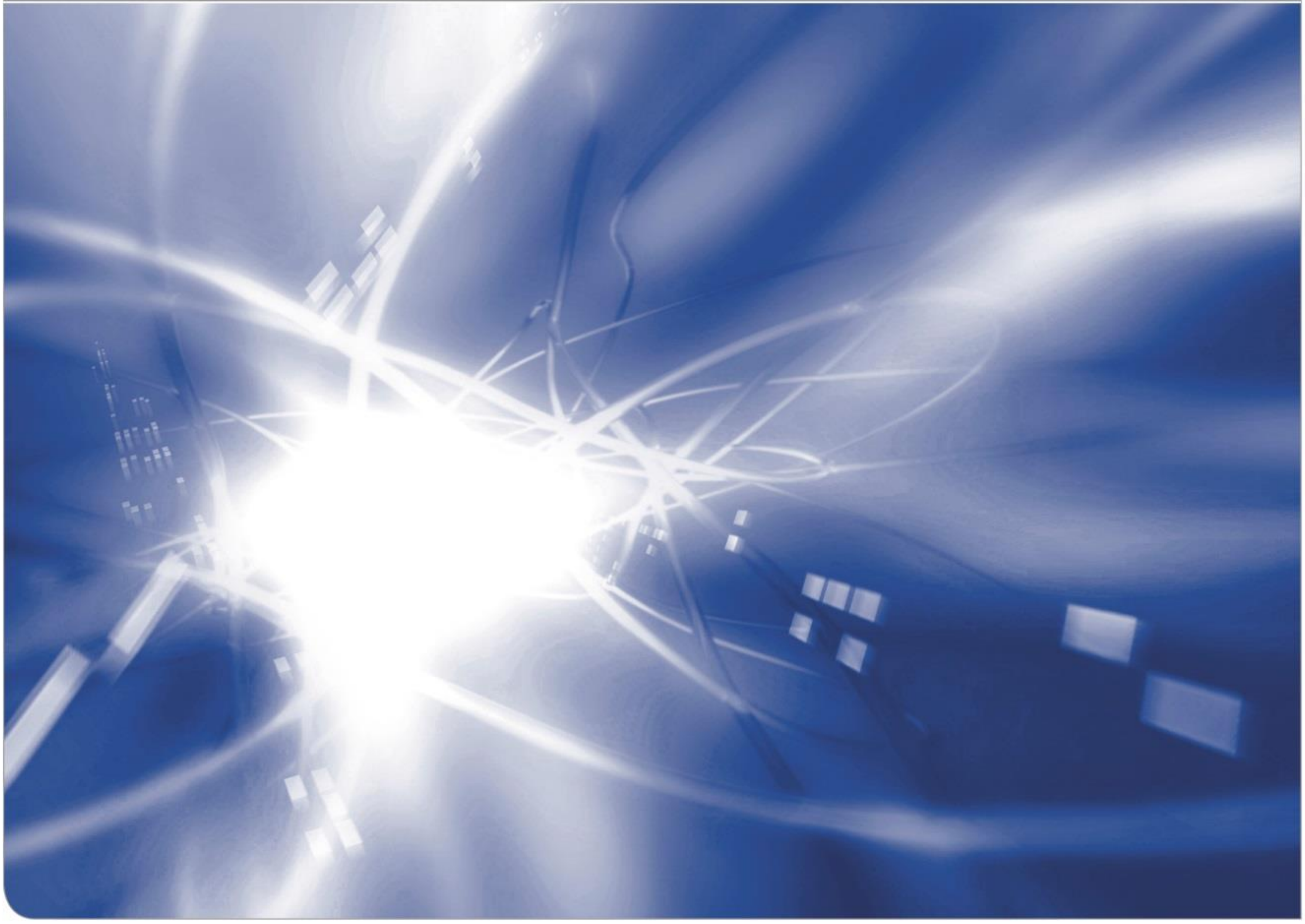
Karlsruhe Institute of Technology

Synthesizing Instruction Selection

by Sebastian Buchwald[1], Andreas Fried[1], Sebastian Hack[2]

KIT SCIENTIFIC WORKING PAPERS 59



[1] Karlsruhe Institute of Technology, Karlsruhe/Germany

[2] Saarland University, Saarbrücken/Germany

Institute for Program Structures and Data Organisation
Programming Paradigms Group
Am Fasanengarten 5
76131 Karlsruhe, Germany
http://pp.ipd.kit.edu/

Imprint

Karlsruhe Institute of Technology (KIT)
www.kit.edu

2017

ISSN: 2194-1629

Abstract

Instruction selection is the part in a compiler that transforms 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 using template-based counter-example guided synthesis (CEGIS). Thereby, 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 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. With respect to the runtime of the compiled programs, we show that the synthesized rules are close to a manually-tuned instruction selector.

Categories and Subject Descriptors D.3.4 [*Processors*]: Retargetable compilers; I.2.2 [*Automatic Programming*]: Program synthesis

Keywords Program Synthesis, Instruction Selection

1. Introduction

Instruction selection is the part of a compiler that translates the platform-independent compiler intermediate representation (IR) into machine code. Modern instruction set architectures (ISAs), even RISC processors, are complex and comprise several hundred instructions. In recent years, ISAs are 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 instructions 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 whose patterns are part of the cover.

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 `inc` include:

$$x + 1 \quad 1 + x \quad x - (-1) \quad -\operatorname{not}(x)$$

Therefore, the number of rules usually 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. To this end, we extend existing techniques [10] for template-based counter-example guided inductive synthesis (CEGIS). The goal of CEGIS is 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 counter-example guided feedback cycle using a synthesis and verification step. Note that every instruction *occurrence* counts: If the template multiset contains three adds, the synthesized program cannot contain more than three adds. We explain CEGIS in more detail in Section 2.4. The algorithm of Gulwani et al. uses an SMT solver for both the verification and the synthesis part and has been shown to be practical for short programs and small template sets (≈ 10 instructions).

