Synthesizing an Instruction Selection Rule Library from Semantic Specifications

Sebastian Buchwald
Karlsruhe Institute of Technology
Germany
buchwald@kit.edu

Andreas Fried
Karlsruhe Institute of Technology
Germany
fried@kit.edu

Sebastian Hack
Saarland University
Germany
hack@cs.uni-saarland.de

Abstract

Instruction selection is the part of a compiler that transforms intermediate representation (IR) code into machine code. Instruction selectors build on a library of hundreds if not thousands of rules. Creating and maintaining these rules is a tedious and error-prone manual process.

In this paper, we present a fully automatic approach to create provably correct rule libraries from formal specifications of the instruction set architecture and the compiler IR. We use a hybrid approach that combines enumerative techniques with template-based counterexample-guided inductive synthesis (CEGIS). Thereby, we overcome several shortcomings of existing approaches, which were not able to handle complex instructions in a reasonable amount of time. In particular, we efficiently model memory operations.

Our tool synthesized a large part of the integer arithmetic rules for the x86 architecture within a few days where existing techniques could not deliver a substantial rule library within weeks. Using the rule library, we generate a prototype instruction selector that produces code on par with a manually-tuned instruction selector. Furthermore, using 63 012 test cases generated from the rule library, we identified 29 498 rules that both Clang and GCC miss.

CCS Concepts • **Software and its engineering → Retargetable compilers**;

Keywords Program Synthesis, Instruction Selection

ACM Reference Format:
Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an Instruction Selection Rule Library from Semantic Specifications. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3168821

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168821

1 Introduction

Modern instruction set architectures (ISAs), even those of RISC processors, are complex and comprise several hundred instructions. In recent years, ISAs have been more frequently extended to accelerate computations of various domains (e.g., signal processing, graphics, string processing, etc.).

Instruction selectors typically use a library of rules to transform the program: Each rule associates a pattern of IR operations to a semantically equivalent, small program of machine instructions. First, the selector *matches* the pattern of each rule in the library to the IR of the program to be compiled. Then, the selector computes a pattern cover of the program and rewrites it according to the rules associated with the patterns.

The rule library contains at least one rule per machine instruction. Some instructions even have multiple (minimal) IR patterns with the same semantics. For example, the patterns for the x86 instruction andn include:

$$\sim x \,\&\, y \quad x \oplus (x \mid y) \quad y \oplus (x \,\&\, y) \quad y - (x \,\&\, y).$$

Since any of these patterns might occur in a program, the instruction selector needs all rules to ensure a match. Usually, the number of rules exceeds the number of ISA instructions by far. Consequently, the rule libraries of modern compilers have considerable size and consist of hundreds if not thousands of rules. Because of the sheer size of the rule library, manually specifying these rules is tedious and error-prone. To remedy this problem, this paper presents a *fully automatic* approach to synthesize *provably correct* instruction selection rules from formal specifications of the compiler IR's and the ISA's semantics.

We identified two existing approaches in the literature that, in principle, allow us to find *all* minimal IR patterns for a given instruction. The first approach enumerates all IR patterns and tests whether they are semantically equivalent to our instruction. This technique is often used in superoptimizers [2, 5, 15, 22]. The second approach [12] uses template-based counterexample-guided inductive synthesis (CEGIS). This technique is used to synthesize a program that is correct with respect to a given specification. For now, it is sufficient to understand that CEGIS constructs and refines candidate programs from a given multiset of template instructions in a counterexample-guided feedback cycle using a synthesis and verification step. Note that every instruction *occurrence*

counts: If the template multiset contains three add instructions, the synthesized program cannot contain more than three add instructions. We explain CEGIS in more detail in Section 2.4.

Unfortunately, both existing approaches reach their limits when it comes to synthesizing instruction selectors. On one hand, even if we could enumerate and check one candidate per CPU cycle, the enumerative approach would need several hours to handle an instruction that needs 7 out of 21 available IR operations, for example. On the other hand, the CEGIS approach is designed to be given exactly the required operations (in the required multiplicity). Let us assume that we want to synthesize a simple x86 `lea` instruction that adds two values and a constant. Since it is not clear *in advance* how many instances of each IR operation will occur in a pattern, we have to add multiple instances of each operation. This leads to a tremendous slowdown: With 2×21 IR instances, the CEGIS approach would need several hours to find a pattern for the `lea` instruction.

We solve this problem by a technique we call *iterative CEGIS* that combines both approaches. It iteratively enumerates template libraries of increasing size and then synthesizes IR patterns using the CEGIS approach. In summary, we make the following contributions:

- We improve the synthesis algorithm of Gulwani et al. [12] to iteratively explore template libraries of increasing size. Our evaluation shows that this makes the synthesis of instruction selection rules feasible in the first place.
- We present a novel encoding of memory operations that avoids array theory, which we experienced as a major performance bottleneck in the synthesis step. This allows for an extension of Gulwani et al. [12] to memory operations, which are essential when synthesizing instruction selection rules.
- Our experimental evaluation shows that our technique is able to synthesize a large part of the rules for an x86 integer arithmetic instruction selector, including the famous addressing modes. Our approach synthesizes a simple rule library that already covers all primitive x86 integer operations in a few minutes. Using existing synthesis techniques, even the simple library could not be synthesized within a reasonable time budget (days, even weeks). We obtain a more comprehensive library with large, intricate patterns in four days using a standard off-the-shelf satisfiability modulo theories (SMT) solver on a standard desktop workstation.
- We generate a prototype instruction selector from the synthesized rule library. Using the SPEC CINT2000 benchmark suite, we show that our prototype produces code that is close to code produced by a carefully hand-tuned instruction selector.
- We identify 31 612 instruction selection rules missing in GCC and 36 365 missing in Clang using a total of 63 012 generated test cases.

The remainder of this paper is structured as follows: In Section 2, we provide some background information and discuss related work. Section 3 gives an overview of our work, and the following sections provide more detail: Section 4 explains how we model instructions, and Section 5 describes our synthesis algorithm, our prototype instruction selector, and the test case generation. Section 6 discusses limitations of our work and opportunities for future improvement. We evaluate the synthesis algorithm and the quality of the synthesized instruction selection rules in Section 7. Finally, Section 8 concludes.

2 Preliminaries and Related Work

In this section, we provide preliminaries for instruction selection and program synthesis techniques. Along the way, we present related work.

2.1 Instruction Selection

Instruction selection is the task of transforming machine-independent IR operations to machine-dependent instructions. Over the past decades, compilers used a variety of instruction selection approaches that differ significantly in complexity and resulting code quality.

Modern compilers usually represent programs in static single assignment (SSA) form. We concentrate our discussion to approaches in this setting. For other techniques and for a comprehensive survey of instruction selection we refer to Blindell [4].

The instruction selectors built into SSA-based compilers typically use directed acyclic graph (DAG) pattern matching/rewriting on SSA data dependence graphs. Koes and Goldstein [17] have shown that this problem is NP-complete without restricting the ISA appropriately. There exist optimal approaches using mathematical optimization [9] that allow for an extension [8] to patterns containing cycles.

Because CEGIS is (currently) limited to loop-free programs, our approach is limited to DAG patterns. However, modern compilers like LLVM [18, 20] or HotSpot [23] also restrict themselves to DAG or tree patterns and some greedy heuristics for selecting an appropriate covering. In our experimental evaluation, we evaluate the synthesized rule libraries in the research compiler libFirm [19], which also uses a greedy DAG-based instruction selector.

2.2 Generating Instruction Selectors

Dias and Ramsey [7] proposed an algorithm to generate instruction selectors from declarative machine descriptions. They express the input program using a fixed set of small IR patterns called *tiles*. Their synthesis algorithm computes a sequence of machine instructions to implement each tile,

given semantics of the machine instructions (the "machine description") and a set of algebraic rewrite rules. They state that the resulting instruction selector generates code that "can be horribly inefficient" and needs further optimization.

