a leaf on depth one: seen as linear combination with a single operand. Operands of that term are recursively stored in the trie. Each term is assigned a unique identifier t when inserted, corresponding to the number of elements in the trie when it was added (an increasing numbering of terms). This implies that two canonicalized terms in the trie are equal iff their identifiers are the same.

Canonicalization of any term happens recursively from the inside to the outside. This implies that when we canonicalize a term, all its operands have been canonicalized already. Inserting atoms (or any other operations that are not a linear combination) into the trie as the base of the recursion is done by inserting a leaf at depth 1, if it does not already exist. When we insert a linear combination, we have to walk the trie from the root to the leaf. At each node, we can continue with the next node in average constant time due to the usage of a hash-map, as we do not have to recursively compare subterms again but can always use their term ids. This allows insertions of each term in $\mathcal{O}(len)$, where len is the length of the canonicalized term, as each part of the term is only inserted once and afterward referred to via its id.

This part only covers the insertion into the trie so far, but there are additional costs for the canonicalization of a term. During canonicalization, we try to construct modulo-n additions by incrementally adding new addends. To correctly handle coefficients, we need an efficient way to check whether a term already exists in the list of addends, which we do with an ordered map based on the ids of the terms. This leads us to a total runtime complexity of canonicalization and insertion into the trie of $\mathcal{O}(n \log n)$, where n is the length of the canonicalized term which (apart from multiplications where the canonicalized representation can be quadratic in the size of the input) is in the order of the input length of a term. As a result, we can check a newly arriving term in roughly $\mathcal{O}(n \log n)$, where n denotes the size of a term, for equivalence against an arbitrarily large set of bitvector terms.

While this equivalence checking is useful during the construction of the trie for deduplication of subterms and numbering the terms, matching an IR pattern against a set of ISA instruction sequence terms requires unification instead of equivalence checking. Given a query pattern, we traverse the trie to retrieve all structurally matching candidates via a unification procedure with backtracking. Unlike pure lookup, unification must consider all successors of a node with a matching root term and, in case, recursively run unification on the subterms. During the unification, we greedily unify free variables in the query pattern with free variables along the search path.

3) Immediate Handling: In the unification, we only unify (vector) registers and immediates with each other, unify *PC + imm* with a single immediate in search patterns (pc-relative addressing), allow the unification of immediates even with different coefficients (as potential alignment/scaling constraints to be materialized into the instruction selection rules), bind excess constants in queries to immediates and bind excess immediates in the trie to zero. For example, the term $x + imm$ could be unified with $a + imm$, as x unifies with a and the immediates unify. Furthermore, the terms x could be unified with its one-length counterpart a and additionally with $x + imm$ as we can bind the excess imm to zero.

Referring to our original example, the canonicalized pattern of add with shifted operand looks like $(bvshl\ x\ (extract_{5:0}\ imm) + y)$ (shift distance modulo bit width) and can be unified with T_5 by unifying $(extract_{5:0}\ imm)$ with imm. Although the terms are technically not structurally equivalent, we allow binding two immediates with different bitwidth, extensions, or extracts; such immediate constraints are later emitted during rule generation (Sec. VI-A). Once the pattern matches in the trie, the term id retrieves the corresponding ISA instruction unless it is an auxiliary subterm without a corresponding instruction sequence. In our concrete example, T_5 would correspond to the `ADDXrs` instruction seen previously.

C. SMT Solver Fallback

As our index can have false negative matches, meaning that no matching instruction sequence is found for a requested pattern, although there is one, we additionally want to fall back to an SMT solver in those cases. To restrict the number of possible instruction candidates, we eliminate potential sequences with simple heuristics, which worked well in practice: The number of register inputs and immediates and the number and size of loads and store terms must match.

To further bound the number of checked instruction sequences, we evaluate the bitvector formulas for a set of fixed, randomly-generated inputs during creation of the ISA sequence pool and cache the results. When searching for potential candidates, we evaluate the searched pattern on the same set of inputs and compare it with the outputs of potential instruction sequences to quickly reject non-matching sequences. In cases where an input value cannot be represented in an immediate binding, we ignore the test input. The sample evaluation cannot produce false negatives; false positives are identified by the SMT solver.

When querying the SMT solver for equivalence of two terms, we have to provide a binding of IR variables to ISA variables. We, therefore, enumerate all possible assignments of input variables and immediates from the IR inputs to the instruction parameters, repeating the test evaluation and SMT solving for each possible assignment. In practice, the number of combinations remains manageable, as most instructions and patterns involve less than four inputs. When binding a larger immediate in the IR formula to a smaller ISA immediate, we additionally have to decide upon the used extension for the smaller one. We use a simple heuristic: if the sign bit of an immediate is accessed more than five times in the formula, we assume it is sign-extended. Despite its simplicity, this heuristic has proven reliable in all tested cases and, if incorrect, would only imply missing a pattern – no effect on correctness.