To automatically generate instruction selection rules, we use CEGIS to synthesize, for each machine instruction, a set of patterns of IR nodes. Each pattern then *provably* has the same semantics as the machine instruction. However, in this setting, the technique of Gulwani et al. technique is not directly applicable for two reasons:

First, we cannot fix a template library of practical size: Even the integer subset of a standard IR has more than 20 different instructions. Then, it is not clear *in advance* how many instances of each IR instruction will occur in a pattern. Even if we provide, say, three instances per IR instruction, the algorithm would not terminate in a reasonable amount of time. We solve this problem by a technique we call *iterative CEGIS* that iteratively explores template libraries of increasing size.

Second, machine instructions sometimes use memory. For example, x86 possesses powerful addressing modes that allow for loading one operand from memory. However, a straightforward extension of Gulwani et al.'s technique using array theory slows down SMT solving to an extent that renders the approach impractical. We solve this by a novel encoding of memory operations into bit vectors that is specific to our application setting.

In summary, we make the following contributions:

- We improve the synthesis algorithm of Gulwani et al. [10] 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. [10] 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 4.5 days using a standard off-the-shelf SMT solver on a standard desktop workstation.

  Concerning the runtime of the compiled programs, our measurements on the SPEC CINT2000 benchmark suite show that our synthesized rule library is close to a carefully hand-tuned one.

The remainder of the 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 and the automatic generation of code to match the synthesized rules. 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 the 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 [3].

The instruction selectors built into these compilers typically use directed acyclic graph (DAG) pattern matching and rewriting on SSA data dependence graphs. Koes and Goldstein [13] have shown that this problem is NP-complete without restricting the ISA appropriately. There exist optimal approaches using mathematical optimization [7] that allow for an extension [6] to patterns containing cycles. However, modern compilers like LLVM [14] or HotSpot [18] restrict themselves to DAG or tree patterns and some greedy heuristics for selecting an appropriate covering.

Because CEGIS is (currently) limited to loop-free programs, our approach is also limited to DAG patterns. However, as outlined above, DAG patterns are the state of the art in modern compilers. In our experimental evaluation, we evaluate the synthesized rule libraries in a research compiler that uses a greedy DAG-based instruction selector similar to the one used by LLVM.

2.2 Generating Instruction Selectors

Dias and Ramsey proposed an algorithm to generate instruction selectors from declarative machine descriptions [5]. For each instruction, the machine description provides a semantically equivalent IR pattern. They then search for an equivalent machine instruction sequence for each IR operation by applying a set of user-provided algebraic laws. The resulting instruction selector generates code that "can be horribly inefficient" and needs further optimization.

In contrast, our work produces patterns which incorporate 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 Satisfiability Modulo Theories (SMT).

2.3 Satisfiability Modulo Theories

In our tool, we chose to use the SMT solver Z3 [4], because it supports queries with a mixture of integer and bit-vector arithmetic. In addition, Z3 has preliminary floating-point support, which gives us opportunities for future work to support floating-point instructions. Z3 follows the widespread SMT-LIB standard [2].

In the algorithms below, we will denote the integer sort as Int, and the sort of n-bit wide bit vectors as BitVec_n. To represent SMT queries, we use the following notation:

$(r, m) \leftarrow \text{SMTSOLVE}(\exists x.\phi(x))$

The function returns a pair, whereby the first element is either sat (satisfiable) or unsat (unsatisfiable), and the second element gives a model for the existentially quantified variables if the formula is satisfiable. We represent the model as a function from a variable's name to its assigned value.

For example, the query

$(r, m) \leftarrow \text{SMTSOLVE}(\exists x.\, \exists y.\, \exists z.\, x * x + y * y = z * z)$,

may return $r = \mathsf{sat}$ and $m(x) = 3, m(y) = 4, m(z) = 5$.

2.4 Synthesis

Gold [9] and Shapiro [21] 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. developed this idea into Counterexample-Guided Inductive Synthesis (CEGIS) [23]. 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 interactively.

Algorithm 1 shows how to use CEGIS with an SMT solver. In this example, we want to synthesize a program p that satisfies ϕ for all inputs y. However, this formula contains a universal quantifier, with which SMT solvers have performance problems. CEGIS eliminates this universal quantifier.

CEGIS uses two calls to the SMT solver (in lines 4 and 8). The first of these is the *synthesis query*. It constructs a candidate p^* that is valid for all $y_i \in Y$. The second SMT call is the *verification query*. It checks whether p^* is in fact 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.

Algorithm 1 CEGIS

```
1: procedure CEGIS($\exists p.\forall y.\phi(p, y)$)
2:     $Y \leftarrow \emptyset$
3:     loop
4:         $(r_1, m_1) \leftarrow \text{SMTSOLVE}(\exists p^*. \bigwedge_{y_i \in Y} \phi(p^*, y_i))$
5:         if $r_1 = \mathsf{unsat}$ then
6:             return $(\mathsf{unsat}, \emptyset)$
7:         end if
8:         $(r_2, m_2) \leftarrow \text{SMTSOLVE}(\exists y^*.\neg\phi(m_1(p^*), y^*))$
9:         if $r_2 = \mathsf{unsat}$ then
10:            return $(\mathsf{sat}, \{p \mapsto m_1(p^*)\})$
11:        else
12:            $Y \leftarrow Y \cup m_2(y^*)$
13:        end if
14:    end loop
15: end procedure
```

Gulwani et al. used this technique to build a superoptimizer [10]. A superoptimizer is a tool to find the shortest possible program that implements a given functionality. 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" [24]. 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 sections 4 and 5 for a detailed discussion.

Another approach in synthesis is to forgo completeness in exchange for a larger search space. Schkufza et al. [20] built their stochastic superoptimizer and synthesizer STOKE on this principle. STOKE considers individual bits, and tries to match these up between goal and candidate. However, this technique has problems when computations yield only a few bits (e.g. boolean results). Because we need to synthesize comparisons and conditional jumps, we decided to use classical, complete synthesis.

2.5 Formal Instruction Semantics

Godefroid and Taly synthesize bit-vector formulas for processor instructions from input/output pairs [8]. 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. present an approach that synthesizes the formal semantics of complex instructions from a small set of basic instructions [11]. Their algorithm starts with a set of test inputs and results for the machine instruction. It then uses a CEGIS-like loop, using STOKE [20] as its synthesizer and an SMT solver as its verifier.