In contrast, our work produces patterns that combine multiple IR operations, and thus make better use of the machine. In addition, whereas Dias and Ramsey rely on a set of rewrite rules, we specify the semantics of IR and machine code using SMT.

2.3 Satisfiability Modulo Theories

In our tool, we use the SMT solver Z3 [6], which follows the wide-spread SMT-LIB standard [3]. In addition, Z3 has preliminary floating-point support, which gives us the opportunity for future support of floating-point instructions.

Even though Z3 supports combining multiple theories in one query, we found that it runs most efficiently with only a single theory. We therefore constrained Z3 to the SMT-LIB theory QF_BV (quantifier-free bit vectors) and modeled all values including indices as bitvectors. Compared to freely combining theories, this reduced the solving time by a factor of two.

2.4 Synthesis

Gold [11] and Shapiro [26] introduced the idea of inductive synthesis. The aim of inductive synthesis is to construct an object (e.g. a program), given a finite set of *test cases*. In the case of program synthesis, these are program arguments along with the expected results.

Solar-Lezama et al. [28] developed this idea into counter-example-guided inductive synthesis (CEGIS). CEGIS iterates inductive syntheses to construct a program that is correct for all possible arguments. It uses two alternating steps to produce new test cases incrementally.

For example, suppose that we want to synthesize a program p that satisfies ϕ for all inputs y. However, the formula $\exists p.\forall y.\phi(p, y)$ contains a universal quantifier, with which SMT solvers have performance problems. CEGIS eliminates this universal quantifier.

CEGIS uses two calls to the SMT solver. The first of these is the *synthesis query*. It constructs a candidate p^* that is valid for all test cases $y_i \in Y$. The second SMT call is the *verification query*. It checks whether p^* is valid for all possible y by querying for a counterexample y^*. If no counterexample is found, p^* is correct for all y. Otherwise, y^* is a useful new test case to refine the next synthesis query.

Gulwani et al. [12] used this technique to build a superoptimizer. A superoptimizer is a tool to find the shortest possible program that implements a given functionality [22]. They developed a representation that can encode loop-free programs as a set of integer variables of limited range, so that a standard SMT solver can enumerate the programs.

They benchmark their tool using examples from the micro-optimization book "Hacker's Delight" [32]. The tool is able to synthesize programs of 16 instructions, albeit with supervision in picking the types of instructions to use.

In our work, we expand the model presented by Gulwani et al. to support more types of instructions, and to be able to synthesize programs unsupervised. See Section 4 and Section 5 for a detailed discussion.

We also investigated modeling instruction selection synthesis as a syntax-guided synthesis (SyGuS) problem [1] and submitted a set of benchmarks to the SyGuS-COMP 2017 [31]. To solve the benchmarks, the participating solvers had to synthesize patterns with at least 6 IR operations. However, the benchmarks already provided the minimal set of necessary IR operations. Unfortunately, none of the participating solvers was able to solve any of these benchmarks.

Another approach in synthesis is to forgo completeness and heuristically prune the search space. Thus, the synthesizer can find significantly larger implementations. This approach has been used in superoptimizers [15, 25] as well as for program synthesis [13, 29, 30]. Unfortunately, this approach is not applicable to instruction selection synthesis, where we need to synthesize all minimal IR patterns for each machine instruction. If a pattern is missing, the instruction selector cannot generate the corresponding instruction if that pattern occurs. If a pattern is not minimal, it is very unlikely to occur, because the compiler will have already optimized the IR.

2.5 Formal Instruction Semantics

Godefroid and Taly [10] synthesize bit-vector formulas for processor instructions from input/output pairs. They examine an instruction's behavior by actually executing it on random test inputs. Then, they synthesize a semantics for the instruction based on a set of templates. Their synthesis algorithm is similar to CEGIS, except that they also search for counterexamples by running more experiments on the actual instruction.

Heule et al. [13] present an approach that synthesizes the formal semantics of complex instructions from a small set of basic instructions. Their algorithm starts with a set of test inputs and results for the machine instruction. It then uses a CEGIS-like loop, using STOKE [25] as its synthesizer and an SMT solver as its verifier.

These approaches are complementary to ours. In principle one could use them to automatically obtain a specification for the machine semantics. However, they cannot be used for synthesizing pattern libraries because they are not complete: Their goal is to obtain a specification and not to enumerate *all* possible corresponding IR patterns.

3 Overview

In this section, we give an overview of our work, before describing its components in more detail in the following sections. Our tool consists of three main components: The

synthesizer, the code generator, and the test case generator. Algorithm 1 gives an overview of the process.

The synthesizer takes semantic models (see Section 4) of both IR operations (parameter I) and machine instructions (parameter M) as its input. Then, the solver runs our iterative CEGIS algorithm (see Section 5) with each instruction from M as the goal g. Each of these runs produces all minimal patterns that implement g using nodes from I (see Section 5.3 for more detail). The synthesizer pairs these patterns with g, and stores all pairs in a pattern database.

The pattern database can aggregate patterns found by different synthesizer runs (see Section 5.5). Either we can run the synthesizer in parallel on multiple machines, or we can first synthesize patterns for a basic set of instructions and expand on these as needed.

The code generator reads the pattern database and produces code for a compiler's instruction selection phase. The code generator is free to use any instruction selection algorithm that works with DAG patterns.

The test case generator reads the pattern database and produces a test case for each pattern. These test cases can be used to identify missing patterns in state-of-the-art compilers (see Section 7.4).

Algorithm 1 Overview

```
1:  procedure Synthesizer(I : {Instruction}, M : {Instruction})
2:      S ← {}                                   ▷ S : {(M × Pattern(I))}
3:      for each g ∈ M do
4:          {p_1, ..., p_n} ← IterativeCEGIS(I, g)   ▷ p_i : Pattern(I)
5:          S ← S ∪ {(g, p_1), ..., (g, p_n)}
6:      end for
7:      Save S to pattern database
8:  end procedure

9:  procedure CodeGenerator(S : {(M × Pattern(I))})
10:     Filter S: Remove non-normalized and duplicated patterns
11:     Sort S from more specific to less specific patterns
12:     for each (g, p) ∈ S do
13:         Emit code: If p matches, replace it with g.
14:                    Otherwise try next pattern
15:     end for
16: end procedure

17: procedure TestCaseGenerator(S : {(M × Pattern(I))})
18:     Filter S: Remove duplicated patterns
19:     for each (g, p) ∈ S do
20:         Emit test case for p.
21:     end for
22: end procedure
```

4 Modeling Instructions

Our synthesizer needs semantic models of both IR operations and machine instructions. Following Gulwani et al. [12], we model these as SMT predicates that relate their inputs and outputs. However, we extend Gulwani's model to include multiple sorts, instructions with preconditions, instructions with multiple results, and instructions that access memory.

An instruction takes n *arguments*, and from these computes m *results*. In addition, some instructions have *internal attributes*, whose values are chosen at synthesis time. For example, a conditional branch instruction has the condition code as an internal attribute.

The sorts of the arguments, internal values, and results form the instruction's *interface*. The interface determines the ways in which instructions may be combined. We specify the interface in three functions S_a, S_i, and S_r. These take an instruction and return the list of argument, internal, and result sorts respectively.

We specify the *behavior* of an instruction by the two SMT formulae defined below. Each of them takes an instruction and three lists of SMT expressions v_a, v_i, and v_r. These are the values to be substituted for the instruction's arguments, internal attributes, and results respectively.

- $P(i, v_a, v_i, v_r)$ is i's *precondition*. If it does not hold, the instruction's behavior is undefined.
- $Q(i, v_a, v_i, v_r)$ is i's *postcondition*. If the precondition holds, $Q(i, v_a, v_i, v_r)$ also holds. Its purpose is to define v_r in terms of v_a and v_i.