Finally, we impose a 500ms timeout per query and stop upon finding a matching instruction sequence for a pattern.

D. Limitations

While our approach enables the automatic generation of a large number of instruction selection rules, it comes with a few limitations.

1) Non-trivial immediate encoding: Our method assumes that immediates appear explicitly as whole values in the instruction term. This prevents us from matching patterns where the immediate needs to be transformed for the encoding.

Examples on AArch64 include logical operations (`and`/`orr`/`eor`), the shift distance in vector shift instructions, and the instruction to encode negative immediates (`movn`).

For these cases, handling is possible by manually adjusting the SAIL representation to replace the complex expression with a single explicit auxiliary immediate. This allows successful matching but implies manual handling when emitting the instruction to re-encode the immediate correctly into the instruction. However, these special handling for such immediates only affect a small number of instructions and often exist in compilers/assemblers anyway. Without manual intervention, instructions with non-trivial encodings can currently not be synthesized by our approach.

2) Undefined IR Behavior: Some IR operations have a limited definition range, e.g., division by zero or out-of-range shifts typically produce unspecified results. Currently, the index-based matcher only checks for strict semantic equivalence. While the SMT solver could, in principle, support such constraints, our current implementation does not attempt to derive or utilize them.

3) Heuristics for termination: To keep the process tractable, we impose strict limits on the pattern search (e.g., length of patterns/instruction sequences, SMT solver time). While these heuristics are essential for termination and scalability, they might lead to missing better matches or fail to match some valid instruction sequences altogether.

4) Unsupported operations: We did not consider floating-point operations in this work, as bitwise SMT formulas would be highly complex without yielding additional optimization opportunities. We also exclude certain complex instructions (e.g., division, cryptographic operations) either due to incomplete support in ISLA or because symbolic execution failed to complete within reasonable time bounds (1 hour).

Additionally, atomic operations and specifically aligned memory accesses are currently not included. Support for specific memory operations with different memory orderings and alignments or floating-point arithmetic can be easily added based on the proposal by [19] to introduce additional symbolic functions for, e.g., IEEE 754 floating-point operations or loading according to total-store-order.

VI. Generating an Instruction Selector

We consider the instruction selection procedure as repeatedly querying a pool of possible instruction sequences to find the best-matching sequence for a given IR pattern. To make this practical and shippable in a compiler, we interpret the instruction selection rules as memoizing the most requested and beneficial mappings.

We query patterns for the most frequently used IR patterns (refer Section VII-B) against our pool of ISA sequences, for each newly discovered instruction sequence of a larger pattern, we verify that it is cheaper than the combination single-IR-operation patterns to ensure that the synthesized rule library contains only meaningful matches, and add emit the corresponding rule into the output.

A. LLVM Integration

The synthesis stages are independent, and the instruction pool can be persisted after construction. Since the first stage does not depend on the IR patterns, it can be reused across different IRs when synthesizing instruction selectors.

```
def : GeneratedPattern<
  (i64 add GPR64:$a, (i64 shl GPR64:$b, imm:$shift)),
  (ADDXrs GPR64:$a, GPR64:$b, extract0:5:$shift)>;
```

Listing 1: Generated AArch64 TableGen pattern for the shift-and-add instruction of a 64-bit register with a 6-bit immediate.

The framework outputs synthesized instruction selection rules in LLVM's TableGen language. The example rule for transforming an addition with a shift into an `ADDXrs` instruction can be seen in Listing 1. Besides the structured representation of the rewrite rule we additionally embed the identified constraints. In this case, we only use the lower 6 bits of the shifted immediate – indicated by the immediate type.

These generated files include both the pattern definitions and the corresponding instruction descriptions. LLVM then uses these definitions to import the rule library into LLVM's GlobalISel backend and uses them in the greedy matching during instruction selection. To ensure completeness, especially for operations not covered by our synthesis, we manually import missing patterns for unsupported operations (e.g., floating-point arithmetic, atomics) from LLVM's existing handwritten rules, ensuring that none of these shadow a generated pattern.

We primarily target the LLVM's new GlobalISel infrastructure, which is already the default for AArch64 `-O0`, due to its modularity and extensibility. With some minimal adjustments to the IR specification and the generated TableGen output, we can also synthesize rules for the older SelectionDAG backend, which works on a different intermediate representation with different legal types and a few different operations. However, the SelectionDAG backend is a lot less modular, with large parts of the lowering being implemented in a single, large C++ component, and legalization often already generates target instructions. This makes evaluation of the synthesized rule library in the context of SelectionDAG very difficult.