Both of these techniques could provide us with models for machine instructions. Then, we would only need to specify the IR manually.

3. Overview

In this section, we give an overview of our work, before describing its components in more detail in the following sections. Refer to Table 1 for notation used in this paper.

Our instruction selection generator consists of two main components: The synthesizer and the code generator. Algorithm 2 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

Operations on lists and bit vectors	
$l = [a, b, c]$	literal list/bit vector
$l[i]$	ith element of l (zero-based)
$l[i...j]$	Elements i down to j of l
$l[i \mapsto x]$	l with ith element set to x
$\|l\|$	Length of l
$k \circ l$	Concatenation of k and l
SMT	
SMTSOLVE	Call to SMT solver, see Section 2.3
$e : S$	expression e has sort S
$l :: L$	expressions in list l have sorts in list L (i.e. $\forall n.\, l[n] : L[n]$)
Miscellaneous	
$\{S\}$	sort of sets with elements of sort S
$\{\{S\}\}$	sort of multisets with elements of sort S
$\{a, b, c\}$	set literal
$\{\{a, b, c\}\}$	multiset literal
let $v \leftarrow x$ in t	define local variable v to be x in term t

Table 1: Notational conventions used in this paper

Algorithm 2 Overview

```
1: procedure SYNTHESIZER($I$ : {Instruction},
           $M$ : {Instruction})
2:     $S \leftarrow \{\}$                    $\triangleright S : \{(M \times \text{Pattern}(I))\}$
3:     for each $g \in M$ do
4:         $\{p_1, \ldots, p_n\} \leftarrow$ ITERATIVECEGIS($I, g$)
                                          $\triangleright p_i$ : Pattern($I$)
5:         $S \leftarrow S \cup \{(g, p_1), \ldots, (g, p_n)\}$
6:     end for
7:     Save $S$ to pattern database
8: end procedure

9: procedure CODEGENERATOR($S : \{(M \times \text{Pattern}(I))\}$)
10:    Sort $S$ from more specific to less specific patterns
11:    for each $(g, p) \in S$ do
12:        Emit code: If $p$ matches, replace it with $g$.
13:                   Otherwise try next pattern
14:    end for
15: end procedure
```

from M as the goal g. Each of these runs produces all minimal patterns with nodes from I which implement g (see Section 5.2.1 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. 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.

In the second step, 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. For our prototype implementation, see Section 7.1.

4. Modeling Instructions

Our synthesizer needs semantic models of both IR operations and machine instructions. Following Gulwani et al. [10], we model these as SMT predicates that connect 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 compile 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 three functions 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. In order to fulfill the interface of the instruction i, we require $v_a :: S_a(i)$, $v_i :: S_i(i)$, and $v_r :: S_r(i)$.

The first two functions return SMT formulas that specify the instrucion's pre- and postcondition:

- $P(i, v_a, v_i, v_r)$ is i's *precondition*. If this formula does not hold, the instruction's behavior is undefined.
- $Q(i, v_a, v_i, v_r)$ is i's *postcondition*. If $P(i, v_a, v_i, v_r)$ 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.

The third function, V, returns a list of SMT expressions, namely the list of *valid pointers* for the instruction i. Depending on whether the instruction is the synthesis' goal or a candidate, we either assume or require that the pointers in $V(i, v_a, v_i, v_r)$ are valid.

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

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

4.1 Memory Access

It is typical for compiler IRs to model memory in a linear way, either implicitly or explicitly. In LLVM for example, a "memory value" is implicitly consumed and produced by every memory instruction, by the fact that instructions are linearly ordered inside a basic block. Other IRs [15, 18] make this explicit by modeling memory as a proper IR value. 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. In this model, load and store instructions have the following types:

$$\begin{aligned}
\text{load} &: M \times \mathit{Pointer} \to M \times \mathit{Value}\\
\text{store} &: M \times \mathit{Pointer} \times \mathit{Value} \to M
\end{aligned}$$

Note that load instructions also produce a new M-value, even though a load instruction does not change the contents of memory. We still need this M-value for two reasons: First, load instructions potentially do have side-effects on volatile memory (e.g. memory-mapped device registers). Second, we must not reorder load instructions with respect to subsequent store instructions, because they might have a read-after-write dependency. Thus, all memory access operations in our programs are lined up on a chain of M-values[1].

We must now ensure that the synthesizer also puts load operations into the M-value chain. In order to force the synthesizer to do so, load instructions must change M-values in some way. Otherwise, the synthesizer would be free to use the memory state before or after a load instruction as input for the next memory access. Therefore, M-values hold two pieces of information for each address: the data located at that address, and an *access flag*. A load operation sets the access flag of the address it loads from, thus providing an artificial change to the M-value.

We now need an SMT sort that can represent M-values. Program verifiers usually model memory using the SMT theory of arrays [22] or an Ackermannized variant of it [16]. However, we found this approach to be unsuitable for our needs: During CEGIS, we have to prove that no memory state is a counterexample for our current synthesis candidate. We found that the SMT solver we used ran out of memory when trying to prove this.

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

Because we only consider one machine instruction g at a time as our goal, we can restrict our memory model to only represent those addresses that g uses. If any instruction in the synthesized pattern accesses another address, the pattern cannot have the same behavior as g, and we exclude it. During synthesis, we check that the valid pointer set of each IR operation is a subset of the valid pointer set of g. Since CEGIS uses concrete test cases in its synthesis step, we need not worry about aliasing effects.

Modeling memory with bit vectors. Given a goal instruction g which accesses the pointers $G = V(g, v_a, v_i, v_r)$, its corresponding M-value sort $M(g)$ is a bit vector of size $|G| \cdot (w+1)$, where w is the bit-width of the bytes being addressed.

This bit vector is laid out as follows: Bits $k \cdot (w+1) + 1$ to $k \cdot (w+1) + w$ hold the value located at the address $V(g, v_a, v_i, v_r)[k]$. The extra bit $k \cdot (w+1)$ holds the access flag.

We can now define our primitive load and store functions ld and st, which access w bits at a time. Their task is simply to extract and overwrite the right bits in the M-value. We only need to handle pointers in G as inputs, because patterns that access other pointers are excluded from synthesis.