In order to fulfill the interface of the instruction i, we require the values in v_a to have the sorts in $S_a(i)$, i.e.

$$\forall 0 \leq k < |S_a(i)|.\, v_a[k] : S_a(i)[k],$$

and similarly for $v_i/S_i(i)$, and $v_r/S_r(i)$.

Example 1 (Right-shift instruction)**.** *Consider the specification of a 32 bit wide right-shift instruction with the semantics of C. In this semantics, the result $v_r[0]$ of the shift is undefined if the shift amount is negative or not less than the bit width. The value to be shifted is $v_a[0]$, the shift amount is $v_a[1]$.*

$$\begin{aligned} S_a(\text{shr32}) &= [\text{BitVec}_{32}, \text{BitVec}_{32}] \\ S_i(\text{shr32}) &= [] \\ S_r(\text{shr32}) &= [\text{BitVec}_{32}] \\ P(\text{shr32}, v_a, v_i, v_r) &= 0 \leq v_a[1] < 32 \\ Q(\text{shr32}, v_a, v_i, v_r) &= (v_r[0] = v_a[0] >> v_a[1]) \end{aligned}$$

4.1 Memory Access

Graph-based IRs typically model the state of memory as an SSA value [19, 23]. We call this memory value *M-value* in the remainder of this paper. Each instruction that accesses memory takes an M-value as an additional argument and produces an M-value as an additional result: The load operation has type $M \times \mathit{Pointer} \to M \times \mathit{Value}$, and the store operation has type $M \times \mathit{Pointer} \times \mathit{Value} \to M$. An operation's M-value result is then used as the argument to the next memory operation. Thus, all memory operations are totally ordered in a chain of M-values[1]. This also holds for load operations, in

[1]This requirement can be relaxed if memory accesses are proven not to alias, but we do not consider this case in our model.

order to model write-after-read dependencies to subsequent store operations.

SMT Model. We exploit this structure in our SMT representation: We model M-values as values of an SMT sort that can hold the relevant state of memory for the synthesis. Thus, the patterns we use in our synthesis are the same as those of the IR.

We must ensure that the memory access operations in our patterns are properly chained. Since a load operation has no effect on the contents of memory, it must change the M-value representation in some other way in order to force the synthesizer to include it in the memory chain. Therefore, the M-value holds two pieces of information for each address: the *memory contents* for that address, and an *access flag*. A load operation sets the access flag of the address it loads from, thus changing the M-value. We assume here that every goal instruction loads from each address at most once. If this is not the case, the access flag can be replaced by a counter.

For basic memory access, we use the SMT functions *ld* and *st*, which load or store a single byte respectively. Their sorts are the same as in the IR, with *Pointer* = BitVec_{32} and *Value* = BitVec_8.

The exact definitions of the sort M for M-values, as well as the functions *ld* and *st*, are specialized to the goal instruction of the synthesis. Therefore, we first specify the goal instruction as a template, using these symbols as placeholders for the actual sorts or functions. Then, we compute the specialized M, *ld*, and *st* from this template, and substitute them for the placeholders.

Representation of M-Values. Program verifiers typically model contents memory using the SMT theory of arrays [27] or an Ackermannized variant of it [21], but we found these approaches to be unsuitable for our needs: During CEGIS, we have to prove that our synthesis candidate is valid for all initial states of memory. When trying to prove this, the SMT solver (Z3) consistently ran out of memory.

Therefore, we use a different technique to represent M-values that is specific to our application: Because we only consider one machine instruction g at a time as our goal, we can restrict the SMT representation of M-values to only represent those addresses that g uses. If a candidate pattern accesses any other memory, it cannot be equivalent to g.

We call the pointers used by g the *valid pointers* for the synthesis of g, and collect them in a list of unevaluated SMT expressions $V(g, v_a, v_i, v_r)$. To obtain this list, we syntactically analyze g's postcondition, and extract all pointer arguments to load and store operations from it.

For example, suppose that our goal instruction is a 32 bit store "store32". We construct this by chaining SMT byte store functions, passing M-values from one to the next.

$$
\begin{aligned}
&Q(\text{store32}, v_a, v_i, v_r) = \\
&\quad \text{let } m_0 \leftarrow st(v_a[0], \quad v_a[1], \quad\; v_a[2][7\ldots0]) \text{ in} \\
&\quad \text{let } m_1 \leftarrow st(m_0, \quad v_a[1]+1, \quad v_a[2][15\ldots8]) \text{ in} \\
&\quad \text{let } m_2 \leftarrow st(m_1, \quad v_a[1]+2, \quad v_a[2][23\ldots16]) \text{ in} \\
&\quad \text{let } m_3 \leftarrow st(m_2, \quad v_a[1]+3, \quad v_a[2][31\ldots24]) \text{ in} \\
&\quad v_r[0] = m_3
\end{aligned}
$$

The valid pointers for synthesizing store32 are then

$$V(\text{store32}, v_a, v_i, v_r) = [v_a[1], v_a[1]+1, v_a[1]+2, v_a[1]+3]$$

Using the valid pointers, we can then define a sort $M(g)$ of M-values, and the primitive load and store functions for the synthesis of g. Even though we define *st* in terms of the valid pointers, this is not a circular dependency, since we only analyzed Q on a *syntactic* level that does not require the definition of *st* to be known.

$M(g)$ is a bit vector of size $|V(g, v_a, v_i, v_r)| \cdot (w+1)$, where w is the bit width of a byte in memory. For each valid pointer, the M-value thus has storage for one byte of memory and its access flag.

In the example above, we have $M(g) = \mathsf{BitVec}_{36}$, whereby bits 0 to 7 store the memory contents for $v_a[1]$ (the first valid pointer), bit 8 stores its access flag, and so on.

SMT Store Function. We can now define the SMT store function *st*. For simplicity, we will fix $V = V(g, v_a, v_i, v_r)$, and not pass it explicitly as an argument. *st* takes an M-value m, a pointer p and a value x to store. It evaluates each valid pointer and compares its value with p. If $V[i] = p$, it returns a modified m, setting the memory contents for $V[i]$ to x.

Thus, continuing the example, we implement *st* as follows:

$$
st(m, p, x) = \begin{cases} \text{replace}(m, 0, x) & \text{if } p = \text{eval}(V[0]) \\ \text{replace}(m, 9, x) & \text{if } p = \text{eval}(V[1]) \\ \text{replace}(m, 18, x) & \text{if } p = \text{eval}(V[2]) \\ \text{replace}(m, 27, x) & \text{if } p = \text{eval}(V[3]) \end{cases},
$$

whereby $\text{replace}(m, i, x)$ returns the bit vector m with the bits from i to $i+7$ replaced by the bit vector x.

The load function *ld* traverses V in the same order to find a matching valid pointer $V[i]$, and then extracts and returns the memory contents for $V[i]$. In addition, it returns the M-value with the access flag for $V[i]$ set.

Note that the valid pointers are not evaluated until the call to *st* or *ld*. Since we use CEGIS, this means that we can substitute concrete values for p and the variables used in V.

The definitions of *ld* and *st* require that their pointer argument is equal to a valid pointer. However, if the synthesizer tries a wrong solution, this might not be the case. We force the synthesizer to only use valid pointers by placing an additional constraint on the synthesis (see Section 5.2).

We must also handle aliasing valid pointers. It is possible that g's arguments alias or that a pointer was expressed in two different but arithmetically equivalent ways in g's

specification. Since we compute V by syntactic analysis, the M-values then have multiple entries referring to the same address. However, *ld* and *st* always check the valid pointers against p in a fixed (albeit arbitrary) order. Therefore, only the first of the aliasing valid pointers will be used for loads and stores, and our model remains consistent.

4.2 Control Flow

In machine language, jumps take a target to jump to when their condition is fulfilled, or fall through. On the other hand, IR jumps make their fall-through target explicit as well. They take references to both basic blocks where execution might continue after the branch.