There are also some restrictions stemming from the use of TableGen. We extended TableGen to support instruction chains – sequences connected via flags (e.g., `cmp` and `cset`). However, there are still patterns that our framework can discover but that TableGen cannot express – most notably patterns with multiple outputs or those that introduce new basic blocks. LLVM implements those patterns in C++ instead. Besides, we must ensure forwarding or generation of meta-information (e.g., register banks, symbolic operands for relocation generation) so that succeeding passes can do their job.

B. Retargeting for Different Compilers and Architectures

Although our integration and initial motivation stem from LLVM's code generation framework, the synthesis of the rule library is fundamentally agnostic to the input IR. Our approach can derive instruction selection rules for arbitrary IRs, provided a formal specification is available — SSA-form is not required. This makes it applicable to GCC's RTL [3], Cranelift's CLIF IR [5], or domain-specific code generators in JIT compilers [11], [9], [10] or databases [43], [44].

Using our approach with a different compiler requires: (1) A formal specification of the input IR to instruction selection, if it does not already exist. Prior verification efforts [45], [46] or existing documentation can often serve as a starting point. (2) Adapting the output format of the synthesized rules to the compiler's instruction selection framework. Cranelift's rule language is structurally similar to LLVM TableGen and could be targeted with only minor adjustments to the current text output, whereas GCC's more complex and less structured RTL requires greater effort. Alternatively, rules can also be directly emitted as source code, e.g., for domain-specific compilers without a dedicated rewrite language. In either case, an expert in the respective framework should have little difficulty formalizing this step.

Porting our approach to a new architecture requires a formal specification of the ISA instructions as bitvector terms. This can be obtained from a SAIL specification via symbolic execution, through other toolchains, or written by hand.

Note that this process is limited to synthesizing instruction selection rules; a complete backend still requires additional components. For example, to create a new LLVM backend from scratch for a new architecture, the other stages, in particular legalization, register allocation, call and function lowerings, and metadata, number of registers, register banks, register classes, must be adjusted manually. Some of this information could be derived automatically from specifications, but this lies outside the scope of this work.

VII. Synthesis Tuning

Our synthesis framework has several hyperparameters that impact the quality of our generated rule library and the synthesis times. The following sections will discuss the most important ones and explain our choices. We synthesized two LLVM backends on an Intel Xeon Gold 6430 with 32 cores and 256GB RAM using Z3 as SMT Solver.

AArch64: We synthesize a backend targeting the AArch64 instruction set, including core and vector instructions (subject to ISLA's limitations as discussed in Section IV).

RISC-V: We synthesize backends for the RISC-V base ISA (rv64imafd). To generate reasonable patterns for 32-bit arithmetic patterns, we extended the searched instruction pool by appending zero-extension instructions to arithmetic operations.

A. Pattern and Instruction Sequence Sizes

One major challenge is the vast search space of possible pattern and instruction sequence combinations. To better guide synthesis, we analyze the existing LLVM rule library to understand which IR patterns and instruction sequences are actually used in practice. We exclude IR pseudo-operations like `COPY` and instructions we currently do not support, like floating-point arithmetic.

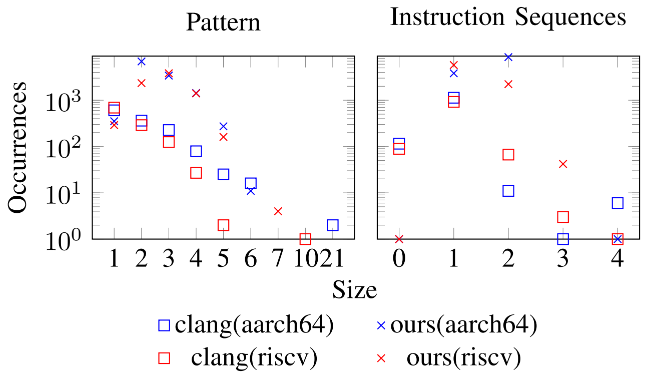


Fig. 6: Pattern and instruction sequence lengths compared between LLVM and our generated pattern set.

Figure 6 shows that nearly all instruction sequences in LLVM TableGen rules are two instructions or fewer. AArch64 includes a few longer sequences (up to four) for materializing 64-bit immediates, and RISC-V occasionally uses three for zero-extended arithmetic. From this, we conclude that limiting instruction sequences to a maximum length of two is sufficient. However, to support 64-bit immediate materialization, we added length-4 sequences where the sum of encoded immediate bits is 64, and for better 32-bit arithmetic on RISC-V, we included sequences with prepended zero extension (length 3).

Most LLVM patterns contain six or fewer gMIR instructions, so we set the pattern size limit to six.

Our generated rule libraries closely match LLVM's distribution, though we miss pseudo-instructions (no instruction sequences of length zero) and generate more rules, e.g., for AArch64 immediates and RISC-V 32-bit arithmetic.

The rule libraries for x86 and PowerPC have similar characteristics in instruction sequence length and pattern sizes. Therefore, these observations should hold as well for new ISAs with similar characteristics.

B. Source of IR Patterns

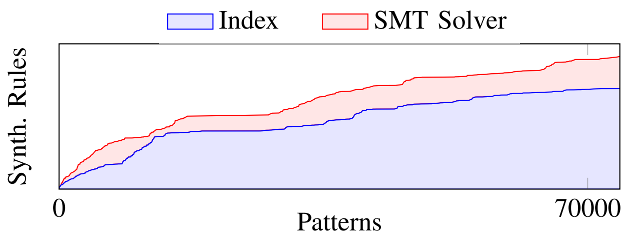


Fig. 7: Synthesized rules based on the number of considered patterns on AArch64 – split between rules found by the term index and rules from the SMT Solver fallback.

Another parameter is the source and number of IR patterns for which rules are generated. We extract these by running LLVM's code generation on the CTMark suite [47], a collection of real-world applications used for testing in LLVM, and tracking all instruction trees up to a depth 6. Figure 7 shows the number of patterns that are generated with an increasing number of considered pattern trees. With an increasing number of considered patterns, the newly discovered rules decline. We decided to cut off the search after 5000 pattern skeletons (turned into 70000 patterns by enumerating feasible types) as the number of newly discovered rules stagnated.

C. Test Input Evaluation

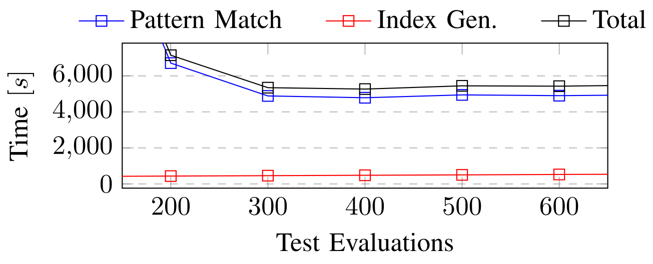


Fig. 8: Comparison of the synthesis time for different numbers of test input evaluations when falling back to the SMT solver when targeting AArch64.

The number of test inputs used to determine potential matching candidates, which are fed into Z3, affects both performance and accuracy: more inputs reduce false positives but increase evaluation time.

Figure 8 shows that pattern matching times remain mostly constant after reaching a minimal number of test evaluations, while index generation time steadily increases. The total synthesis time is minimal at around 400 test inputs, which we use for subsequent experiments.

D. Synthesis Time

Synthesizing an instruction selection backend should be tractable in practice. Previous work has reported runtimes ranging from several hours to multiple days – even for relatively simple instruction sets [18]. Our approach significantly reduces synthesis time: we generate a complete rule library for RISC-V in under 40 minutes and for AArch64 in under 2 hours. A detailed breakdown of AArch64 synthesis is shown in Table II. The final matching stage runs on 60 threads, and reported times show wall-clock duration; subcomponent times represent the average cpu-time per threads. RISC-V follows a similar pattern, with a shorter total runtime (36 minutes) and less generated rules (8000 vs. over 12000 for AArch64).

The results demonstrate that our indexing strategy is effective – 65% of the patterns are synthesized via the index, with minimal time spent compared to SMT fallback. Canonicalization and index insertion are also efficient, while the majority of the runtime is spent in the SMT solver, confirming the index impact in reducing expensive equivalence checks. Without the index, the total synthesis time would double, due to more sample evaluations; the number of SMT solver invocations would only increase by $\sim 30\%$. Without sample evaluation, synthesis did not terminate within 5 days.

TABLE II: Synthesis time for the AArch64 backend. Top-level stages report wall-clock time, while individual components average CPU time per thread.

Instruction Generation	*2122445 instr. seq.*		10.3min
Canonicalize		5.0min	
SMT Test Eval.		3.6min	
Index Insert		21s	
Pattern Generation	*76225 Patterns*		29.3min
Canonicalize		15.0min	
Lookup (parallel)	*12448 Rules*		1.4h
Index Lookup	*8181 Rules*	0.4min	
SMT Test Eval.		0.5min	
Z3 Time	*4267 Rules*	1h	
Total			2h

VIII. Evaluation

While the previous sections evaluated the synthesis time itself, we will next evaluate the quality of the code that is generated with our synthesized instruction selector.