To define these functions, we use the following abbreviations:

$l(k) = k \cdot (w+1) + 1$	Lower bound of k-th byte
$u(k) = k \cdot (w+1) + w$	Upper bound of k-th byte
$f(k) = k \cdot (w+1)$	Flag for k-th byte

The load function ld takes an M-value m and an address a. It returns an updated M-value with the access flag set, and the value stored at a. We define ld by cases ranging over G: If $a = G[k]$, then

$$ld(m, a) = (m[f(k) \mapsto 1], m[u(k) \ldots l(k)])$$

Similarly, we define st to store the value v into memory represented by m at address a. If $a = G[k]$, then

$$st(m, a, v) = m[n \cdot (w+1) \ldots u(k)+1] \circ v \circ m[l(k)-1 \ldots 0]$$

Example 2 (32 bit store instruction)**.** *To construct wider load and store instructions, we can chain load and store operations, passing M-values from one to the next. This is* *the definition of a 32 bit wide store instruction if $w = 8$:*

$$
\begin{aligned}
S_a(\text{store32}) &= [M(g), \mathsf{BitVec}_{32}, \mathsf{BitVec}_{32}]\\
S_i(\text{store32}) &= []\\
S_r(\text{store32}) &= [M(g)]
\end{aligned}
$$

$$
\begin{aligned}
&P(\text{store32}, v_a, v_i, v_r) = \mathsf{true}\\
&Q(\text{store32}, v_a, v_i, v_r) =\\
&\quad \text{let } m_0 \leftarrow st(v_a[0], v_a[1], v_a[2][7\dots 0]) \text{ in}\\
&\quad \text{let } m_1 \leftarrow st(m_0, v_a[1]+1, v_a[2][15\dots 8]) \text{ in}\\
&\quad \text{let } m_2 \leftarrow st(m_1, v_a[1]+2, v_a[2][23\dots 16]) \text{ in}\\
&\quad \text{let } m_3 \leftarrow st(m_2, v_a[1]+3, v_a[2][31\dots 24]) \text{ in}\\
&\quad v_r[0] = m_3\\
&V(\text{store32}, v_a, v_i, v_r) =\\
&\quad [v_a[1], v_a[1]+1, v_a[1]+2, v_a[1]+3]
\end{aligned}
$$

4.2 Control Flow

We also want to synthesize jump instructions with our tool. 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, too. 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. However, we interpret the basic block references as *results* of the jump instruction, and encode the decision to jump using the values of these results. The jump instructions return the boolean value True for the branch being taken, and False for all others.

Thus, when two jump instructions return the same values given the same arguments in our model, they will perform the same jumps in the IR or machine language.

4.3 Compound instructions

We usually use a single machine instruction as our synthesis goal, but we can also combine a DAG of instructions into one specification called a *compound instruction*.

One example that necessitates compound instructions is that IRs and processor architectures differ in the way in which they handle conditional control flow. For example, a conditional jump in x86 reads the processors flags, which have been set by a previous instruction. Since IRs usually do not use flags, we cannot find an IR equivalent for just a conditional jump node. Instead, we must consider a combination of one comparison and one conditional jump as our goal.

To define a compound instruction, we add a fresh intermediate variable for each argument and each result in the DAG. Then, we construct $P(i)$, $Q(i)$ and $V(i)$ for each instruction i in the DAG. Finally, for each edge in the DAG, we assert that the values it connects are equal.

Example 3 (Compare and jump if less)**.** *If we are given the instructions "cmp" and "jcc", we can model their combination c thus (i_n are the intermediate variables, and the condition code for "less than" is 12):*

$$
\begin{aligned}
P(c, v_a, v_i, v_r) &= Q(\text{cmp}, [i_1, i_2], [], [i_3]) \land\\
&\quad\, Q(\text{jcc}, [i_4], [12], [i_5, i_6])
\end{aligned}
$$

$$
\begin{aligned}
Q(c, v_a, v_i, v_r) &= Q(\text{cmp}, [i_1, i_2], [], [i_3]) \land\\
&\quad\, Q(\text{jcc}, [i_4], [12], [i_5, i_6]) \land\\
&\quad\, i_1 = v_a[0] \land\\
&\quad\, i_2 = v_a[1] \land\\
&\quad\, i_4 = i_3 \land\\
&\quad\, v_r[0] = i_5 \land\\
&\quad\, v_r[1] = i_6
\end{aligned}
$$

$$
\begin{aligned}
V(c, v_a, v_i, v_r) &= V(\text{cmp}, [i_1, i_2], [], [i_3]) \cup\\
&\quad\, V(\text{jcc}, [i_4], [12], [i_5, i_6])
\end{aligned}
$$

This compound instruction is equivalent to a combination of one "set if less than" and one "jump if set" IR operation, which is how IRs would model this control flow [1].

5. Instruction Selection Synthesis

In this section, we discuss the core problem of synthesizing instruction selectors: Given a goal 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 must run a separate synthesis for each possible assignment to them. IR operations, on the other hand, may have internal attributes. For example, we have an IR operation Const, which has one result and one internal attribute, and whose postcondition sets $v_r[0] = v_i[0]$. We use this operation to synthesize constants inherent in goal instructions, such as the constant 1 inherent in an increment.

Let us assume that we have S_i^+, P^+, Q^+, and V^+ that extend S_i, P, Q, and V from single operations to patterns. Then, the problem we have to solve is this:

$$
\begin{aligned}
&\exists p : \text{Pattern}(I).\, \exists v_i :: S_i^+(p).\, \forall v_a :: S_a(g).\, \forall v_r :: S_r(g).\\
&\; P^+(p, v_a, v_i, v_r) \land Q^+(p, v_a, v_i, v_r) \implies\\
&\quad (V^+(p, v_a, v_i, v_r) \subseteq V(g, v_a, v_i, v_r) \land\\
&\quad\; P(g, v_a, [], v_r) \land Q(g, v_a, [], v_r))
\end{aligned}
$$

Our task is to combine operations from I into a pattern p, and to find an assignment to p's internal attributes v_i, such that the following holds: If we have any list v_a of arguments that satisfies p's precondition, and we have the list v_r of results of applying p to v_a, then p must only access valid pointers (i.e. those also accessed by g) in computing those results, the precondition of g must be fulfilled, and g must also compute v_r.