We follow the idea of IR jumps with their explicit control flow. Our representation of a jump operation computes as many boolean results as the original operation has jump targets (usually two). It sets the result for the branch taken to True, and the other results to False. Thus, when two jump operations return the same values given the same arguments in our model, they will perform the same jumps in the IR or machine language.

Different IRs and processor architectures differ in the way in which they represent conditional jumps. They either use a test for a specific condition and a jump if the test succeeded (e.g. MIPS, libFirm, LLVM), or they use a generic comparison instruction and specify the condition in the jump instruction (e.g. x86, ARM). In order to unify these different approaches, we consider a combination of one comparison and one conditional jump as our goal instruction.

5 Instruction Selection Synthesis

In this section, we discuss the core problem of instruction selection synthesis: Given a goal machine instruction g and the multiset of IR operations I, synthesize an IR pattern that implements g.

We assume that g has no internal attributes. To synthesize a goal instruction with internal attributes, we run a separate synthesis for each possible assignment to them. For example, we have to synthesize a pattern for each condition code of a compare-and-jump pair.

5.1 Pattern Representation

Given the current state of SMT solver technology, we cannot directly represent DAG patterns as an SMT datatype. Instead, we use an extended version of the encoding presented by Gulwani et al. [12]. This encoding uses *location variables* L in order to place operations from a given multiset in a linear order and to determine the operations' arguments. These can either be the result of another operation or one of the pattern's/machine instruction's arguments. In addition, further location variables select the source of the pattern's/machine instruction's results. The *well-formed program constraint* ϕ_{wf} ensures that the assignment to the location variables represents a valid program.

We extend this encoding to support instructions with multiple results, and values of different sorts. To support multiple sorts in a program, we restrict each location variable associated with an operation's argument to those sources with the same sort. We also exclude the ill-sorted connections from the *connection constraint*, as they would produce invalid SMT formulae.

To support operations with multiple results, we associate a set of consecutive locations with each operation, so that further instructions can select any of the return values as their arguments. We therefore have to adapt Gulwani et al.'s *consistency constraint* (a part of ϕ_{wf}), which ensures that each location only has one operation assigned to it. If $L(o)$ is the location of operation $o \in I$, and distinct(S) holds if all elements of S are pair-wise distinct, our consistency constraint becomes

$$\psi_{cons} = \text{distinct}(\{0, \ldots, |S_a(g)| - 1\} \cup \bigcup_{o \in I} \{L(o), \ldots, L(o) + |S_r(o)| - 1\}).$$

The predicate "distinct" is included in SMT-LIB [3] and therefore supported by all conforming SMT solvers.

With this pattern encoding, we have achieved two things: First, the SMT solver can select any program of IR operations by assigning values to the location variables. Second, using the location variables we can construct an SMT formula that assigns values to the pattern's results according to the semantics of the IR program and the pattern's arguments. Gulwani et al. call this the connection constraint; we write $Q^+(I, L, v_a, v_i, v_r)$ by analogy with a single operation's Q.

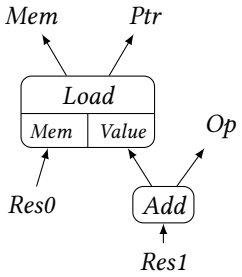
Similarly, we extend other parts of the specification to whole patterns, namely P^+ (by conjunction of all P), V^+ (by union of all V), and S_i^+ (by union of all S_i).

Example 2 (Pattern representation)**.** *We want to synthesize an addition instruction that loads one of its operands from memory. The instruction has three arguments (M-value, pointer, register-operand) and two results (M-value, sum). Given the set of IR operations $I = \{Add, Load\}$, we obtain the set of location variables L. Figure 1c shows a well-formed assignment to L. This assignment places the operations as shown in Figure 1b and fully encodes the pattern shown in Figure 1a.*

Considering the pattern semantics Q^+, we can substitute the location variables with their assignments. This allows us to partially evaluate Q^+ until we receive the formula shown in Figure 1c. This formula contains intermediate variables e_0 to e_6, which hold the argument and result values of the operations.

5.2 Search Algorithm

We now formulate the SMT queries for our synthesis. Again, these are an extension of the scheme presented by Gulwani et al. [12].



(a) IR pattern

Mem	0
Ptr	1
Op	2
Load-Mem	3
Load-Value	4
Add	5

(b) Locations

$$Q^+(I, L, v_a, v_i, v_r) = Q(Load, [e_0, e_1], [], [e_2, e_3])$$
$$\wedge\, Q(Add, [e_4, e_5], [], [e_6]) \wedge e_0 = v_a[0] \wedge e_1 = v_a[1]$$
$$\wedge\, e_4 = e_3 \wedge e_5 = v_a[2] \wedge v_r[0] = e_2 \wedge v_r[1] = e_6$$
$$L: \quad l_{Load} = 3 \quad l_{Load\text{-}Arg0} = 0 \quad l_{Load\text{-}Arg1} = 1$$
$$l_{Add} = 5 \quad l_{Add\text{-}Arg0} = 4 \quad l_{Add\text{-}Arg1} = 2$$
$$l_{Res0} = 3 \qquad l_{Res1} = 5$$

(c) Pattern postcondition Q^+ and location variables L

Figure 1. Well-formed assignment of an IR pattern to locations by location variables L. The provided SMT formula Q^+ depicts the partially evaluated postcondition fixing the location variables as shown in the assignment to L.

Roughly speaking, we are looking for an IR pattern (i.e. an assignment to the location variables) that behaves the same as the goal instruction for every set of arguments. This query is in principle solvable by an SMT solver, because the location variables have finite range. However, it contains universal quantifiers, and thus solving it still takes an impracticable amount of memory and time. Therefore, Gulwani et al. use CEGIS to split the query into a *synthesis* and a *verification* step.

The synthesis query produces an assignment to the location variables and the internal arguments of the IR operations that is valid for a small set of test cases. We define a test case to be the list $t_a :: S_a(g)$ of pattern arguments. The set of all test cases is T. The synthesis formula $\phi_{synth}(I, T, g)$ is then as follows:

$$\exists L : \text{LocationVariables}(I).\ \exists v_i :: S_i^+(I).\ \phi_{wf}(I) \wedge$$
$$\bigwedge_{t_a \in T} \Big(\exists v_r :: S_r(g).\ P^+(I, t_a, v_i, v_r) \Longrightarrow$$
$$P(g, t_a, [], v_r) \wedge Q(g, t_a, [], v_r) \wedge Q^+(I, L, t_a, v_i, v_r) \wedge$$
$$V^+(I, t_a, v_i, v_r) \subseteq V(g, t_a, v_i, v_r)\Big)$$

If ϕ_{synth} is satisfiable, we obtain a model for the location variables L^* and the internal attributes v_i^*. In the next step, we check them with the verification formula:

$$\exists t_a :: S_a(g).\ \exists v_r :: S_r(g).\ \exists v_r' :: S_r(g).$$
$$P^+(I, t_a, v_i^*, v_r) \wedge Q^+(I, L^*, t_a, v_i^*, v_r) \wedge Q(g, v_a, [], v_r') \wedge$$
$$(\neg P(g, t_a, [], v_r) \vee \quad (1)$$
$$v_r \neq v_r' \vee \quad (2)$$
$$V^+(I, t_a, v_i, v_r) \nsubseteq V(g, t_a, v_i, v_r)) \quad (3)$$

The verification formula verifies that the IR pattern represented by L^* and v_i^* is equivalent to g. It does this by searching for a test case t_a, which could (1) meet the pattern's but not g's precondition, (2) cause the pattern and g to produce different results, or (3) lead to an invalid memory access. If the solver finds a t_a^* fulfilling these conditions, the candidate pattern is not equivalent to g, and we discard it. Then, we add t_a^* to the set of test cases T.