For RISC-V evaluation, we used a Milk-V board equipped with a SOPHON SG2042 processor supporting the RV64IMAFD instruction set and 128GB of RAM. The AArch64 evaluation was performed on an Apple M2 system with 16 GiB of RAM. Both platforms were running Linux and used the *LLVM-20* stable release to integrate the synthesized instruction selection rules.

All benchmarks use the SPEC CPU 2017 Integer benchmark suite [48]. This suite provides a representative and diverse set of complex, real-world programs written in C, C++, and Fortran, making it a suitable target for evaluating instruction selection quality. We compiled each benchmark with Clang and varying backends using `-O2` middle-end optimizations.

A. gMIR Coverage

In addition to the synthesized patterns, we manually specified a few patterns to successfully compile programs (e.g., traps and divisions). Although our approach exclusively generates TableGen patterns, there are still C++ fallbacks for specific instructions like `GLOBAL` symbols, which require more elaborate handling that is not expressible in TableGen. Nevertheless, compared to existing backends our approach has significantly fewer selections that require C++.

TABLE III: GlobalISel Fallbacks

	Functions	aarch64		riscv	
		clang	ours	clang	ours
600.perlbench	7182	3	6	0	0
602.gcc	33159	20	33	0	0
605.mcf	116	0	0	0	0
620.omnetpp	17507	1	1	0	0
623.xalancbmk	35200	4	10	0	0
625.x264	2665	5	45	1	1
631.deepsjeng	308	0	0	0	0
641.leela	1010	0	0	0	0
657.xz	1067	0	0	0	0

To quantify the coverage of our synthesized patterns, Table III lists the GlobalISel fallbacks – cases in which the compilation had to fall back to SelectionDAG for that function. Note that even the baseline has some fallbacks due to missing IR patterns (e.g., missing operations on vector of pointers) and more structural reasons (e.g., failed legalization of phi instruction with certain vector operands).

On AArch64, we have only slightly more fallbacks than the baseline. In those cases, ISLA was not able to formalize the required ISA instructions (e.g., vectorized comparison for 8-bit scalars across 16 lanes) within an hour. However, those fallbacks have no impact on the execution time evaluation, as less than 1% of execution time is spent in those functions. Only in *x264*, our approach falls back in one of the core functions ($\approx$ 35% of execution time) due to a signed-less-than vector-comparison across 8 lanes, which requires an instruction that is not available due to an ISLA timeout.

On RISC-V, there are close to no fallbacks with both the current LLVM GlobalISel backend and our synthesized rule library. This shows that our synthesized backend covers all occurring gMIR patterns.

B. Pattern Coverage

Next, we identify rules in the existing LLVM GlobalISel backend, which are currently expressed in C++ coding, while our approach synthesizes declarative rules for them. We facilitate that by automatically generating test cases from our synthesized rule library and inspecting whether the input gMIR can be selected without falling back to C++.

We generated approximately 9100 test cases (6800 AArch64 and 2300 RISC-V). Nearly 5000 test cases required a fallback to C++ code to complete instruction selection, while our approach can handle all cases without falling back to C++. This demonstrates that a large portion of current backend functionality implemented imperatively, can, in fact, be expressed declaratively and thus made easier to work with.

C. Quality of Generated Code

To assess the quality of the generated binary code from our synthesized backend, we compare the runtime of the SPEC benchmarks with the current LLVM GlobalISel implementation, the FastISel backend, which is used for fast code generation at the cost of slightly worse binary code (default for O0) and the SelectionDAG backend, which is the most involved and stable one, which is used as the fallback for the other two and full optimized code generation (default for O2).

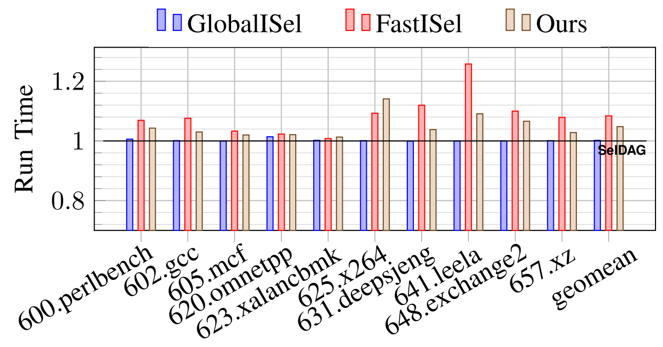


Fig. 9: Runtime of SPEC Benchmarks on AArch64 when compiled with different LLVM backends – normalized to the runtime of SelectionDAG.

Figure 9 depicts the runtime normalized to the SelectionDAG runtime on AArch64. We are on average less than 4% slower than GlobalISel and 5% than SelectionDAG but consistently faster than naive FastISel; apart from *x264* where FastISel falls back to SelectionDAG for all relevant functions.

In addition to the runtime, we also measured the size of the resulting binaries. The binary size of all benchmarks increases by roughly 5%, matching the result of the runtime evaluation.

There are three main reasons for the introduced runtime and binary size overhead: While we support various immediate encodings (16-, 32-, 48-, and 64-bit), we fall short of LLVM's sophisticated constant materialization. For instance, we emit a 4-instruction sequence for a 64-bit constant that could be encoded with a single instruction when only the upper 16 bits are set. More advanced, LLVM turns the addition with a 32 bit constant into two additions with shiften immediates (if possible), while we first materialize the constant and afterwards emit the addition. Both cases could be addressed with C++ implementations or more specific constant patterns.

Although our backend is compatible with LLVM's pipeline, it lacks deep integration with optimization passes. LLVM deliberately misses some advanced patterns during instruction selection (e.g., aggressive address mode folding) to enable more effective subsequent optimizations like load-store optimizations or sinking/folding, whereas we apply more foldings directly. When disabling architecture-specific passes, our backend degrades by 4–6%, while LLVM itself degrades by 5–14%, indicating the importance of tight integration.

Greedy selection requires more care than simply favoring large rules: A severe case adjusted from the *leela* benchmark is shown in Figure 10. Consider having rules for zero-extension with select and select with comparison (both clearly beneficial over the sum of partial covers) in addition to rules for each gMIR instruction. Pattern application starting at `%x0` and `%w0`, applies the respectively largest rules – zero-extension and the select, select and comparison. Finally, we emit code for the comparison (as still required by our first matching). Note that although it is a possible solution in this example, larger rules in general are not – independent of how large the rules are; there could always be another covering that would be more beneficial. LLVM solves the issue with C++ coding and multiple passes to clean such artifacts. This highlights that a good rule library alone is not sufficient for optimal instruction selection but also requires more coordination on what rules should be actually generated – this poses an interesting challenge for future work.

```
-- ins: %x10, $x11, $w1, $w2 | outs: $w0, $x0 --
%x12:_(s1) = G_ICMP intpred(eq), %x10:_, %$x11:_
%w0:_(s32) = G_SELECT %x12:_(s1), %w1:_, %w2:_
%x0:_(s64) = G_ZEXT %w0:_(32)
```

GlobalISel:

```
cmp     x10, x11
csel    x0, w1, w2, eq
```

Ours:

```
cmp     x10, x11
cset    x12, eq
cmp x10, x11
csel w0, w1, w2, eq
cmp x12, #0
csel x0, w1, w2, ne
```

Fig. 10: Different assembly output due to greedy matching.

Figure 11 shows normalized runtimes on RISC-V (there is no FastISel backend for RISC-V). Our synthesized backend performs comparably with an average slowdown of 2% compared to GlobalISel and produces binaries that are 3% larger on average. Performance varies more than on AArch64, reflecting the active development of the RISC-V backend; in particular GlobalISel lacking behind SelectionDAG performance with the exception of *leela*, where GlobalISel and ours significantly outperform SelectionDAG despite executing more instructions. This suggests that improving the cost model beyond simple instruction counts could yield further gains.

IX. Discussion

Previous work [21] synthesized rules for 32-bit x86, PowerPC, and ARM, but the resulting instruction selectors lagged 11–65% behind GCC `-O1`. Others [18] required over 100 hours to synthesize rules for just 20 x86-32 instructions (basic arithmetic, mov, control-flow) — a tiny fraction of the AArch64 ISA. With our approach, generating an x86-32 rule library from their simplified formal specification takes under 5 minutes, though yielding fewer rules ($\sim$1300) since many discovered patterns rarely occur in practice and much of the search space cannot be represented with their limited instruction subset (e.g., no multiplication, no 64-bit arithmetic). As their runtime grows exponentially with the number of IR instructions, the approach clearly does not scale to the complexity of AArch64 and hundreds of gMIR operations, especially given its reliance on simplified handwritten ISA specifications rather than today's authoritative vendor specifications. Thus, automatically generating an instruction selector for large parts of AArch64 approaching the performance of handwritten backends was previously out of reach.

We were able to effectively prune the search space with derived heuristics based on prior usage statistics, an efficient term-index structure, and our approach of viewing instruction selection as memoization of frequent patterns. Our approach synthesizes a correct-by-construction rule library in under two hours, achieving performance within roughly 5% of LLVM, a

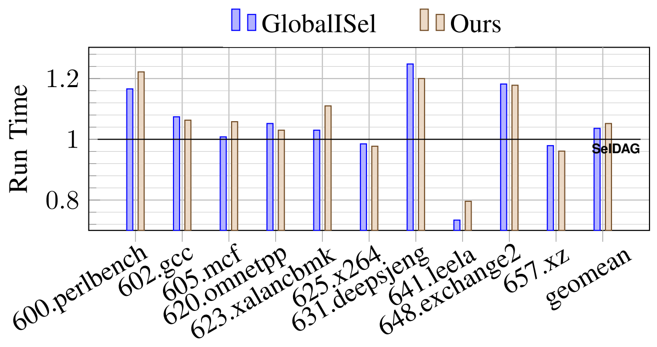


Fig. 11: Runtime of SPEC Benchmarks on RISC-V when compiled with different LLVM backends – normalized to the runtime of SelectionDAG.

production-grade framework refined over many years. The same holds for RISC-V, where we are similarly close performance-wise, and the reduced binary size shows that we indeed find complex and desirable patterns.

Crucially, our results show that much of instruction selection logic can be expressed declaratively, reducing engineering and maintenance effort compared to LLVM's scattered C++ implementations. While LLVM is unlikely to replace its mature, highly optimized rule libraries, our results demonstrate that it is feasible to significantly reduce the manual effort required to create a handwritten rule library. Achieving within 4% of LLVM's GlobalISel performance makes our approach a practical and efficient starting point for further tuning – cutting the total effort of multiple person-years typically required for backend creation and tuning.

X. Summary & Outlook

We presented an approach to instruction selector synthesis based on the memoization of the most relevant IR patterns and their corresponding instruction sequences. Our approach combines efficient term indexing with SMT solving and supports real-world ISAs, like AArch64 and RISC-V.

We integrated the synthesized rule libraries into LLVM's GlobalISel framework, achieving almost on-par performance for the resulting code – guiding an effective way to reduce the effort in crafting and maintaining huge sets of rules by hand.

Beyond LLVM, our approach applies to other compiler infrastructures. It can reduce the effort of backend development and is an alternative to handwritten code generators.

References

[1] ARM, *Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile*, 2022.

[2] Intel, *Intel Advanced Performance Extensions – Architecture Specification*, 2025.

[3] Free Software Foundation, *GCC, the GNU Compiler Collection*, https://gcc.gnu.org/, Free Software Foundation, 2024.

[4] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88.

[5] Bytecode Alliance. (2025) Cranelift. https://cranelift.dev/. [Accessed: 2025-05-21].

[6] T. Kotzmann, C. Wimmer, H. Mössenb öck, T. Rodriguez, K. B. Russell, and D. Cox, "Design of the java hotspot™ client compiler for java 6," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, pp. 7:1–7:32, 2008.

[7] T. Würthinger, C. Wimmer, A. Wö ß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, A. L. Hosking, P. T. Eugster, and R. Hirschfeld, Eds. ACM, 2013, pp. 187–204.

[8] A. Pardoe. (2013) Ryujit: The next-generation jit compiler for .net. https://devblogs.microsoft.com/dotnet/ryujit-the-next-generation-jit-compiler-for-net/. [Accessed: 2025-05-21].

[9] Google. (2025) V8. https://v8.dev. [Accessed: 2025-05-21].

[10] M. Foundation. (2025) SpiderMonkey. https://firefox-source-docs.mozilla.org/js/index.html. [Accessed: 2025-05-21].

[11] Apple. (2025) Webkit. https://webkit.org/. [Accessed: 2025-05-21].

[12] RISC-V International. (2020) Unlocking javascript: V8-RISCV open sourced. https://riscv.org/blog/2020/08/unlocking-javascript-v8-riscv-open-sourced/. [Accessed: 2025-05-25].

[13] O. Community. (2023) Revolutionary optimization: Firefox of openeuler RISC-V achieves 40x performance boost. https://www.openeuler.org/en/blog/20230113-RISC/RISC.html. [Accessed: 2025-05-21].

[14] ARM Ltd., *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, https://developer.arm.com/documentation/ddi0487/latest, ARM Ltd., 2022.

[15] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, https://riscv.org/technical/specifications/, RISC-V Foundation, 2019.

[16] A. Reid. (2022) Machine readable specifications at scale. https://alastairreid.github.io/mrs-at-scale/. [Accessed: 2025-05-21].

[17] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 404–415.

[18] S. Buchwald, A. Fried, and S. Hack, "Synthesizing an instruction selection rule library from semantic specifications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, J. Knoop, M. Schordan, T. Johnson, and M. F. P. O'Boyle, Eds. ACM, 2018, pp. 300–313.

[19] R. Daly, C. Donovick, J. Melchert, R. Setaluri, N. Tsiskaridze, P. Raina, C. W. Barrett, and P. Hanrahan, "Synthesizing instruction selection rewrite rules from RTL using SMT," in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 139–150.

[20] R. Daly, C. Donovick, C. Terrill, J. Melchert, P. Raina, C. W. Barrett, and P. Hanrahan, "Efficiently synthesizing lowest cost rewrite rules for instruction selection," in *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, N. Narodytska and P. Rümmer, Eds. IEEE, 2024, pp. 8–17.

[21] J. Dias and N. Ramsey, "Automatically generating instruction selectors using declarative machine descriptions," in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, M. V. Hermenegildo and J. Palsberg, Eds. ACM, 2010, pp. 403–416.

[22] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[23] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," http://www.smt-lib.org, 2016.

[24] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 62–73.

[25] S. Himmelbauer, C. Hochrainer, B. Huber, N. Mischkulnig, P. Paulweber, T. Schwarzinger, and A. Krall, "The vienna architecture description language," *CoRR*, vol. abs/2402.09087, 2024.

[26] A. Graf, "Compiler backend generation using the vadl processor description language," 2021.

[27] M. Mukherjee and J. Regehr, "Hydra: Generalizing peephole optimizations with program synthesis," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, pp. 725–753, 2024.

[28] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 394–403.

[29] S. Buchwald, "Optgen: A generator for local optimizations," in *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, B. Franke, Ed., vol. 9031. Springer, 2015, pp. 171–189.

[30] R. Joshi, G. Nelson, and K. H. Randall, "Denali: A goal-directed superoptimizer," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 304–314.

[31] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *CoRR*, vol. abs/1711.04422, 2017.

[32] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: modelling, simulation, testing, and data-mining for weak memory," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, p. 40.

[33] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the armv8 architecture, operationally: concurrency and ISA," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 608–621.

[34] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the armv7 instruction set architecture," in *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, ser. Lecture Notes in Computer Science, M. Kaufmann and L. C. Paulson, Eds., vol. 6172. Springer, 2010, pp. 243–258.

[35] G. Morrisett, G. Tan, J. Tassarotti, J. B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger SFI for the x86," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 395–404.

[36] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 379–391.

[37] X. Leroy, "A formally verified compiler back-end," *CoRR*, vol. abs/0902.2137, 2009.

[38] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 71:1–71:31, 2019.

[39] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell, "Isla: integrating full-scale ISA semantics and axiomatic concurrency models (extended version)," *Formal Methods Syst. Des.*, vol. 63, no. 1, pp. 110–133, 2024.

[40] M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell, "Islaris: verification of machine code against authoritative ISA semantics," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 2022, pp. 825–840.

[41] P. Mundkur, J. French, B. Campbell, R. NortonWright, A. Armstrong, T. Bauereiss, S. Flur, C. Pulte, and P. Sewell. (2025) Sail RSIC-V isa model. https://github.com/riscv/sail-riscv.

[42] C. W. Barrett, D. L. Dill, and J. R. Levitt, “A decision procedure for bit-vector arithmetic,” in *Proceedings of the 35th Conference on Design Automation, Moscone center, San Francico, California, USA, June 15-19, 1998*, B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds. ACM Press, 1998, pp. 522–527.

[43] T. Kersten, V. Leis, and T. Neumann, “Tidy tuples and flying start: fast compilation and fast execution of relational queries in umbra,” *VLDB J.*, vol. 30, no. 5, pp. 883–905, 2021.

[44] H. Funke, J. Mühlig, and J. Teubner, “Efficient generation of machine code for query compilers,” in *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, D. Porobic and T. Neumann, Eds. ACM, 2020, pp. 6:1–6:7.

[45] X. Leroy, “Formal certification of a compiler back-end or: programming a compiler with a proof assistant,” in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. G. Morrisett and S. L. P. Jones, Eds. ACM, 2006, pp. 42–54. [Online]. Available: https://doi.org/10.1145/1111037.1111042

[46] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown, “Lightweight, modular verification for webassembly-to-native instruction selection,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds. ACM, 2024, pp. 231–248. [Online]. Available: https://doi.org/10.1145/3617232.3624862

[47] LLVM Project. (2024) LLVM test suite – CTMark benchmarks. https://github.com/llvm/llvm-test-suite/tree/main/CTMark. Accessed: 2025-05-22.

[48] Standard Performance Evaluation Corporation, “SPEC CPU 2017 benchmark suite,” https://www.spec.org/cpu2017, 2017, [Accessed: 2025-05-20].