In order to solve this task, we need to model patterns as SMT formulas, and implement the $^+$-functions used in the query in this model. For this, we follow the work of Gulwani et al. [10] and discuss the basics of, and our extensions to their approach in the next section.

5.1 Pattern Representation and Semantics

The main concept of Gulwani et al.'s pattern representation is the idea of *locations*. A location is any place in the pattern where data is produced. In a pattern, there are two kinds of locations: Each argument to the whole pattern is a location, and each result of an operation is a location.

Furthermore, Gulwani et al. introduce a set of location variables L. This set contains three kinds of location variables, which specify the locations of (1) operations, (2) operations' arguments, and (3) the pattern's results. Thus, L fully describes the modeled pattern. However, an assignment to L can also model an invalid pattern. To exclude these patterns, Gulwani et al. place a well-formedness constraint on L, which we will call ϕ_{wf}.

Given a well-formed assignment to L, Gulwani et al. now combine the semantics of the operations to obtain a semantics of the whole pattern by adding two additional constraints. The first constraint asserts that each operation's postcondition holds for its arguments and result. The second constraint connects each operation's argument with the data produced at the argument's location. Altogether, we obtain the postcondition Q^+ of the whole pattern.

Since we use a more general instruction model, we have to extend Gulwani et al.'s pattern representation. First, we support operations with multiple results by assigning multiple consecutive locations to them. We model this by adapting Gulwani et al.'s consistency constraint: Let $L(o)$ be the location associated with IR operation $o \in I$, and let g be the goal instruction. Our consistency constraint is then

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

where $\text{distinct}(S)$ holds if all elements of S are pair-wise distinct. This predicate is included in SMT-LIB [2] and therefore supported by all conforming SMT solvers.

Second, we support multiple sorts. Therefore, we add further clauses to the well-formedness constraint that restrict an argument's location to those locations that hold a result of the same sort.

Example 4 (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.

Now that we are able to represent IR patterns and their semantics in the SMT formulas, we are able to enumerate IR patterns and identify those patterns that match our goal instruction.

5.2 Search Algorithm

We now refine our sketch from the beginning of Section 5 in several steps. First, we replace the pattern p from the sketch with the representation we developed in Section 5.1, using the location variables L and the evaluation variables E. The latter hold the argument and result values of the operations, as we have already seen in Example 4. We then arrive at the following formula, where I is again the multiset of IR operations.

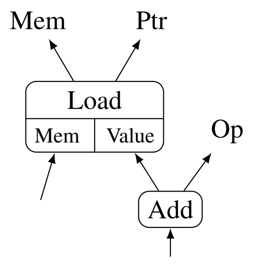
$$\begin{aligned}
&\exists L : \text{LocationVariables}(I).\, \exists v_i :: S_i^+(I).\, \phi_{wf}(I) \wedge \\
&\forall v_a :: S_a(g).\, \forall v_r :: S_r(g).\, \exists E : \text{EvalVariables}(I). \\
&P^+(I, E, v_a, v_i, v_r) \wedge Q^+(I, L, E, v_a, v_i, v_r) \implies \\
&V^+(I, E, v_a, v_i, v_r) \subseteq V(g, v_a, v_i, v_r) \wedge \\
&P(g, v_a, [], v_r) \wedge Q(g, v_a, [], v_r)
\end{aligned}$$

This new formula is in principle solvable by an SMT solver, because the variables in L are integers of finite range, and the variables in E are constrained to single values by Q^+. However, it contains universal quantifiers, and thus solving it still takes an impracticable amount of memory and time. In the second refinement step, we use CEGIS to eliminate the universal quantifiers.

For CEGIS, we need two formulas: The synthesis formula (1) produces a solution that is valid for the current set of test cases, and the verification formula (2) verifies the solution or produces a counterexample test case.

We define a test case to be the list $t_a :: S_a(g)$ of pattern arguments. The set of all current test cases is T. The synthesis formula $\phi_{synth}(I, T, g)$ is then as follows:

$$\begin{aligned}
&\exists L : \text{LocationVariables}(I).\, \exists v_i :: S_i^+(I).\, \phi_{wf}(I) \wedge \\
&\bigwedge_{t_a \in T} \Big(\exists E : \text{EvalVariables}(I).\, \exists v_r :: S_r(g). \\
&\quad P^+(I, E, t_a, v_i, v_r) \implies \\
&\quad\quad V^+(I, E, t_a, v_i, v_r) \subseteq V(g, t_a, v_i, v_r) \wedge \\
&\quad\quad Q^+(I, L, E, t_a, v_i, v_r) \wedge \\
&\quad\quad P(g, t_a, [], v_r) \wedge Q(g, t_a, [], v_r)\Big)
\end{aligned} \tag{1}$$



(a) IR pattern

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

(b) Locations

$$Q^+(I, L, E, v_a, v_i, v_r) = Q(\mathit{Load}, [e_0, e_1], [], [e_2, e_3])$$
$$\wedge\, Q(\mathit{Add}, [e_4, e_5], [], [e_6]) \wedge e_0 = v_a[0] \wedge e_1 = v_a[1]$$
$$\wedge\, e_4 = e_3 \wedge e_5 = v_a[2] \wedge v_r[0] = e_2 \wedge v_r[1] = e_6$$

$$L: \quad l_{\mathit{Load}} = 3 \quad l_{\mathit{Load\text{-}Arg}0} = 0 \quad l_{\mathit{Load\text{-}Arg}1} = 1$$
$$l_{\mathit{Add}} = 5 \quad l_{\mathit{Add\text{-}Arg}0} = 4 \quad l_{\mathit{Add\text{-}Arg}1} = 2$$
$$l_{\mathit{Res}0} = 3 \qquad l_{\mathit{Res}1} = 5$$

(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.

In the synthesis formula, we have replaced the universal quantifiers with a conjunction over the set of test cases.