If no counterexample exists, L^* and v_i^* represent a pattern that is equivalent to g. We can then reconstruct this pattern from L^* and v_i^* as described by Gulwani et al. [12].

5.3 Finding All Patterns

We have already remarked that an instruction selector needs to know about *all* IR patterns matching a goal instruction. We therefore repeat the CEGIS algorithm, each time requesting a pattern not seen before. We do this by adding clauses that exclude patterns that we have already found. Let F be the set of patterns we have already found, consisting of pairs (L_f, v_f), where L_f : LocationVariables(I) and $v_f :: S_i^+(I)$. We then add the following condition to the synthesis constraint to exclude patterns in F:

$$\bigwedge_{(L_f, v_f) \in F} (L \neq L_f \vee v_i \neq v_f)$$

We refer to this algorithm as CEGISAllPatterns below.

5.4 Iterative CEGIS

The biggest performance issue with synthesis using classical CEGIS is the size of I. It must contain every operation sufficiently often to synthesize any machine instruction, but each single machine instruction will only use a small part of I. In iterative CEGIS, we exploit this discrepancy by replacing one large CEGIS with several smaller ones.

The iterative CEGIS algorithm (Algorithm 2) takes the simple set of IR operations I as input (containing each operation only once). Then, it performs a classical CEGIS for each ℓ-multicombination of I with increasing ℓ. It returns all results with minimal ℓ (i.e. all IR patterns of minimal size).

For a given ℓ we have $\left(\!\binom{|I|}{\ell}\!\right) = \binom{|I|+\ell-1}{\ell}$ iterations of the inner loop. Knuth [16] presents several efficient iteration algorithms over multicombinations.

Refining the Iteration. Many of our goal instructions access memory, which means that they take an argument and produce a result of sort $M(g)$. We can surmise that they will

Algorithm 2 Iteration over all multisets

```
1:  procedure MultisetIteration(I : {Instruction},
        g : Instruction)
2:      ℓ ← 1
3:      R ← ∅
4:      while R = ∅ do
5:          for each ℓ-multicombination I′ of I do
6:              R ← R ∪ CEGISAllPatterns(I′, g)
7:          end for
8:          ℓ ← ℓ + 1
9:      end while
10: end procedure
```

not leave this value untouched, which we can check with this simple SMT query (suppose that m and m' are the indices of the argument and result in question):

$$\neg\exists v_a :: S_a(g), v_i :: S_i(g), v_r :: S_r(g).$$
$$Q(g, v_a, v_i, v_r) \wedge v_a[m] \neq v_r[m'].$$

If this query is satisfiable, $v_r[m']$ must be the result of an IR operation, and we can be sure that g requires a memory operation. By checking whether $v_a[m]$ and $v_r[m']$ differ in memory contents or in an access flag, we can even find out whether g requires a load, store, or both operations.

Assume that our analysis showed that g requires the memory operations $O \subseteq \{\text{load}, \text{store}\}$. We can then prune our iteration space in the following way: Where we would normally iterate over multisets of size ℓ, we instead take O as the fixed first members of I', and only iterate over the remaining $\ell - |O|$ members. This means that we can reduce our work from $\left(\!\binom{|I|}{\ell}\!\right)$ iterations to $\left(\!\binom{|I|}{\ell-|O|}\!\right)$ iterations. For example, if $|I| = 21$, $\ell = 6$, and $|O| = 2$, we require 10 626 instead of 230 230 iterations.

In addition, we sometimes know a priori that no valid pattern can be produced from a certain I'. We implement two criteria that allow us to skip an iteration:

- Assume n operations in I' each have only one value of a certain sort S as their result, but there are $m < n$ consumers of values of S. Then, the pattern must ignore the result of at least one of these operations (say, o), and could have been synthesized from $I' \setminus \{o\}$. Because this has lower cost than I', we must have already tried it unsuccessfully, and we can thus skip synthesis for I'.
- Assume that I' contains an operation that requires an argument of sort S. In this case, we require that there is a *source* of S. A source is either a pattern argument, or an instruction that has a value of S as its result without requiring one as its argument. If we cannot find a source, we skip synthesis for I'.

The second criterion is useful in skipping all patterns with memory access operations if the goal instruction does not access memory.

Search Space Estimate. To compare the search space of classical and iterative CEGIS, we only consider the number of possible arrangements of components in the pattern. Classical CEGIS has to consider all arrangements of $|I|$ elements (more if some operations occur multiple times), giving a search space of $|I|!$. With iterative CEGIS, we have to perform $\left(\!\binom{|I|}{\ell}\!\right)$ CEGIS runs with ℓ operations to find patterns with ℓ operations. Its search space therefore comes to $\sum_{\ell=1}^{\ell_{max}} \left(\!\binom{|I|}{\ell}\!\right) \cdot \ell!$.

If we take $|I| = 21$ and $\ell_{max} = 7$, this yields search spaces of $\approx 2^{65}$ for classical CEGIS, and $\approx 2^{32}$ for iterative CEGIS. We can see that the oversupply of IR operations presents an issue to classical CEGIS.

5.5 Aggregation and Post-Processing

We collect all synthesized patterns in a pattern library. This allows us to aggregate patterns from different (parallel) synthesizer runs. We also perform some filtering on the pattern library to remove duplicated patterns that might stem from commutative arithmetic operations or from similar goal instructions.

5.6 Code Generation

After loading the rule library, the code generator can perform a compiler-dependent filtering step that removes all rules with non-normalized IR patterns. Then, it sorts the rules to match more specific ones first, and generates the instruction selection code.

Of course, the code generator is tightly coupled to the targeted compiler and its instruction selection mechanism. However, our synthesis algorithm is independent from the type of instruction selector used, as long as that instruction selector can work with DAG patterns. In particular, our results are suitable to generate any instruction selector discussed by Blindell [4], except those based on macro expansion.

5.7 Test Case Generation

We also implemented a test case generator that creates a C program for each pattern of the pattern database. This allows us to test which patterns are supported by a certain C compiler, while allowing the compiler to normalize and optimize the given program. Of course, we can also use these test cases to test the code generator of the synthesized instruction selector.

6 Limitations and Future Work

There are several areas where improvement on our work is still possible. In some cases, we are restricted by the available technology in SMT solving:

Division. Division of bit vectors is especially hard for SMT solvers: It is usually specified indirectly in terms of multiplication, which is already a complex operation. The current performance of SMT solvers in the face of division operations is insufficient for our needs, and we chose to exclude division from our set of IR operations.

In practice, one might choose to put a timeout on the verification and accept any pattern where verification times out. We chose not to do this, as it compromises the provable correctness of the generated instruction selection.

Floating-Point Arithmetic. We did not include floating-point arithmetic at all in our work, because there is no efficient way to use it in an SMT query at present. The SMT-LIB project has defined a theory [3, 24], and the SMT solver Z3 has preliminary support for it, but its performance is not yet up to our needs.

Other limitations are due to our program representation and search algorithm:

Different Bit Widths. Currently, we only synthesize instructions for 32 bit wide values. There are three approaches to this problem, none of which is satisfactory:

- We can run separate syntheses for the different bit widths. This approach has tolerable performance, but cannot exploit interactions between operations with different bit widths. For example, the x86 instruction `setcc` only operates on 8-bit-registers but can still be useful for other bit widths.
- We can include IR operations for all bit widths in our synthesis. With our present synthesis algorithm, adding IR operations has exponential performance impact. We would therefore have to restrict ourselves to patterns of approximately size 3.
- We can model instructions in a way that they can stand in for their smaller counterparts if possible. For example, a 32-bit addition can also implement a 16- or 8-bit addition, but a right-shift instruction only works in one bit width. This approach requires us to model unknown bits, because some smaller-width instructions (e.g. Loads) leave the upper bits of their destination in an undefined state. The possibility of unknown bits makes all models more complicated, and again hinders synthesis performance.

Patterns with Loops. Our pattern representation can only handle straight-line programs. The representation can support conditional assignments, but not actual conditional execution or loops. We have this restriction because SMT solvers cannot work with recursive definitions, and therefore also not with unbounded loops.

In program verification, a standard technique is to unroll loops a limited number of times. Using this approach, we could synthesize instructions with fixed iteration length (e.g. SIMD instructions). We could synthesize a pattern with the loop unrolled, and then "roll up" the loop for the purposes of matching. However, our current approach does not scale to the necessary size of pattern.

Unrolling or rolling up loops is only possible for synthesis if the number of iterations is fixed. When this is not the case (e.g. with x86's `rep` prefix), we need more powerful synthesis tools. A fixpoint engine [14] is now part of Z3, although it has not yet been used in program synthesis.

The last item does not affect the synthesis but the further processing of the rule library.

Handling Compile-Time Constants. When handling immediates we currently do not distinguish between compile-time constants and link-time constants. This causes some problems in combination with constant folding. For instance, let us consider the rule that transforms if $(x <_s (x \,\&\, (\sim c)))$... into test x, c; js/jns. If c is a compile-time constant, we will not find the pattern, because the compiler will constant-fold ~c to a new constant c'.

A solution to this problem is to also create a (symbolically) constant-folded pattern for compile-time constants. The main challenge then is to reconstruct the immediate from the folded constants. In our example, we can simply use c = ~c'. However, if the immediate is used multiple times by more complex IR operations, this might become tricky. For the test generator, another challenge is to find particular constants that prevent additional optimizations.

7 Evaluation

In this section, we evaluate the instruction selection synthesis as well as the quality of the resulting prototype instruction selector.

7.1 Setup

For the evaluation, our goal is to generate a prototype instruction selector for the libFirm compiler [19]. Since libFirm provides a well-tuned 32-bit x86 backend, we choose x86 as our target architecture. We provided bit-vector formulas for libFirm's IR operations and our target set of 32-bit x86 integer instructions. We also extended our synthesis tool to generate matcher code for libFirm's greedy instruction selection algorithm. The resulting prototype instruction selector first checks the synthesized patterns and falls back to existing patterns if no synthesized pattern matches. For a given program, the *coverage* is the ratio of IR operations translated by the synthesized instruction selection rules.

We consider two setups for the synthesis: The *basic* setup only contains the register variants of the machine instructions, whereas the *full* setup also contains more complex variants. The two setups aim for different goals. The basic setup aims to minimize the synthesis time while having the same coverage as the full setup. On the other hand, the full

Table 1. Runtime of generated executables for different instruction selections with standard deviation σ. Handwritten refers to the greedy instruction selection implemented in libFirm, whereas Basic and Full refers to the synthesized prototype instruction selections using the corresponding synthesis setups. The coverage column shows the ratio of IR operations translated by the synthesized instruction selector.

Benchmark	Synthesized					Handwritten	$\sigma_{Handwritten}$	$\frac{\text{Basic}}{\text{Handwritten}}$	$\frac{\text{Full}}{\text{Handwritten}}$
	Coverage	Basic	σ_{Basic}	Full	σ_{Full}				
164.gzip	68.39 %	59.37 s	0.48 s	55.78 s	0.18 s	56.41 s	0.28 s	105.25 %	98.87 %
175.vpr	64.72 %	43.97 s	0.12 s	42.06 s	0.36 s	39.55 s	0.15 s	111.16 %	106.34 %
176.gcc	79.26 %	20.56 s	0.03 s	18.67 s	0.02 s	18.38 s	0.02 s	111.84 %	101.60 %
181.mcf	88.35 %	21.52 s	0.11 s	20.12 s	0.10 s	20.12 s	0.12 s	106.95 %	99.97 %
186.crafty	84.32 %	28.68 s	0.03 s	24.16 s	0.03 s	24.79 s	0.03 s	115.67 %	97.43 %
197.parser	73.23 %	57.22 s	0.06 s	52.72 s	0.05 s	52.11 s	0.05 s	109.80 %	101.17 %
253.perlbmk	76.01 %	51.14 s	0.04 s	38.62 s	0.04 s	39.14 s	0.04 s	130.67 %	98.68 %
254.gap	66.04 %	23.84 s	0.13 s	22.53 s	0.12 s	22.14 s	0.13 s	107.70 %	101.78 %
255.vortex	79.37 %	42.75 s	0.19 s	38.97 s	0.14 s	37.42 s	0.11 s	114.26 %	104.15 %
256.bzip2	78.92 %	46.47 s	0.19 s	42.84 s	0.13 s	42.24 s	0.16 s	110.02 %	101.44 %
300.twolf	75.02 %	61.48 s	0.20 s	58.78 s	0.26 s	58.06 s	0.15 s	105.89 %	101.26 %
Geom. Mean	75.46 %							111.56 %	101.13 %

setup aims to maximize the quality of the instruction selection at the cost of an increased synthesis time. Since our modular approach allows to iteratively add new goal instructions to the basic setup, the two chosen setups show the range of all possible setups with the same coverage.

The synthesis as well as the measurements were performed on an Intel Core i7-6700 3.40 GHz with 64 GB RAM. The machine runs a 64-bit Ubuntu 16.04 distribution that uses the 4.4.0-92-generic version of the Linux kernel. We use the SMT solver Z3 in version 4.5.0.

7.2 Synthesis

In this section, we investigate synthesis time. Since our approach allows for modular synthesis, we first synthesize the basic setup and iteratively extend the resulting instruction selection by synthesizing all variants of several instruction groups.

Table 2 shows the chosen instruction groups and the corresponding synthesis time. The synthesis time strongly depends on the maximum pattern size and ranges from a few seconds to several hours for a single goal instruction. For the basic setup, we need 3 min 25 s to synthesize the patterns and 5 s to generate the instruction selection code. Based on this setup, we can improve the quality of the instruction selection by incrementally adding more complex goal instructions. Eventually, this leads to the full setup, which needs 100 h 50 min 54 s to synthesize the patterns and 1 h 06 min 08 s to generate the instruction selection code.

A simple experiment puts these results into contrast with the original CEGIS algorithm [12]. We then tried to synthesize an x86 addition instruction with a memory operand.

Table 2. Synthesis time for different groups of goal instructions. For each instruction group, we also depict the number of goal instructions, the number of synthesized patterns, and the maximum pattern size. The instruction groups contain multiple variants of the following x86 instructions: Load/Store: `mov`; Unary: `neg`, `not`, `inc`, `dec`; Binary: `add`, `and`, `lea`, `or`, `rol`, `ror`, `sar`, `shl`, `shr`, `sub`, `xor`; Flags: `cmp`, `jcc`, `jmp`, `test`. The basic setup only contains the basic variants of `mov`, `neg`, `not`, `and`, `lea`, `or`, `sar`, `shl`, `shr`, `sub`, `xor`, `cmp`, `jcc`, `jmp`.

Group	#Goals	Patterns		Synthesis Time
		#	Size	
Basic	39	575	4	3 min 25 s
Load/Store	35	607	4	5 min 45 s
Unary	70	2106	7	18 h 10 min 58 s
Binary	260	6316	6	10 h 27 min 06 s
Flags	265	145441	7	72 h 07 min 05 s
Total	630	154470	7	100 h 50 min 54 s

This instruction uses 3 IR operations (Load, Add, Store) and takes 5 seconds to synthesize with our iterative approach. Running the original CEGIS algorithm on the same machine, the synthesis for this instruction did not finish within 64 hours.

7.3 Generated Instruction Selection

In this section, we evaluate the quality of our generated prototype instruction selector. For this purpose, we compile the C programs of the SPEC CINT2000 benchmark suite with

our generated instruction selector and the existing instruction selector of libFirm. When compiling with the generated instruction selector, we measure its coverage, i.e. the ratio of IR operations that it can translate. Furthermore, we compare the time taken by the instruction selection phase and the runtime of the generated executables.