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 this verification formula:

$$\begin{aligned}
&\exists t_a^* :: S_a(g).\, \exists v_r :: S_r(g).\, \exists v_r' :: S_r(g).\\
&\exists E : \mathrm{EvalVariables}(I).\\
&P^+(I, E, t_a^*, v_i^*, v_r) \wedge Q^+(I, L^*, E, t_a^*, v_i^*, v_r) \wedge\\
&Q(g, v_a, [], v_r') \wedge\\
&(\neg P(g, t_a^*, [], v_r) \vee v_r \neq v_r' \vee\\
&V^+(I, E, t_a^*, v_i, v_r) \not\subseteq V(g, t_a^*, v_i, v_r))
\end{aligned} \tag{2}$$

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 counterexample input 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 one of these is the case, 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. [10].

5.2.1 Finding All Patterns

Many machine instructions have multiple different equivalent IR patterns. To produce a complete instruction selector, we have to find all those patterns that can occur in input programs. However, we can assume that the compiler has performed several optimizations before the instruction selection phase. Hence, not all patterns are equally likely to occur during instruction selection. For example, it is reasonable to assume that the compiler has already simplified $\mathrm{not}(x) + 1$ to $-x$ before instruction selection. To prioritize the patterns that are more likely to be exposed to instruction selection, we allow operations to be annotated with a cost. The cost of a pattern is then the sum of the costs of the operations it consists of. Our goal is to find all equivalent patterns with minimal cost.

The iterative CEGIS algorithm (discussed below) explores IR operation multisets in the order of increasing cost. We only have to extend the synthesis formula to be able to synthesize different patterns from the same multiset of IR operations. We do this by repeating the CEGIS algorithm, excluding the patterns 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 : $\mathrm{LocationVariables}(I)$ 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.

5.3 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 3) takes the simple set of IR operations I as input (containing each operation only once). Then, it produces multisets I' with elements taken from I in order of increasing cost. It then uses the I' as input for classical CEGIS, and forms the union of the results. When it has found at least one result for an I', the algorithm terminates after it has exhausted all multisets of equal cost. Thus, we find all patterns with minimal cost.

Note that we do not prefer any combination of IR operations apart from those with less cost. We could easily make the synthesis faster by preferring combinations of operations that usually occur together in machine instructions. However, we do not want to impose any preconceptions as to the form of machine instructions on our algorithm, and we might not find all possible patterns if we did so.

Costly operations The iteration produces multisets in order of increasing cardinality. However, if there are operations with a cost greater than 1 (*costly* operations), pattern

Algorithm 3 Iteration over all multisets

```
1: procedure MultisetIteration($I$ : {Instruction},
       $g$ : Instruction)
2:     Arbitrarily order $I$
3:     $\ell \leftarrow 1$
4:     $R \leftarrow \emptyset$
5:     while $R = \emptyset$ do
6:         $K = [0, \dots, 0]_{0 \le * < \ell}$
7:         loop
8:             $I' \leftarrow \{\{I[K[0]], \dots, I[K[\ell - 1]]\}\}$
9:             Collapse $I'$, skip iteration if discarded
10:            Skip iteration if a skip condition holds
11:            $R \leftarrow R \cup$ CEGISAllPatterns($I'$, $g$)
12:            $k \leftarrow \max\{k \mid K[k] < \ell - 1\}$
13:            if no $k$ exists then
14:                break
15:            end if
16:            $K[k] \leftarrow K[k] + 1$
17:            $K[k + 1, \dots, \ell - 1] \leftarrow [K[k], \dots, K[k]]$
18:        end loop
19:        $\ell \leftarrow \ell + 1$
20:    end while
21: end procedure
```

cardinality is not equal to pattern cost. In order to iterate in order of increasing cost, we *collapse* I' after generating it. For a costly operation o of cost c, we replace c instances of o with a single instance. If o does not occur a multiple of c times in I', we discard I'. For example, assume that the only costly operation is Mul, with cost 2. Then, we collapse {{Add, Mul, Mul, Store}} to {{Add, Mul, Store}}, and discard {{Add, Mul, Mul, Mul, Store}}.

Skipping iterations In some cases, we can know a priori that no valid pattern can be produced from a certain I'. We say that a *skipping condition* holds for I' in these cases, and skip synthesis for it. We implement two skipping conditions:

- 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 skipping condition is useful in skipping all patterns with memory access operations if the goal instruction does not access memory.

Search space estimate In the following, we assume that we have 22 different IR operations, of which each instruction uses at most 6. Being totally naïve, this would lead to a classical CEGIS problem with $|I| = 132$. We give classical CEGIS the benefit of the doubt though, and assume that I contains each instruction only as often as any pattern needs it. This comes to $|I| = 25$.[2]

To estimate the search space of both approaches, we only consider the number of possible arrangements of components in the pattern. In the case of classical CEGIS, there are 25! possible arrangements of instructions, which is a search space of $\approx 2^{84}$. With iterative CEGIS, we have to iterate over $\binom{|I|+\ell-1}{\ell}$ multisets of cardinality ℓ, each with $\ell!$ possible arrangements. Taking our example numbers, we have a search space of $\sum_{\ell=1}^{6} \binom{22+\ell-1}{\ell} \cdot \ell! \approx 2^{28}$ patterns for iterative CEGIS. In this estimate, we have not yet taken into account the number of possibilities to choose arguments for operations, and the ability of iterative CEGIS to skip iterations.

Of course, classical CEGIS could ignore all permutations which differ only in locations greater than those of all pattern results. However, it is not clear how to convey this information to the SMT solver. Gulwani et al.'s work contains no optimization in this respect.

5.4 Code Generation

From the results of the synthesis, we can easily extract the IR DAG patterns required for the code generator. In order to combine the results with those from earlier runs, or runs on other machines, we first serialize the patterns to the pattern database, and later reload a combined pattern list to generate 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 [3], expect those based on macro expansion.

6. Limitations

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 or by the interfaces we have to conform to:

Floating-Point Arithmetic We did not cover 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 [2, 19], and the SMT solver Z3 has preliminary support for it, but uses a "bit-blaster".

[2] An addition with its destination operand in memory and full address mode needs the following set of operations: Scale, Add, Add, Load, Add, Store

A bit-blaster takes an SMT query, and, without further optimization, translates it to a SAT query, which it then solves. Because any knowledge of the underlying arithmetic is lost in this process, the performance is not acceptable for our needs.

Patterns with Multiple Roots A DAG-matching instruction selector requires all IR patterns to have a single root node, i.e. a node from which the whole pattern is reachable, and which produces all the pattern's results.

However, the x86 instruction set has instructions which load an operand from memory and then perform an arithmetic operation on it. These instructions do not have a single root: The Load operation produces a new M-value as its result, which is not used by the arithmetic operation, but is a result of the whole pattern.

The compiler we integrate with treats these "source address mode" instructions specially. In order to support at least this important class of multiply-rooted patterns, we replicate its behavior.

This limitation is only due to our need to integrate the synthesized instruction selector with an existing compiler. If we were to design an instruction selector form scratch, we could work around this problem.

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 `set`*`cc`* 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. Therefore, we would 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- oder 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 [12] is now part of Z3, although it has not yet been used in program synthesis.

Multiplication and Division Bit vector multiplication and division are especially hard for SMT solvers: The bit-wise definition of multiplication is quadratic in the bit-width of the operand, and division is usually specified indirectly through multiplication. With our tool, we made the following observations: We can provide multiplication operations as parts of our set I without much performance impact. If we try to synthesize a multiplication machine instruction, we can rule out wrong patterns with the same speed as with other goal instructions. It is only the verification of the correct pattern that takes impractically long (more than 8 hours). 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.

On the other hand, the 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.

7. Evaluation

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

7.1 Setup

For the evaluation, our goal is to generate a (partial) instruction selection for the LIBFIRM compiler [15]. 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 instruction selection first checks the synthesized patterns and falls back to existing patterns if no synthesized pattern matches.

We consider two setups for the synthesis: The *basic* setup only contains the register variants of the machine instructions, whereas the *full* setup contains all variants. The two setups aim for different goals. The basic setup wants to minimize the synthesis time while having the same IR coverage as the full setup. On the other hand, the full setup wants to maximize the quality of the instruction selection at the costs of an increased synthesis time. Since our modular approach allows to iteratively add new goal instructions to the basic setup, the two chosen setups shows the range of all possible setups with the same IR coverage.

The synthesis as well as the measurements are per performed on an Intel Core i7-3770 3.40 GHz with 16 GB RAM. The machine runs a 64-bit Ubuntu 16.10 distribution that uses the 4.8.0-26-generic version of the Linux kernel.

7.2 Synthesis

In the following, we want to investigate the synthesis time. Since we allow for a 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 2 min 39 s to synthesize the patterns and 1 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 110 h 53 min 04 s to synthesize the patterns and 1 min 47 s to generate the instruction selection code.

For a comparison with Gulwani et al.'s approach [10], we adapt our tool to use classical CEGIS instead of our iterative approach. We then tried to synthesize an x86 addition instruction with its destination in memory. This instruction uses 3 IR operations (Load, Add, Store) and takes 7.7 seconds to synthesize with our iterative approach. Running the classical CEGIS algorithm on the same machine, the synthesis did not finish within 64 hours.

7.3 Generated Instruction Selection

In this section, we want to evaluate the quality of our generated instruction selector. For this purpose, we compile the SPEC CINT2000 benchmarks with our generated instruction selector and the existing instruction selector of libFirm. When compiling with the generated instruction selector, we measure the ratio of IR operations that it can translate. Furthermore, we compare the time taken by the instruction selection phase, as well as the runtime and number of executed instructions of the generated executables.

When comparing the time taken by the instruction selector, we observe that the basic setup takes about 2.39 × as long as the existing instruction selector, with the whole compiler backend taking 16 % longer. However, the full setup takes 212 × as long as the existing instruction selector, and the whole backend takes 13.6 × as long. We suspect that this is due to two factors. First, our code generator is a prototype that matches one pattern at a time. With a more advanced

Group	#Goals	Patterns		Synthesis Time
		#	Size	
Basic	26	381	5	2 min 39 s
LoadAM	16	141	5	8 min 53 s
StoreAM	16	141	5	5 min 30 s
UnopAM	46	532	7	32 h 43 min 47 s
BinopAM	170	2556	7	43 h 49 min 37 s
Shift	51	426	7	25 h 48 min 43 s
Lea	13	136	4	2 min 05 s
UnopExtra	2	30	3	13 s
BinopExtra	6	78	5	23 min 02 s
Flags	36	19776	7	7 h 48 min 35 s
Total	382	24197	7	110 h 53 min 04 s

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. Starting with LoadAM, the instruction groups contain multiple variants of the following x86 instructions: `mov`; `mov`; `neg`, `not`, `inc`, `dec`; `add`, `and`, `or`, `sub`, `xor`; `shl`, `shr`, `sar`; `lea`; `blsi`, `blsr`; `btc`, `btr`, `bts`, `andn`, `rol`, `ror`; `cmp`, `test`, `jcc`, `jmp`. In contrast, the basic setup only contains the basic variants of these instructions, excluding `add`, `blsi`, `blsr`, `btc`, `btr`, `bts`, `andn`, `rol`, and `ror`.

pattern matching algorithm, we expect a significant performance improvement. Second, we did not exploit normalization invariants of the IR that allow the existing instruction selector to consider fewer patterns.

Considering the coverage of our instruction selector, Table 3 shows that it transforms 70.96 % of all IR operations on average. The remaining operations include function calls, operations involving other bit widths than 32 bit, and variadic ϕ-functions.

Table 3 also compares the runtime of the generated executables. 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 1.44 % and 8.73 %, respectively. However, the 255.vortex benchmark shows an significant improvement of the synthesized instruction selector compared to the handwritten one. On the other hand, the 186.crafty benchmark shows an increased runtime of 124.74 % with the basic setup. This slowdown originates in the extensive use of bit operations and demonstrates the importance of a good instruction selection.