When comparing the time taken by the instruction selector, we observe that the basic setup takes about 1.66× as long as the existing instruction selector, with the whole compiler backend taking 14 % longer. However, the full setup takes between 1217× and 1804× as long as the existing instruction selector, and the whole backend takes between 57× and 132× as long. This is because our pattern library still has 60 000 rules after post-processing, which the prototype instruction selector tries one by one. Note that this is only a deficiency of the prototype instruction selector and orthogonal to the synthesis of the pattern library. With more advanced pattern matching algorithms, which can exploit the common sub-structure among patterns, we expect a significant reduction in compile time.

Table 1 shows the results of compiling and running the benchmarks with both basic and full synthesized instruction selection. The coverage column shows that our instruction selector can transform 75.46 % of all IR operations on average. The remaining operations include function calls, operations involving other bit widths than 32 bit, floating-point operations, and variadic ϕ-functions.

In addition, we compare the runtime of the generated executables when compiled with the basic and full synthesized instruction selector, and the pre-existing handwritten instruction selector. The depicted times show the average of 20 executions. Compared to the handwritten instruction selector, the full and basic setup increases the average runtime by 1.13 % and 11.56 %, respectively.

The performance benefit of the handwritten selector comes from several hand-coded "tricks" that our automatically generated selector does not (yet) support. First, the handwritten instruction selector supports overlapping patterns in some cases. In particular, it can repeat the computation of the same effective address in multiple machine instructions to reduce register pressure. Our generated prototype instruction selector strictly avoids overlapping patterns and would hold the effective address in a register. Again, note that this is not an issue of the synthesized pattern library but of the instruction selector itself. Second, the handwritten instruction selector tries to use existing arithmetic operations to produce desired x86 flags. For instance, it can combine subtraction and comparison operations with the same operands into a single sub instruction. Such a pattern has multiple roots and our prototype instruction selector currently cannot match it.

7.4 Testing State-of-the-Art Compilers

We used our generated test programs (cf. Section 5.7) to identify unsupported patterns in GCC 7.2 and Clang 5.0. With a total of 63 012 test programs, we found 31 612 unsupported patterns in GCC and 36 365 unsupported patterns in Clang. We give some examples of the 29 498 patterns that are not supported by both compilers:

- While both compilers support x & (x − 1) →blsr x, they do not support x + (x | −x) →blsr x.
- Full addressing mode for the lea instruction is not supported: &bytes[x + 4 ∗ y + 42] →lea bytes+42(x,y,4). Another lea pattern that is missing in both compilers is &bytes[42 + (x << 2) − x] →lea bytes+42(x,x,2), which uses the same value as base and index register.
- The majority of missing patterns involves cmp or test instructions, combined with conditional jumps. For instance, int z = x − y; if (z < ~z)... can be translated to cmp x, y; js/jns ..., but both compilers fail to recognize that if(z < ~z) tests the sign bit.

We provide the tools to create a full HTML report of missing patterns as part of our artifact evaluation (see Section A.4). In addition, we provide a website accompanying this paper at http://libfirm.org/selgen.

8 Conclusion

In this paper, we presented a fully automatic approach to create provably correct rule libraries for instruction selection. Our approach is based on template-based counterexample-guided inductive synthesis (CEGIS), a technique to automatically synthesize programs that are correct with respect to a formal specification. We overcome several shortcomings of an existing SMT-based CEGIS approach, which was not applicable to our setting in the first place. We propose a novel way of handling memory operations and show how the search space can be iteratively explored to synthesize rules that are relevant for instruction selection.

Our approach automatically synthesized a large part of the integer arithmetic rules for the x86 architecture within a few days where existing techniques could not deliver a substantial rule library within weeks. Using the rule library, we generate a prototype instruction selector that produces code on par with a manually-tuned instruction selector. Furthermore, using 63 012 test cases generated from the rule library, we identified 29 498 rules that both Clang and GCC miss. This shows that even state-of-the-art compilers have room for improvements in their instruction selections.

A Artifact Description

A.1 Abstract

We provide a Docker image to demonstrate the different stages of our workflow. You can synthesize an instruction selection rule library (Section 7.2), test the state-of-the-art compilers (Section 7.4), build a C compiler using that library, and run it on the SPEC CINT2000 benchmarks (Section 7.3, benchmarks must be provided separately for licensing reasons). The stages can be executed in sequence or separately.

A.2 Description

A.2.1 Check-List (Artifact Meta Information)

- **Algorithm:** Synthesis of instruction selection patterns.
- **Program:** SPEC CINT2000 (proprietary, not included).
- **Compilation:** GCC 7.2.0 and Clang 5.0.0-3 included.
- **Binary:** Pre-built compiler with synthesized instruction selection pass included.
- **Run-time environment:** Any Linux, Ubuntu 17.10 included.
- **Hardware:** x86-64, multicore recommended.
- **Execution:** Disable power-saving for accurate benchmark results.
- **Output:** Instruction selection pattern library, compiler with synthesized instruction selection pass, SPEC CINT2000 benchmark results, comparison table of supported patterns between compilers.
- **Experiment workflow:** Shell scripts.
- **Experiment customization:** Yes.
- **Publicly available?:** Yes.

A.2.2 How Delivered

The AE archive can be downloaded from https://doi.org/10.5281/zenodo.1095055.

Most required software is bundled in a Docker container. Due to license restrictions, the user has to provide an ISO image of the SPEC CPU2000 benchmarks.

A.2.3 Hardware Dependencies

The software runs on any x86-64 processor. We recommend a recent multi-core model due to the long running time of our experiments. On an eight-core Intel i7-6700, the full experiment runs for approx. five days.

A.2.4 Software Dependencies

We require a Linux host system with Docker installed. In order to mount the SPEC ISO image, the host system must support loop devices, and our container must run in privileged mode.

A.2.5 Datasets

An ISO image of the SPEC CPU2000 benchmarks (named `cpu2000-1.3.1.iso`) is required to run the compiler benchmarking experiments. The other experiments do not need external datasets.

A.3 Installation

Docker may have problems accessing network file systems. We recommend saving the SPEC ISO image and the AE archive on a local hard drive.

1. (optional) Place the SPEC CPU2000 ISO image in a directory readable by the Docker daemon. We refer to this directory as `ISODIR`.
2. Extract the AE archive into a directory of your choice.
3. (with SPEC ISO image) Start the Docker container with `./start.sh ISODIR`. Pass the name of the directory where the SPEC ISO image resides as the argument, not the image file itself.
3. (without SPEC ISO image) Start the Docker container with `./start.sh`.

A.4 Experiment Workflow

Each experiment is provided as a script in the directory `/app/experiments`. You should be in this directory when the container has started.

To replicate our full workflow, run these experiments in order:

full-synthesis.sh: This experiment synthesizes instruction selection patterns for all instructions that are also supported by the manually written instruction selector.

On a recent eight-core machine, this synthesis takes four days. If you choose to skip the synthesis, you can run the remaining experiments with the pre-built rule library.

build-compiler.sh: This experiment takes the rule library synthesized by `full-synthesis.sh`, generates an instruction selector from it, and builds a compiler with this instruction selector.

The compilation of the compiler takes approx. five hours. If you choose to skip the synthesis, you can run the remaining experiments with the pre-built compiler.

With the rule library and compiler you can now reproduce our evaluation using the following scripts.

run-tests.sh: This experiment checks whether the different compilers support each pattern from the synthesized pattern library. It generates a short C program from each pattern and compiles it to assembly using GCC, Clang, and libFirm with the handwritten and synthesized instruction selector. Then, it counts how many assembler instructions each of the compilers required.

This script runs for approx. three hours.

spec.sh: This experiments runs the SPEC CPU2000 benchmarks using libFirm with the handwritten and synthesized instruction selector.