Since measured execution times may vary, we back up our measurements by using the valgrind tool [17] to count the number of executed instructions. Table 4 shows the corresponding results for executables generated by the different instruction selectors. In general, the differences in the number of instructions are more significant than the measured

Benchmark	Synthesized			Handwritten	$\frac{\text{Full}}{\text{Handwritten}}$	$\frac{\text{Basic}}{\text{Handwritten}}$
	Coverage	Full	Basic			
164.gzip	60.09 %	61.56 s	63.82 s	61.08 s	100.78 %	104.48 %
175.vpr	61.51 %	49.18 s	49.84 s	47.06 s	104.49 %	105.91 %
176.gcc	77.10 %	24.25 s	25.79 s	23.87 s	101.59 %	108.07 %
181.mcf	86.89 %	21.47 s	22.80 s	21.27 s	100.93 %	107.17 %
186.crafty	81.19 %	30.45 s	37.35 s	29.94 s	101.71 %	124.74 %
197.parser	71.00 %	58.97 s	63.84 s	58.26 s	101.22 %	109.57 %
253.perlbmk	70.58 %	55.21 s	59.96 s	52.26 s	105.66 %	114.75 %
254.gap	65.12 %	27.69 s	28.97 s	27.03 s	102.41 %	107.16 %
255.vortex	75.12 %	45.95 s	49.70 s	47.17 s	97.42 %	105.38 %
256.bzip2	67.83 %	49.45 s	52.43 s	49.76 s	99.37 %	105.36 %
300.twolf	68.70 %	64.89 s	67.79 s	64.56 s	100.51 %	105.01 %
Geom. Mean	70.96 %				101.44 %	108.73 %

Table 3: Run time of generated executables for different instruction selections. Handwritten refers to the greedy instruction selection implemented in LIBFIRM, whereas Full and Basic refers to the synthesized instruction selections using the corresponding synthesis setups. The coverage column shows the ratio of IR operations translated by the synthesized instruction selector.

Benchmark	Handwritten	Full	Basic	$\frac{\text{Full}}{\text{Handwritten}}$	$\frac{\text{Basic}}{\text{Handwritten}}$
164.gzip	301 855 577 776	316 947 791 760	364 496 778 614	105.00 %	120.75 %
175.vpr	217 958 003 605	222 590 751 625	273 515 990 579	102.13 %	125.49 %
176.gcc	163 070 829 641	172 735 606 140	206 999 343 151	105.93 %	126.94 %
181.mcf	51 077 666 177	55 014 576 850	76 335 823 489	107.71 %	149.45 %
186.crafty	214 948 836 004	223 875 650 189	305 758 329 884	104.15 %	142.25 %
197.parser	307 070 154 893	323 238 607 369	401 109 642 519	105.27 %	130.62 %
253.perlbmk	omitted due to lacking setjmp/longjmp support in valgrind				
254.gap	227 067 494 209	235 902 934 180	262 257 771 410	103.89 %	115.50 %
255.vortex	387 761 541 189	384 183 996 563	470 292 189 160	99.08 %	121.28 %
256.bzip2	308 801 052 992	326 790 033 610	387 667 595 529	105.83 %	125.54 %
300.twolf	302 318 550 069	303 323 066 348	361 535 199 684	100.33 %	119.59 %
Geom. Mean				103.90 %	127.36 %

Table 4: Number of executed instructions of generated executables for different instruction selections. Handwritten refers to the greedy instruction selection implemented in LIBFIRM, whereas Full and Basic refers to the synthesized instruction selections using the corresponding synthesis setups.

runtimes, since the processor can compensate a bad instruction selection to a certain degree.

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 counter-example guided 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. With respect to the runtime of the compiled programs, we show that the synthesized rules are close to a manually-tuned instruction selector.

Acknowledgments

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] LLVM language reference manual – instruction reference. URL `http://llvm.org/docs/LangRef.html#instruction-reference`.

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

[3] G. H. Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2016. ISBN 9783319340173.

[4] L. de Moura and N. Bjørner. 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, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-78799-0.

[5] J. Dias and N. Ramsey. 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, pages 403–416, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706346.

[6] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec. Generalized instruction selection using SSA-graphs. In K. Flautner and J. Regehr, editors, *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*, pages 31–40. ACM, 2008. ISBN 978-1-60558-104-0. doi: 10.1145/1375657.1375663. URL `http://dl.acm.org/citation.cfm?id=1375657`.

[7] E. Eckstein, O. König, and B. Scholz. Code instruction selection based on SSA-graphs. In A. Krall, editor, *Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings*, volume 2826 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2003. ISBN 978-3540201458. doi: 10.1007/978-3-540-39920-9_5.

[8] P. Godefroid and A. Taly. 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, pages 441–452, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254116.

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

[10] 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 '11, pages 62–73, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993506.

[11] S. Heule, E. Schkufza, R. Sharma, and A. Aiken. 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, pages 237–250, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908121.

[12] K. Hoder, N. Bjørner, and L. de Moura. *μZ– An Efficient Engine for Fixed Points with Constraints*, pages 457–462. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-22110-1. doi: 10.1007/978-3-642-22110-1_36.

[13] D. R. Koes and S. C. Goldstein. Near-optimal instruction selection on DAGs. In M. L. Soffa and E. Duesterwald, editors, *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, pages 45–54. ACM, 2008. ISBN 978-1-59593-978-4. doi: 10.1145/1356058.1356065. URL `http://dl.acm.org/citation.cfm?id=1356058`.

[14] 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*, pages 75–88. IEEE Computer Society, 2004. ISBN 978-0769521022. doi: 10.1109/CGO.2004.1281665. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9012`.

[15] libFirm. libFirm – The FIRM intermediate representation library. URL `http://libfirm.org`.

[16] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737965.

[17] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[18] M. Paleczny, C. A. Vick, and C. Click. The java hotspot server compiler. In S. Wold, editor, *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. USENIX, 2001. ISBN 978-1880446119. URL `http://www.usenix.org/publications/library/proceedings/jvm01/paleczny.html`.

[19] P. Rümmer and T. Wahl. 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*, 2010.

[20] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451150.

[21] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983. ISBN 9780262192187.

[22] C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1929004.1929011`.

[23] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM. ISBN 978-1595934512. doi: 10.1145/1168857.1168907.

[24] H. S. Warren Jr. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012. ISBN 9780321842688.

KIT Scientific Working Papers
ISSN 2194-1629