Both SPEC runs together take approx. ten hours.

We have also included two experiments that demonstrate our full workflow but require less time.

basic.sh: This experiment runs the full workflow with a minimal set of patterns. It synthesizes a rule library in approx. 5 minutes, compiles libFirm with an instruction selector synthesized from this library, and finally compares it against the handwritten instruction selector in the SPEC CINT2000 benchmarks.

Since the SPEC runs dominate this experiment, it also takes approx. 10 hours.

bmi.sh: This experiment shows how to extend libFirm's handwritten instruction selector with a synthesizsed instruction selector that supports new instructions. It synthesizes a rule library for the bit manipulation instructions `andn`, `blsi`, `blsmsk`, `blsr`, `btc`, `btr`, and `bts`. Then, it compiles libFirm with an instruction selector synthesized from this library.

Finally, it generates tests (see `run-tests.sh`) to check whether GCC and Clang support the patterns for the bit manipulation instructions.

This experiment takes about five minutes.

A.5 Evaluation and Expected Results

All experiments place their results in `/app/results/` within the container.

The result directory can also be accessed from outside the container as the directory `results/` in the extracted AE archive. You need root privileges to write to this directory outside the container, because user IDs within and without Docker do not match up.

The result directory is cleaned every time the container starts. Save any results you would like to keep before starting the container.

full-synthesis.sh: This experiment places the rule library in `rule-library.dat`.

The rule library should have 154 470 entries.

build-compiler.sh: This experiment saves the synthesized instruction selector as `instruction-selector.c` and produces the compiler binary, named `cparser`. The compiler fully supports C99 and is largely GCC-compliant. When given the option `-mautotransform`, the compiler uses the synthesized instruction selector.

run-tests.sh: This experiment produces an HTML table in `test-result.html`. The table contains one row for each pattern where at least one compiler produced more instructions than expected. The entries for each compiler give the number of instructions produced, with non-optimal entries being colored red.

You can expand each table cell by clicking on it, showing the source code of the tests and the assembly generated by each compiler.

The table should show 31 612 unsupported rules for GCC and 36 365 unsupported rules for Clang.

spec.sh: This experiment stores the usual SPEC results in `spec-handwritten/` for the handwritten instruction selector and `spec-autotransform/` for the synthesized instruction selector.

In addition, it produces an HTML table comparing the instruction selectors in `spec-comparison-full.html`. This table contains the average running times of the binaries produced by the handwritten and the synthesized instruction selector for each benchmark. It also contains the standard deviations of the running times for each benchmark. In addition, it provides the ratio between the running times and the geometric mean over all benchmarks.

Compare this table with the "Full/Handwritten" column in Table 1 in the paper.

basic.sh: This experiment produces SPEC results in the same way as `spec.sh`, namely:

- `spec-handwritten/`
- `spec-basic-autotransform/`
- `spec-comparison-basic.html`

Compare the results with the "Basic/Handwritten" column in Table 1 in the paper.

bmi.sh: This experiment produces a table of test cases in the same way as `run-tests.sh`.

Observe here that libFirm with the synthesized instruction selector can handle all patterns, but the other compilers miss some of them. Note also that libFirm with the handwritten instruction selector does not support any of the instructions. They are fully implemented by the synthesized instruction selector.

A.6 Experiment Customization

Due to the nature of Docker, all customizations will be lost after you exit the container.

You can customize the set of goal instructions for which `full-synthesis.sh` generates patterns by editing the script `/app/selgen/run_groups.sh`.

This script calls `run.sh` with different groups of instructions to synthesize. To speed up the synthesis, you can skip one or more groups by deleting or commenting out the calls to `run.sh`. Refer to Table 2 in the paper to get an estimate of how much time each group will take. For example, to skip the synthesis of the `cmp` and `test` instructions, delete lines 30–33 from `run_groups.sh`.

Alternatively, you can also reduce the number of addressing modes. They are listed after `--srcam` and `--destam` for source and destination addressing modes respectively. An instruction's synthesis takes longer the more components (base, index, scale, displacement) its addressing mode has. Furthermore, an instruction using a destination addressing mode needs one more IR operation than the corresponding instruction using the same source addressing mode.

Acknowledgments

We thank Christoph Mallon, Manuel Mohr, Maximilian Wagner, Andreas Zwinkau, our shepherds Tipp Moseley and Santosh Nagarakatte, as well as the anonymous reviewers for their many helpful comments and suggestions.

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). (2016). http://www.smt-lib.org

[4] Gabriel Hjort Blindell. 2016. *Instruction Selection: Principles, Methods, and Applications* (1st ed.). Springer Publishing Company, Incorporated.

[5] Sebastian Buchwald. 2015. *Optgen: A Generator for Local Optimizations*. Springer Berlin Heidelberg, Berlin, Heidelberg, 171–189. https://doi.org/10.1007/978-3-662-46663-6_9

[6] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[7] João Dias and Norman Ramsey. 2010. Automatically Generating Instruction Selectors Using Declarative Machine Descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 403–416. https://doi.org/10.1145/1706299.1706346

[8] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. 2008. Generalized instruction selection using SSA-graphs. In *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*, Krisztián Flautner and John Regehr (Eds.). ACM, 31–40. https://doi.org/10.1145/1375657.1375663

[9] Erik Eckstein, Oliver König, and Bernhard Scholz. 2003. Code Instruction Selection Based on SSA-Graphs. In *Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings (Lecture Notes in Computer Science)*, Andreas Krall (Ed.), Vol. 2826. Springer, 49–65. https://doi.org/10.1007/978-3-540-39920-9_5

[10] Patrice Godefroid and Ankur Taly. 2012. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 441–452. https://doi.org/10.1145/2254064.2254116

[11] E. Mark Gold. 1967. Language identification in the limit. *Information and Control* 10, 5 (1967), 447–474.

[12] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[13] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[14] Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. *μZ– An Efficient Engine for Fixed Points with Constraints*. Springer Berlin Heidelberg, Berlin, Heidelberg, 457–462. https://doi.org/10.1007/978-3-642-22110-1_36

[15] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566

[16] Donald E. Knuth. 2005. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional.

[17] David Ryan Koes and Seth Copen Goldstein. 2008. Near-optimal instruction selection on DAGs. In *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, Mary Lou Soffa and Evelyn Duesterwald (Eds.). ACM, 45–54. https://doi.org/10.1145/1356058.1356065

[18] Chris Lattner and Vikram S. Adve. 2004. 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, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[19] libFirm Website 2017. Firm – Optimization and Machine Code Generation. (2017). http://libfirm.org

[20] LLVM Website 2017. The LLVM Compiler Infrastructure Project. (2017). http://llvm.org

[21] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[22] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 122–126.

[23] Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, Saul Wold (Ed.). USENIX. http://www.usenix.org/publications/library/proceedings/jvm01/paleczny.html

[24] Philipp Rümmer and Thomas Wahl. 2010. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*.

[25] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[26] Ehud Y. Shapiro. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA.

[27] Carsten Sinz, Stephan Falke, and Florian Merz. 2010. A Precise Memory Model for Low-level Bounded Model Checking. In *Proceedings of the 5th International Conference on Systems Software Verification (SSV'10)*. USENIX Association, Berkeley, CA, USA, 1–9.

[28] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[29] Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of Machine Code from Semantics. *SIGPLAN Not.* 50, 6 (June 2015), 596–607. https://doi.org/10.1145/2813885.2737960

[30] Venkatesh Srinivasan, Tushar Sharma, and Thomas Reps. 2016. Speeding Up Machine-code Synthesis. *SIGPLAN Not.* 51, 10 (Oct. 2016), 165–180. https://doi.org/10.1145/3022671.2984006

[31] SyGuS-COMP 2017. SyGuS-COMP 2017. (2017). http://www.sygus.org/SyGuS-COMP2017.html

[32] Henry S. Warren Jr. 2012. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional.