Efficiently Synthesizing Lowest Cost Rewrite Rules for Instruction Selection

Ross Daly
Stanford University
Stanford, CA, USA
rdaly525@cs.stanford.edu

Caleb Donovick
Stanford University
Stanford, CA, USA
donovick@cs.stanford.edu

Caleb Terrill
Stanford University
Stanford, CA, USA
cterill@stanford.edu

Jackson Melchert
Stanford University
Stanford, CA, USA
melchert@stanford.edu

Priyanka Raina
Stanford University
Stanford, CA, USA
praina@stanford.edu

Clark Barrett
Stanford University
Stanford, CA, USA
barrett@cs.stanford.edu

Pat Hanrahan
Stanford University
Stanford, CA, USA
hanrahan@cs.stanford.edu

***Abstract*—Compiling programs to an instruction set architecture (ISA) requires a set of rewrite rules that map patterns consisting of compiler instructions to patterns consisting of ISA instructions. We synthesize such rules by constructing SMT queries, where solutions represent two functionally equivalent programs. These two programs are interpreted as an instruction selection rewrite rule. Furthermore, we address existing work's limitations by developing a set of optimizations that prevent synthesis of duplicate rules, composite rules, and higher-cost rules. We evaluate our algorithm on multiple ISAs. Without our optimizations, the majority of synthesized rewrite rules are either duplicates or higher cost. With our optimizations, we only synthesize unique and lowest-cost rules, resulting in total speedups of up to 212x.**

I. Introduction

As we approach the end of Moore's law and Dennard scaling, drastically improving computing performance and energy efficiency requires designing domain-specific hardware architectures (DSAs) or adding domain-specific extensions to existing architectures [22]. As a result, many DSAs have been developed in recent years [4], [8], [24], [28], [31], each with its own custom instruction set architecture (ISA) or ISA extension.

Targeting such ISAs from a compiler's intermediate representation (IR) requires a custom library of instruction selection rewrite rules. A rewrite rule is a mapping of an IR pattern to a functionally equivalent ISA pattern. Manual specification of rewrite rules is error-prone, time-consuming, and often incomplete. It is therefore desirable to automatically generate valid rewrite rules.

When specifying instruction selection rewrite rules, there are two common cases. When ISAs have complex instructions, rewrite rules will often map multi-instruction IR patterns to a single ISA instruction. When ISAs have simple instructions, rewrite rules will often map a single IR instruction to a multi-instruction ISA pattern. A rewrite rule generation tool should be able to create rewrite rules for both cases. We call such rewrite rules *many-to-many* rules.

Generating instruction selectors is not a new idea. Most relevant to this work is Gulwani et al. [21] who use a satisfiability modulo theories (SMT) solver to synthesize a loop-free program that is functionally equivalent to a given specification. Their approach is called component-based program synthesis (CBPS), as each synthesized program must include functional components from a given component library. Buchwald et al. [6] use and extend CBPS to efficiently generate multi-instruction loop-free IR programs equivalent to a single ISA instruction program; that is, they solve the many-to-one rewrite rules synthesis problem. However, multi-instruction ISA programs cannot be synthesized.

A problem with both of these algorithms is that they produce many *duplicate* rules, which are removed during a post-processing step. As we show, this adds significant additional cost. Another issue is that CBPS as currently formulated does not incorporate the notion of optimizing for cost. In practice, we often want only the set of lowest-cost rules, making it unnecessary (and expensive) to generate equivalent higher-cost rules.

This paper presents an algorithm for automatically generating a complete set of many-to-many rewrite rules. We address the above issues by preventing the synthesis of both duplicate and high-cost rules at rule generation time, using both narrowing and exclusion techniques. As a further optimization, we generate rules in stages and exclude *composite* rules, i.e rules that can be composed of smaller rules found in previous stages.

These and other optimizations ensure we produce a minimal but complete set of rewrite rules. Compared to previous work, our approach eliminates unnecessary rules and reduces the time required to produce the unique necessary ones.

Our contributions are as follows:

- We define generalized component-based program synthesis (GCBPS) as the task of synthesizing two functionally equivalent programs using two component libraries. We then present an SMT-based synthesis approach inspired by Gulwani et al. to solve it.
- We present an iterative algorithm $genAll$ to generate all unique many-to-many rules up to a given size. We identify a set of equivalence relations for patterns encoded as programs and for rules that map IR programs to ISA programs. We use these relations to enumerate and exclude duplicate rules. Furthermore we directly exclude composite rewrite rules resulting in up to a 47x speedup.
- We present an algorithm $genAll_{LC}$ which generates only lowest-cost rules by incorporating a cost metric. This results in up to an additional speedup of up to 92x.

The rest of the paper is organized as follows. In Section II, we give a background on instruction selection, existing rule generation methods, SMT, and component-based program synthesis. In Section III, we describe a program synthesis query that can be used to generate many-to-many rules. In Section IV, we present an algorithm for generating all unique rewrite rules and define duplicates and composites. In Section V, we present an algorithm for synthesizing all lowest-cost rules. We evaluate both algorithms in Section VI and discuss limitations and further optimizations in Section VII.

II. Background and Related Work

A. Instruction Selection

Instruction selection is the task of translating code in the compiler's intermediate representation (IR) to functionally equivalent code for a target ISA. Typically a library of rewrite rules is used in instruction selection. A rewrite rule is a mapping from an IR pattern consisting of IR instructions to a functionally equivalent ISA pattern consisting of ISA instructions. Such patterns can be expression trees or directed acyclic graphs (DAGs).

Significant work has been devoted to developing rewrite rule tiling algorithms to perform instruction selection [1], [5], [13], [15]–[18], [20], [26], [30]. For each rule in the rule library, a tiling algorithm first finds all fragments from the IR program in which the rule's IR pattern exactly matches that fragment. Then, the instruction selector finds a tiling of these matches that completely covers the basic block and minimizes the total rule cost according to some cost metric.

Simple instruction selectors only handle tree-based IR patterns, which is inefficient when computations are reused. Modern instruction selectors, such as LLVM, do DAG-based matching that allows for both a richer set of rules and higher quality tiling. Koes et al. [26] describe a similar near-optimal DAG-based instruction selection algorithm [5]. An essential requirement in our work is to be able to generate rules that can be used with such modern instruction selectors.

B. Generating Instruction Selectors

Generating instruction selectors from instruction semantics has been a topic of research interest [6], [7], [9], [11], [23]. Dias and Ramsey [11] introduce an algorithm for generating rewrite rules based on a declarative specification of the ISA. While this solves part of the many-to-many rule task, their work relies on an existing set of algebraic rewrite rules for synthesizing semantically equivalent rules. Our work uses SMT theories for the instruction and program semantics. However, incorporating certain kinds of algebraic rewrite rules could be an avenue for future optimizations.

Daly et al. [9] propose a way to synthesize instruction selection rewrite rules from the RTL specification of a processor. Their algorithm requires a set of pre-specified IR patterns. In contrast, we can efficiently synthesize rules that consider all possible multi-instruction IR patterns up to a given size. Their approach for synthesizing complex instruction constants and handling floating point could be combined with the approaches in this paper.

Most relevant to this work is Buchwald et al. [6] who leverage component-based program synthesis to generate rules with multi-IR patterns and single-instruction ISA patterns. In contrast, our work synthesizes rules with both multi-IR patterns and multi-ISA patterns. Our work differs from all of the approaches mentioned above in that it additionally prevents the synthesis of duplicate, composite, and high-cost rewrite rules.

C. Program Synthesis and Equivalence

We use SMT-based program synthesis to enumerate a complete set of instruction selection rewrite rules. In program synthesis enumeration, it is common to remove equivalent solutions [3]. We use an equivalence relation defined in section IV-A to determining equivalent rewrite rules. Observational equivalence (i.e., programs with the same semantics) has been used for de-duplication [2], however this is too aggressive of a filter for rewrite rules and cannot be used. Observational equivalence does not take into account the structure of the program which is essential for rewrite rule pattern matching.

D. Logical Setting and Notation

We work in the context of many-sorted logic (e.g., [14]), where we assume an infinite set of variables of each sort. Terms are denoted using non-boldface symbols (e.g., X). Boldface symbols (e.g., $\mathbf{X}$) are used for sets, tuples, and multisets, whose elements are either terms or other collections of terms. $\mathbf{Y} := (Y_1, ..., Y_N)$ defines a tuple, where $|\mathbf{Y}| = N$ and Y_i refers to the i-th element. $\mathbf{Z} := \{z^n\}$ defines a multiset, where the multiplicity of element z is $n \in \mathbb{N}$. Both ψ and ϕ are used to denote formulas. $\psi(\mathbf{X})$ is a formula whose free variables are a subset of $\mathbf{X}$. We use $\mathcal{M} \vDash \psi(\mathbf{X})$ to denote the *satisfiability* relation between the interpretation $\mathcal{M}$ and the formula ψ. Assuming $\mathbf{X}$ is a collection of variables, $\mathcal{M}_{\mathbf{X}}$

denotes the *assignment* to those variables induced by $\mathcal{M}$. For an assignment α, we write $\alpha \models \psi(\mathbf{X})$ if $\mathcal{M} \models \psi(\mathbf{X})$ for every model $\mathcal{M}$ such that $\mathcal{M}_{\mathbf{X}} = \alpha$.

E. Component-based Program Synthesis

CBPS is a program synthesis task introduced by Gulwani et al. The inputs to the task are:

- A *specification* $\mathbf{S} := (\mathbf{I}^S, O^S, \phi_{spec}(\mathbf{I}^S, O^S))$ containing a tuple of input variables $\mathbf{I}^S$, a single output variable O^S, and a formula $\phi_{spec}(\mathbf{I}^S, O^S)$ relating the inputs and output.
- A *library of components* (e.g.,instructions) $\mathbf{K}$, where the k-th component $\mathbf{K}_k := (\mathbf{I}_k, O_k, \phi_k(\mathbf{I}_k, O_k))$ consists of a tuple of input variables $\mathbf{I}_k$, a single output variable O_k, and a formula $\phi_k(\mathbf{I}_k, O_k)$ defining the component's semantics.

An example component for an addition instruction is shown below using QF_BV where $BV_{[n]}$ is an n-bit sort.

$$((I_0 : BV_{[16]}, I_1 : BV_{[16]}), O : BV_{[16]}, I_0 +_{[16]} I_1 = O)$$

The task is to synthesize a valid program functionally equivalent to the specification using each component from $\mathbf{K}$ exactly once.

For notational convenience, we group together the set of all inputs and outputs of the components: $\mathbf{W} := \cup_{(\mathbf{I}_k, O_k, \_) \in \mathbf{K}} (O_k \cup (\cup \mathbf{I}_k))$. Gulwani encodes the program structure using a *connection constraint*: $\phi_{conn}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W})$. This is a formula representing how the program inputs ($\mathbf{I}^S$) and program output (O^S) are connected via the components. The connections are specified using *location variables* $\mathbf{L}$. We do not go into the details of how location variable encode connections (they are in [21]). It is sufficient for our purposes to know that these are integer variables, and an assignment to them uniquely determines a way of connecting the components together into a program. The *program semantics* ϕ_{prog} are defined as the components' semantics conjoined with the connection constraint:

$$\begin{aligned} &\phi_{prog}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}) := \\ &\quad \left(\bigwedge_k \phi_k(\mathbf{I}_k, O_k) \right) \wedge \phi_{conn}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}). \end{aligned} \tag{1}$$

Gulwani defines a verification constraint that holds if a particular program is both well-formed (specified using a well-formedness constraint ψ_{wfp}) and satisfies the specification ϕ_{spec}:

$$\begin{aligned} &\phi_{verif} := \psi_{wfp}(\mathbf{L}) \wedge \forall \mathbf{I}^S, O^S, \mathbf{W}. \\ &\quad \phi_{prog}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}) \implies \phi_{spec}(\mathbf{I}^S, O^S). \end{aligned} \tag{2}$$

A *synthesis formula* ϕ_{synth} existentially quantifies $\mathbf{L}$ in (2):

$$\begin{aligned} &\phi_{synth} := \exists \mathbf{L}. \forall \mathbf{I}^S, O^S, \mathbf{W}. \\ &\quad \psi_{wfp}(\mathbf{L}) \wedge \left(\phi_{prog}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}) \implies \phi_{spec}(\mathbf{I}^S, O^S) \right). \end{aligned} \tag{3}$$

This formula can be solved using a technique called Counter-Example Guided Inductive Synthesis (CEGIS). CEGIS solves such exist-forall formulas by iteratively solving a series of quantifier-free queries and is often more efficient than trying to solve the quantified query directly. More details are in [21]. For our purposes, we assume the existence of a CEGIS implementation $CEGIS$ which takes an instance of ϕ_{synth} and returns a model $\mathcal{M}$ with the property that $\mathcal{M}_{\mathbf{L}} \models \phi_{verif}$, from which a program that is a solution to CBPS can be constructed.

III. Component-based Program Synthesis for Many-to-Many Rules

Given the IR and ISA instruction sets $\mathbf{K}^{IR}$ and $\mathbf{K}^{ISA}$, Buchwald et al. [6] use CBPS to synthesize rewrite rules. They use a single ISA instruction $\mathbf{k}^{ISA} \in \mathbf{K}^{ISA}$ for the CBPS specification and a subset of the IR instructions for the CBPS components. A solution to the resulting ϕ_{synth} formula gives a program $\mathbf{P}^{IR}$. If $\mathbf{P}^{ISA}$ is the single-instruction program consisting of $\mathbf{k}^{ISA}$, they interpret the pair $(\mathbf{P}^{IR}, \mathbf{P}^{ISA})$ as an instruction selection rewrite rule.

However, Buchwald et al.'s solution is insufficient for generating many-to-many rules, as they cannot synthesize IR and ISA programs that both contain multiple instructions. Instead, two functionally equivalent programs need to be synthesized. We first define an extension to CBPS called Generalized Component Based Program Synthesis (GCBPS) to address this problem. Then we show how to construct a synthesis query whose solutions represent pairs of functionally equivalent programs. David et al. [10] mention a use case for synthesizing multiple programs, but their example does not require the two programs to be functionally equivalent.

A. Generalized Component Based Program Synthesis

We define the GCBPS task as that of synthesizing two programs, $\mathbf{P}^a$ and $\mathbf{P}^b$, represented using location variables $\mathbf{L}^a$ and $\mathbf{L}^b$, given two sets of components $\mathbf{K}^a$ and $\mathbf{K}^b$, two sets of inputs $\mathbf{I}^a, \mathbf{I}^b$ where $|\mathbf{I}^a| = |\mathbf{I}^b|$, and two outputs O^a, O^b where the following conditions hold true:

1) $\mathbf{P}^a$ uses each component in $\mathbf{K}^a$ exactly once.
2) $\mathbf{P}^b$ uses each component in $\mathbf{K}^b$ exactly once.
3) $\mathbf{P}^a$ is functionally equivalent to $\mathbf{P}^b$.

B. Solving GCBPS

We start with the CBPS verification constraint from (2) using components $\mathbf{K}^a$ (and a corresponding set of inputs and outputs $\mathbf{W}^a$), but modify it slightly by introducing variables $(\mathbf{I}^a, O^a)$ that are fresh copies of $(\mathbf{I}^S, O^S)$:

$$\begin{aligned} &\psi_{wfp}(\mathbf{L}^a) \wedge \forall \mathbf{I}^a, O^a, \mathbf{W}^a, \mathbf{I}^S, O^S. \\ &\quad (\phi^a_{prog}(\mathbf{L}^a, \mathbf{I}^a, O^a, \mathbf{W}^a) \wedge \phi_{spec}(\mathbf{I}^S, O^S)) \implies \\ &\quad \left((\wedge_i \; I^a_i = I^S_i) \implies O^a = O^S \right). \end{aligned} \tag{4}$$

Assuming the formulas for both the program and the specification, if their inputs are the same, their outputs must also be the same.

We next replace the specification program with a different component-based program using components $\mathbf{K}^b$ and quantify over that program's inputs $\mathbf{I}^b$, output O^b, and component variables $\mathbf{W}^b$:

$$\phi_{verif} := \psi_{wfp}(\mathbf{L}^a) \wedge \psi_{wfp}(\mathbf{L}^b) \wedge \forall \mathbf{I}^a, \mathbf{I}^b, O^a, O^b, \mathbf{W}^a, \mathbf{W}^b. \tag{5}$$
$$(\phi^a_{prog}(\mathbf{L}^a, \mathbf{I}^a, O^a, \mathbf{W}^a) \wedge \phi^b_{prog}(\mathbf{L}^b, \mathbf{I}^b, O^b, \mathbf{W}^b)) \implies$$
$$((\wedge_i\, I^a_i = I^b_i) \implies O^a = O^b)$$

This is our generalized verification constraint stating the correctness criteria for when two component-based programs are semantically equivalent.

To synthesize such a pair of programs, a synthesis formula ϕ_{synth} is defined by existentially quantifying $\mathbf{L}^a$ and $\mathbf{L}^b$ in the verification formula (5):

$$\phi_{synth} := \exists \mathbf{L}^a, \mathbf{L}^b. \forall \mathbf{I}^a, \mathbf{I}^b, O^a, O^b, \mathbf{W}^a, \mathbf{W}^b. \tag{6}$$
$$\psi_{wfp}(\mathbf{L}^a) \wedge \psi_{wfp}(\mathbf{L}^b) \wedge$$
$$\Big((\phi^a_{prog}(\mathbf{L}^a, \mathbf{I}^a, O^a, \mathbf{W}^a) \wedge \phi^b_{prog}(\mathbf{L}^b, \mathbf{I}^b, O^b, \mathbf{W}^b)) \implies$$
$$((\wedge_i\; I^a_i = I^b_i) \implies O^a = O^b) \Big)$$

As above, we assume that calling *CEGIS* on ϕ_{synth} returns a model $\mathcal{M}$ such that $\mathcal{M}_{\mathbf{L}^a \cup \mathbf{L}^b} \models \phi_{verif}$. This can be converted into a pair of programs $(\mathbf{P}^a, \mathbf{P}^b)$ representing a rewrite rule that is a solution for the GCBPS task. We write $rewriteRule(\mathbf{K}^a, \mathbf{K}^b, \mathcal{M}^a_{\mathbf{L}}, \mathcal{M}^b_{\mathbf{L}})$ for the rewrite rule constructed from a specific model $\mathcal{M}$ using the component sets $\mathbf{K}^a$ and $\mathbf{K}^b$.

IV. Generating All Many-to-Many Rewrite Rules

Buchwald et al. [6] describe an iterative algorithm, *IterativeCEGIS*, to synthesize rewrite rules using CBPS. This algorithm iterates over all multisets of IR instructions up to a given size and only runs synthesis on each such multiset. Compared to running synthesis using all the IR instructions at once, this iterative algorithm works better in practice.

However, *IterativeCEGIS* cannot synthesize rewrite rules with both multi-instruction IR programs and multi-instruction ISA programs. Furthermore, it produces duplicate rewrite rules which are then filtered out in a post-synthesis filtering step. Although the results are correct, this approach is highly inefficient because each call to CEGIS is expensive, and a CEGIS call is made, not just for some duplicate rules, but for every possible duplicate rule. In our approach, we make the requirement that a solution is not a duplicate part of the CEGIS query itself, ensuring that each successful CEGIS query finds a new, non-redundant rewrite rule.

Our iterative algorithm, *genAll*, is shown in Figure 1. It takes as parameters the IR and ISA component sets, $\mathbf{K}^{IR}$ and $\mathbf{K}^{ISA}$ respectively, as well as a maximum number of components of each kind to use in rewrite rules, N^{IR} and N^{ISA}, and iteratively builds up a set $\mathbb{S}_R$ of rewrite rules, which it returns at the end. Line 3 shows that n_1 and n_2 iterate up to these maximum sizes. Line 4 iterates over all

1 $genAll(\mathbf{K}^{IR}, \mathbf{K}^{ISA}, N^{IR}, N^{ISA})$:
2   $\mathbb{S}_R \leftarrow \{\}$
3   **for** $n_1, n_2 \in [1, N^{IR}] \times [1, N^{ISA}]$:
4   **for** $\mathbf{m}^{IR} \in multicomb(\mathbf{K}^{IR}, n_1)$:
5   **for** $\mathbf{m}^{ISA} \in multicomb(\mathbf{K}^{ISA}, n_2)$:
6   **for** $\mathbf{I}^{IR}, \mathbf{I}^{ISA} \in allInputs(\mathbf{m}^{IR}, \mathbf{m}^{ISA})$:
7           $\phi, \mathbf{L}^{IR}, \mathbf{L}^{ISA} \leftarrow$ $GCBPS(\mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{I}^{IR}, \mathbf{I}^{ISA})$
8           $\phi \leftarrow \phi\ \wedge \neg AllComposites(\mathbb{S}_R, \ldots)$
9           $\mathbb{S}_R\ \leftarrow \mathbb{S}_R\ \cup$ $CEGISAll(\phi, \mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{L}^{IR}, \mathbf{L}^{ISA})$
10   **return** $\mathbb{S}_R$

Fig. 1: Iterative algorithm to generate all unique rewrite rules up to a given size.

1 $CEGISAll(\phi, \mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{L}^{IR}, \mathbf{L}^{ISA})$:
2   $\mathbb{S}_R = \{\}$
3   **while** True:
4     $\mathcal{M} \leftarrow CEGIS(\phi)$
5   **if** $\mathcal{M} = \bot$: **return** $\mathbb{S}_R$
6     $\mathbf{R} \leftarrow rewriteRule(\mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathcal{M}_{\mathbf{L}^{IR}}, \mathcal{M}_{\mathbf{L}^{ISA}})$
7     $\mathbb{S}_R \leftarrow \mathbb{S}_R \cup \{\mathbf{R}\}$
8     $\phi \leftarrow \phi \wedge \neg\psi_{dup}(\mathbf{R}, (\mathbf{L}^{IR}, \mathbf{L}^{ISA}))$

Fig. 2: AllSAT algorithm to synthesize all unique rules. Line 8 excludes all rules that are duplicates of the current synthesized rewrite rule.

multisets of elements from $\mathbf{K}^{IR}$ of size n_1 using a standard multicombination algorithm *multicomb* [25] (not shown). Line 5 is similar but for multisets from $\mathbf{K}^{ISA}$ of size n_2. Next, for a given choice of multisets, line 6 enumerates all possible ways of selecting input vectors from those multisets that could create well-formed programs. Line 7 constructs fresh sets of location variables $\mathbf{L}^{IR}$ and $\mathbf{L}^{ISA}$ and returns them along with the instantiated GCBPS synthesis formula (using Equation (6)).[1] Line 8 excludes all *composite rules* from the synthesis search space. Composite rules are rules that can be constructed using the current set of rules $\mathbb{S}_R$ and are thus unnecessary for instruction selection. We discuss this in more detail in Section IV-B. Finally, on line 9, the current set of rules $\mathbb{S}_R$ is updated with the result of calling *CEGISAll*, which we describe next.

Figure 2 shows the *CEGISAll* algorithm. Its parameters are the synthesis formula ϕ, the multisets $\mathbf{m}^{IR}$ and $\mathbf{m}^{ISA}$, and the location variables $\mathbf{L}^{IR}$ and $\mathbf{L}^{ISA}$. It returns a set $\mathbb{S}_R$ of rewrite rules. Initially this set is empty. The algorithm iteratively calls a standard *CEGIS* algorithm to solve the synthesis query, constructing a new rewrite rule $\mathbf{R}$, which is added to the set $\mathbb{S}_R$ of rewrite rules, when the call to *CEGIS* is successful. The iteration repeats until the *CEGIS* query returns $\bot$, indicating that there are no more rewrite rules to be found. Note that after each iteration, the ϕ_{synth} formula is refined by adding

[1] We augment the well-formed program constraint in 6 to prevent synthesizing programs containing dead code and unused inputs. This can be accomplished by enforcing that each input and intermediate value is used in at least one location.

the negation of a formula capturing the notion of duplicates for this rule. We describe how this is done next.

A. Duplicates

Consider the two distinct rules below. As a syntactical convention, infix operators are used for IR patterns and function calls for ISA patterns.

$$I_1 + (I_2 \cdot I_3) \rightarrow add(I_1, mul(I_2, I_3))$$
$$(I_1 \cdot I_3) + I_2 \rightarrow add(I_2, mul(I_1, I_3))$$

The two IR patterns represent the same operation despite the fact that the variable names and the order of the commutative arguments to addition are both different. Both rules would match the same program fragments in an instruction selector and would result in the same rewrite rule application. Thus, we consider such rules to be equivalent and would like to ensure that only one is generated by our algorithm.

We first define a rewrite rule equivalence relation, $\sim_{rule}$. Informally, two rules are equivalent if replacing either one by the other has no discernible effect on the execution of an instruction selection algorithm. We make this more formal by considering various attributes of standard instruction selection algorithms.

Commutative Instructions Modern pattern matching algorithms used for instruction selection try all argument orderings for commutative instructions [5]. We define the commutative equivalence relation $\sim_{C^{IR}}$ as $\mathbf{P}_1^{IR} \sim_{C^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_2^{IR}$ is a remapping of $\mathbf{P}_1^{IR}$'s commutative instruction's arguments.

Same-kind Instructions Programs $\mathbf{P}$ generated by GCPBS have a unique identifier, the program line number, for each instruction. This means that if two instructions of the same kind appear in a program, interchanging their line numbers results in a different program, even though it makes no difference to the instruction selection algorithm. We define the same-kind equivalence relation $\sim_{K^{IR}}$ as $\mathbf{P}_1^{IR} \sim_{K^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_2^{IR}$ is the result of remapping the line numbers for same-kind instructions in $\mathbf{P}_1^{IR}$.

Data Dependency Modern instruction selection algorithms perform pattern matching, not based on a total order of instructions, but on a partial order determined by data dependencies. Many different sequences may thus lead to the same partial order. We define $\sim_{D^{IR}}$ as $\mathbf{P}_1^{IR} \sim_{D^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_1^{IR}$ and $\mathbf{P}_2^{IR}$ have the same data dependency graph.

Rule Input Renaming For a given rewrite rule, the input variables used for the IR program must match the input variables used for the ISA program, but the specific variable identifiers used don't matter. We define the equivalence relation $\sim_{I^{rule}}$ on rules (i.e., pairs of programs) as $\mathbf{R}_1 \sim_{I^{rule}} \mathbf{R}_2$ iff $\mathbf{R}_2$ is the result of remapping variable identifiers in $\mathbf{R}_1$.

Rule Equivalence The first three equivalence relations defined above are for IR programs, but the analogous relations ($\sim_{C^{ISA}}$, $\sim_{K^{ISA}}, \sim_{D^{ISA}}$) for ISA instructions are also useful.

Putting everything together, we define rule equivalence $\sim_{rule}$ as follows.

$$\sim_{IR} \; := \uplus\{\sim_{C^{IR}}, \sim_{K^{IR}}, \sim_{D^{IR}}\} \tag{7}$$
$$\sim_{ISA} \; := \uplus\{\sim_{C^{ISA}}, \sim_{K^{ISA}}, \sim_{D^{ISA}}\} \tag{8}$$
$$\sim_{rule} \; := \uplus\{(\sim_{IR} \otimes \sim_{ISA}), \sim_{I^{rule}}\} \tag{9}$$

Overall IR equivalence is defined as the transitive closure of the union (notated with $\uplus$) of the three individual IR relations. ISA equivalence is defined similarly. Overall rewrite rule equivalence is then defined using the $\otimes$ operator, where $\sim_\otimes = \sim_a \otimes \sim_b$ is defined as: $(a_1, b_2) \sim_\otimes (a_2, b_2)$ iff $a_1 \sim_a a_2$ and $b_1 \sim_b b_2$. Specifically, rule equivalence is obtained by combining IR equivalence in this way with ISA equivalence, and then combining the result with $\sim_{I^{rule}}$ using $\uplus$.

The set of all duplicates of rule $\mathbf{R}$ is the rule equivalence class $[\mathbf{R}]_{rule}$, where $\mathbf{R}' \in [\mathbf{R}]_{rule} \iff \mathbf{R} \sim_{rule} \mathbf{R}'$. ψ_{dup} can be constructed by enumerating all elements of the equivalence class $[\mathbf{R}]_{rule}$.

Narrowing For a particular equivalence relation $\sim_X$, it might also be possible to construct a narrowing constraint $\psi_X(\mathbf{L})$ that reduces the size of the equivalence class for $\mathbf{R}$. A more formal definition of narrowing can be found in [27].

We use two narrowing constraints $\psi_C(\mathbf{L})$ and $\psi_K(\mathbf{L})$ that narrow the commutativity and same-kind relations by enforcing an ordering on commutative arguments and instructions respectively. Using these narrowing constraints for both $\mathbf{L}^{IR}$ and $\mathbf{L}^{ISA}$ results in a speedup which we show in the evaluation.

B. Composite Rules

In addition to excluding duplicates, we exclude any rule whose effect can already be achieved using the current set of generated rules (line 8 of Figure 1). We elucidate this using a simple example. Assume the algorithm just constructed a new query for the multisets $\mathbf{m}^{IR}$, $\mathbf{m}^{ISA}$, and the input $\mathbf{I}^{IR}$ (line 7 of Figure 1) and assume that the rule library $\mathbb{S}_R$ currently contains rules for addition ($I_1 + I_2 \rightarrow add(I_1, I_2)$), and multiplication ($I_1 \cdot I_2 \rightarrow mul(I_1, I_2)$). Consider the following cases.

1) If $\mathbf{I}^{IR} = (I_1), \mathbf{m}^{IR} = \{+\}, \mathbf{m}^{ISA} = \{add\}$, then the rule $I_1 + I_1 \rightarrow add(I_1, I_1)$ will be synthesized by *CEGISAll*. But this rule is a *specialization* of the existing rule for addition. Any use of this specialized rule could instead use the more general and can thus be excluded. Note that we order the inputs on line 6 of Figure 1 to guarantee that the most general version of a rule is found first.
2) If $\mathbf{I}^{IR} = (I_1, I_2, I_3), \mathbf{m}^{IR} = \{+, \cdot\}, \mathbf{m}^{ISA} = \{add, mul\}$, then the composite rule $(I_1 + (I_2 \cdot I_3)) \rightarrow add(I_1, mul(I_2, I_3))$ will be synthesized by *CEGISAll*. Using similar logic, any use of this composite rule could instead use the simpler and more general rules for addition and multiplication, and can thus be excluded. The multiset ordering used in lines 4 and 5 of Figure 1 ensures that subsets are visited before supersets, guaranteeing that smaller rules are found first.

1 $\mathit{genAll}_{LC}(\mathbf{K}^{IR}, \mathbf{K}^{ISA}, N^{IR}, N^{ISA}, \mathit{cost})$:
2 $\quad \mathbf{K}_{sorted} \leftarrow \mathit{sortByCost}(\mathbf{K}^{ISA}, N^{ISA}, \mathit{cost})$
3 $\quad \mathbb{S}_R \leftarrow \{\}$
4 $\quad$ **for** $n \in [1, N^{IR}]$:
5 $\quad\quad$ **for** $\mathbf{m}^{IR} \in \mathit{multicomb}(\mathbf{K}^{IR}, n)$:
6 $\quad\quad\quad$ **for** $\mathbf{m}^{ISA} \in \mathbf{K}_{sorted}$:
7 $\quad\quad\quad\quad c_{cur} \leftarrow \mathit{cost}(\mathbf{m}^{ISA})$
8 $\quad\quad\quad\quad$ **for** $\mathbf{I}^{IR}, \mathbf{I}^{ISA} \in \mathit{allInputs}(\mathbf{m}^{IR}, \mathbf{m}^{ISA})$:
9 $\quad\quad\quad\quad\quad \phi, \mathbf{L}^{IR}, \mathbf{L}^{ISA} \leftarrow$
$\quad\quad\quad\quad\quad\quad \mathit{GCBPS}(\mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{I}^{IR}, \mathbf{I}^{ISA})$
10 $\quad\quad\quad\quad\quad \phi \leftarrow \phi \wedge \neg \mathit{AllComposites}_{LC}(\mathbb{S}_R, c_{cur}, \ldots)$
11 $\quad\quad\quad\quad\quad \mathbb{S}_R \leftarrow \mathbb{S}_R \cup$
$\quad\quad\quad\quad\quad\quad \mathit{CEGISAll}_{LC}(\phi, \mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{L}^{IR}, \mathbf{L}^{ISA})$
12 $\quad$ **return** $\mathbb{S}_R$

Fig. 3: Iterative algorithm to generate all lowest-cost rules. ISA multisets are ordered by cost. *CEGISAll* is modified to exclude rules with duplicate IR programs.

Excluding Composite Rules Only a subset of composite rules built from existing rules need to be excluded for each synthesis query. In general, for a specific query based on $\mathbf{m}^{IR}$, $\mathbf{m}^{ISA}$, and $\mathbf{I}^{IR}$, we exclude composite rules $\mathbf{R} := (\mathbf{P}^{IR}, \mathbf{P}^{ISA})$ that meet the following criteria:

- $\mathbf{R}$ has exactly $|\mathbf{I}^{IR}|$ inputs.
- $\mathbf{P}^{IR}$ has the same components as $\mathbf{m}^{IR}$.
- $\mathbf{P}^{ISA}$ has the same components as $\mathbf{m}^{ISA}$.
- $\mathbf{P}^{IR}$ is built from the IR programs of already-found rules in $\mathbb{S}_R$.
- $\mathbf{P}^{ISA}$ is the result of applying the rewrite rules used to build $\mathbf{P}^{IR}$.

These checks are encapsulated by the call to *AllComposites* on line 8 of Figure 1.

V. Generating All Lowest-Cost Rules

Because all duplicates are excluded, the *genAll* algorithm generates only unique rewrite rules. However, two unique rules can share the same IR pattern. For a particular IR pattern, only the lowest-cost rule is needed for some cost metrics. Knowing the instruction selection cost metric at rule-generation time presents another time-saving opportunity because we can also prevent the synthesis of high-cost rules.

We make a few assumptions about such a cost metric.

- The cost for an instruction selection tiling is equal to the sum of the costs of each tiling rule's ISA program.
- The cost of an ISA program $\mathbf{P}^{ISA}$ only depends on the instruction *contents*, not the program structure. This cost is simply the sum of the cost of each instruction in the program.

While these assumptions are a restriction on the space of possible cost metrics, it is sufficient to represent common ones like code size and energy. If the desired cost metric violates these assumptions, the *genAll* algorithm can still be used. This restricted space of cost metrics has the important property that the cost of any rule that would be synthesized using the components $\mathbf{m}^{ISA}$ can be determined up front as the sum of the cost of each component.

Figure 3 shows our synthesis algorithm updated to only synthesize the lowest-cost rules for each unique IR pattern. The first change is to sort all possible mulitsets of ISA instructions up to size N^{ISA} by cost (lower cost first) (line 2). This ordering ensures that the first rule synthesized for a particular IR program will be the lowest-cost version of that rule. Therefore, after synthesizing a new rule, all rules with a duplicate IR program can be excluded.

The second change excludes rules with duplicate IR programs. A duplicate IR program is defined using the IR equivalence relation:

$$\sim_{IR_{LC}} := \uplus\{\sim_{C^{IR}}, \sim_{K^{IR}}, \sim_{D^{IR}}, \sim_{I^{IR}}\} \tag{10}$$

This is the same definition as (7), but with an additional relation $\sim_{I^{IR}}$ defined as $\mathbf{P}_1^{IR} \sim_{I^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_2^{IR}$ is the result of remapping variable identifiers in $\mathbf{P}_1^{IR}$. The $\mathit{CEGISAll}_{LC}$ function called on line 9 is the same as *CEGISAll*, except that it uses $\sim_{IR_{LC}}$ instead of $\sim_{IR}$ when constructing ψ_{dup}.

The third change modifies *AllComposites* to use the known up-front cost $\mathit{cost}(\mathbf{m}^{ISA})$. To see how this works, we consider again the example from Section IV-B. As before, we assume $\mathbb{S}_R$ currently contains two rules: one for addition $(I_1 + I_2 \rightarrow \mathit{add}(I_1, I_2))$ and one for multiplication $(I_1 \cdot I_2 \rightarrow \mathit{mul}(I_1, I_2))$. We assume the target (ISA) expressions for these rules have cost 5 and 10, respectively. Consider the following situation:

- Suppose $\mathbf{I}^{IR} = (I_1, I_2, I_3), \mathbf{m}^{IR} = \{+, \cdot\}$. It might be possible to synthesize a rule that has IR pattern $(I_1 + (I_2 \cdot I_3))$. We know that the composite rule $(I_1 + (I_2 \cdot I_3)) \rightarrow \mathit{add}(I_1, \mathit{mul}(I_2, I_3))$ would have a cost of 15 since rule costs are additive. Therefore, we can exclude any rule that matches this IR pattern and has $\mathit{cost}(\mathbf{m}^{ISA}) \geq 15$.

To implement this, only one adjustment needs to be made to the conditions in Section IV-B. Instead of requiring $\mathbf{P}^{ISA}$ to have the same components as $\mathbf{m}^{ISA}$, we simply require $\mathit{cost}(\mathbf{P}^{ISA}) \geq \mathit{cost}(\mathbf{m}^{ISA})$, i.e., for rules matching the other conditions, if the ISA program has a cost equal to or greater than cost of the ISA program in the current rule, it is excluded. These conditions are encapsulated by the call to $\mathit{AllComposites}_{LC}$ (line 10).

VI. Evaluation

Our evaluation strategy is threefold. We first show that our algorithm is capable of producing a variety of many-to-many rules. A good set of rewrite rules involves both many-to-one and one-to-many rules. We also show that by removing duplicates and composites, we produce a much smaller set of rewrite rules. Second, we analyze the effect on performance of the optimizations described above. We show that they all significantly reduce the time spent in synthesis. Finally, we show that by using different cost metrics, we can generate different sets of lowest-cost rewrite rules.

A. Implementation

All instructions are formally specified using the hwtypes python library [12], which leverages pySMT [19] to construct (quantifier-free) SMT queries in the theory of bit-vectors. As part of our specification, we provide annotations indicating which instructions are commutative. We use Boolector [29] as the SMT solver and set a timeout of 12 seconds for each CEGIS invocation. Every synthesized rewrite rule is verified as a valid rewrite rule by independently proving that the IR program is functionally equivalent to the ISA program.

B. Instruction Specifications

To evaluate our algorithms, we selected small but non-trivial sets of IR and ISA instructions, all operating on 4-bit bit-vectors.

IR We define the IR instruction set to be constants (0, 1), bitwise operations (not, and, or, xor), arithmetic operations (neg, add, sub), multiplication (mul), unsigned comparison operations (ult, ule, ugt, uge), equality (eq), and dis-equality (neq).

ISA 1 This is a minimal RISC-like ISA containing only 6 instructions: $nand$, sub, three comparison instructions ($cmpZ$, $cmpN$, $cmpC$) which compute the Zero (Z), Sign (N), and Carry (C) flags respectively for a subtraction, and a flag inverting instruction (inv).

ISA 2 This is an ISA specialized for linear algebra. It supports the 5 instructions: neg, add, $add3$ (addition of 3 values), mul, and mac (multiply-accumulate).

C. Rewrite Rule Synthesis

For each ISA we run three different experiments. The first (All Lowest-Cost) generates all lowest-cost rules using the $genAll_{LC}$ algorithm in Figure 3. A code-size cost metric is used, i.e., $cost(\mathbf{K})$ is just the number of components in $\mathbf{K}$. The second (All Unique) generates all unique rules (regardless of cost) by excluding all duplicates and composites using the $genAll$ algorithm. The third (Dup + Comp) synthesizes all rules *including* duplicates and composites. This third experiment is similar to Buchwald et al.'s $IterativeCEGIS$ algorithm.

For ISA 1, we split the rule generation into two parts. The first part (ISA 1a) synthesizes rules composed of bitwise and arithmetic IR instructions using the ISA's $nand$ and sub instructions. The second part (ISA 1b) synthesizes rules composed of constants and comparison instructions using the four instructions $cmpZ$, $cmpN$, $cmpC$, and inv. For 1a, we synthesize rewrite rules up to an IR program size of 2 and an ISA program size of 5 (written 2-to-5). For 1b, we go up to 2-to-4. For ISA 2, we synthesize all rewrite rules composed of constant, and arithmetic (including mul) IR instructions up to size 3-to-2.

The number of rewrite rules produced for ISA 1a, 1b, and 2 are shown in Tables I, II, and III, respectively. Each table entry is the number of rewrite rules synthesized for a particular IR and ISA program size. We limit the total algorithm execution time to be 9 hours and an 'X' indicates when this limit was exceeded. We note the percentage of synthesized rewrite rules that are duplicates or composites in the "% Dup + Comp" row. We also note the percentage of synthesized rewrite rules that are equal or higher cost than the lowest-cost rule in the "% High-Cost" row.

For both ISAs we were able to synthesize 1-to-many and many-to-1 rules for both IR and ISA instructions. For all ISAs, there are no rules synthesized by (Dup + Comp) for which an equivalent rule was not also synthesized by (All Unique). This is evidence that $genAll$ successfully excludes only redundant duplicates, of which there are many. In ISA 2 with two IR instructions, 96.6% of the 3276 (287 + 2989) rewrite rules that were synthesized are duplicates or composites.

Similarly, there are no rules synthesized by (All Unique) for which an equivalent rule was not also synthesized by (All Lowest-Cost), suggesting that $genAll_{LC}$ only excludes higher-cost rules. For ISA 1b, there were 20,237 (959 + 19278) unique rewrite rules using 3 ISA instructions, but 99.7% of them were higher cost.

D. Synthesis Optimizations for $genAll$

In this section, we compare the effect of 4 different optimizations on the total synthesis time required by $genAll$. The first experiment enables only the same-kind narrowing constraint. The second enables only the commutative narrowing constraint. The third only excludes duplicate rules. The fourth only excludes duplicate rules and composite rules. We also compare these experiments with the effect of disabling all optimizations and enabling all optimizations.

For each $GCBPS$ query, we note the time required (t_{sat}) to run $CEGISAll$. Next, we measure the number of unique rules (N_{unique}) found by $CEGISAll$. We then add the pair (N_{unique}, t_{sat}) to our dataset. We plot the cumulative synthesis time versus the number of unique rules found by doing the following. Each data point is sorted by its slope (t_{sat}/N_{unique}). Then, the increase in both t_{sat} and N_{unique} is plotted for each sorted point. Some data points have $N_{unique} = 0$ indicating that every synthesized rule was redundant. This is shown using a vertical slope.

The synthesis time plot for unique rewrite rules for ISA 1b up to size 2-to-2 is shown in Figure 4a. Each optimization found the same number of unique rules compared to the baseline. The two narrowing constraints show 1.9x and 2x speedup respectively. Excluding all duplicates shows a 5.5x speedup. Excluding both duplicates and composites shows a 6.3x speedup. Combining all optimizations shows over a 9.5x speedup for this configuration.

E. Lowest-Cost Rules Optimizations for $genAll_{LC}$

Finally, we compare two additional optimizations for lowest-cost rules. We use the previous optimized (All Unique) experiment as a baseline. The first excludes rules with duplicate IR programs (IR Dup Exclusion). The second excludes rules with duplicate IR programs and higher-cost composite

		Program Size (ISA)														
		All Lowest-Cost					All Unique					Dup + Comp				
		1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Program	1	3	4	2	1	0	3	10	96	X	X	5	32	X	X	X
size (IR)	2	40	74	47	14	6	40	198	1817	X	X	76	1718	X	X	X
% High-Cost							0%	62.5%	97.4%	X	X	46.9%	95.5%	X	X	X
% Dup + Comp												46.9%	88.1%	X	X	X

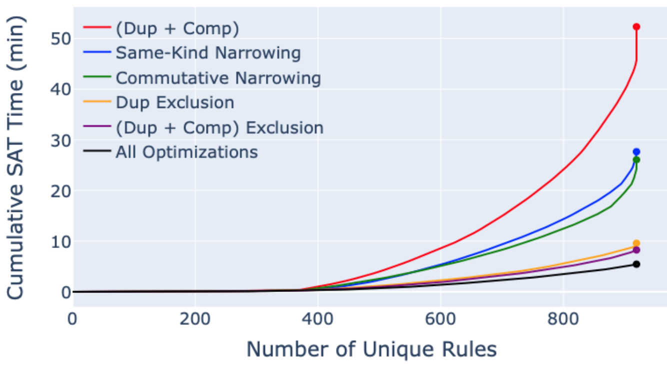
TABLE I: Number of synthesized rewrite rules for ISA 1a.

		Program Size (ISA)											
		All Lowest-Cost				All Unique				Dup + Comp			
		1	2	3	4	1	2	3	4	1	2	3	4
Program	1	7	3	0	0	9	51	959/873	X	17	71	X	X
size (IR)	2	52	112	69	9	78	1004	19278	X	89	3945	X	X
% High-Cost						32.2%	89.1%	99.7%	X	44.3%	97.1%	X	X
% Dup + Comp										17.9%	73.7%	X	X

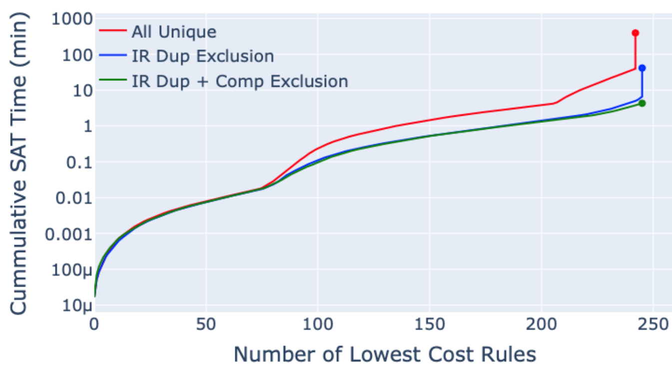
TABLE II: Number of synthesized rewrite rules for ISA 1b.

		Program Size (IR)								
		All Lowest-Cost			All Unique			Dup + Comp		
		1	2	3	1	2	3	1	2	3
Program	1	3	14	311	3	14	316	11	287	X
size (ISA)	2	1	40	834	3	97	1407	10	2989	X
% High-Cost					33.3%	51.4%	33.5%	81%	98.4%	X
% Dup + Comp								71.4%	96.6%	X

TABLE III: Number of synthesized rewrite rules for ISA 2.



(a) *genAll* Optimizations.



(b) $genAll_{LC}$ Optimizations.

Fig. 4: Cumulative time comparison with various optimizations for ISA 1b. Experiments were run on rules up to size 2-to-2 for (a) and size 2-to-3 for (b).

rules (IR Dup + Comp Exclusion). We use the same experimental setup as before with the following difference: In addition to filtering duplicates and composites when computing N_{unique}, all higher-cost rules are also filtered. The synthesis time plot for lowest-cost rewrite rules for ISA 1b up to size 2-to-3 is shown in Figure 4b.

Excluding rules with duplicate IR programs provides a 9.5x speedup while finding 6 additional unique lowest-cost rewrite rules. Also excluding high-cost composites provides a 92x speedup over the baseline (All Unique) configuration.

F. Total Speedup

We summarize the total speedups for all configurations in Table IV. The speedups depend on many parameters including the maximum size of the rewrite rules, the number of possible instructions, the commutativity of the instructions, and the semantics of the instructions. Again, an 'X' in a column indicates that the experiment took longer than 9 hours to run and alludes to the fact that a significant speedup would occur if that experiment were to run to completion. Clearly, the combination of all optimizations discussed in this paper can produce a speedup of several orders of magnitude.

ISA	Rewrite Rule Size (IR, ISA)	All Lowest-Cost Speedup	All Unique Speedup	Total Speedup
1a	(2, 2)	3.2x	24.9x	80.0x
1b	(2, 2)	6.6x	9.6x	63.4x
2	(2, 2)	4.5x	47.1x	212x
1a	(2, 3)	37.7x	X	X
1b	(2, 3)	91.8x	X	X
2	(3, 2)	14.1x	X	X

TABLE IV: Speedups for different configurations.

ISA	Rewrite Rule Size (IR, ISA)	Unique (CS)	Unique (E)	Common
1a	(2, 5)	109	145	82
1b	(2, 4)	147	182	105
2	(3, 2)	104	171	1099

TABLE V: Number of unique and common rewrite rules synthesized for a code size (CS) cost metric and an energy (E) cost metric.

G. Cost Metric Comparisons

Our final experiment explores how the choice of cost metric influences the rules. We have implemented two cost metrics: a code size metric (CS) and an estimated energy metric (E). The energy metric was created to correspond to real hardware energy data. For example the cost ratio for mul and add is $1:1$ for code size, but is $2.5:1$ for energy. The number of common and unique lowest-cost rewrite rules for each ISA is shown in Table V.

While there is some overlap in common rules, each cost metric produces a differing set of unique lowest-cost rules. For code size, the synthesized lowest-cost rule shown in (11), below, uses two $nand$s and a sub. The lowest-cost rule for the same IR pattern with the energy metric uses four $nand$ instructions instead.

$$not(or(I_1, I_2)) \rightarrow sub(nand(I_1, nand(I_2, I_2)), I_2) \quad (11)$$

VII. Conclusion and Future Work

We showed that it is possible to synthesize many-to-many instruction selection rewrite rules for a variety of ISAs using program synthesis.

This supports two major trends in computer architecture. The first is the trend towards simple or reduced instruction architectures where multiple instructions are needed for simple operations. It also supports the trend to introduce more complex domain-specific instructions for energy efficiency. In this case, a single instruction can implement complex operations.

We also showed that our algorithms are efficient. Removing duplicates, composites, and higher-cost rules results in speedups of over a hundred-fold. Despite these drastic speedups, synthesizing many-to-many rewrite rules for modern IRs and ISAs requires further optimizations. Many of our synthesized rules contain program fragments that would have been optimized away by a compiler before instruction selection (e.g., $sub(X, X)$). Excluding these from the synthesis could result in further speedups.

Buchwald et al. [6] presented generalizations for instructions with multiple sorts, multiple outputs, preconditions, and internal attributes, thus enabling the modeling of memory and control flow instructions. The synthesis query and algorithms presented in this paper are orthogonal and could thus be modified to incorporate these features, allowing for a wider variety of possible instruction sets.

As is the case in prior work we limit the synthesis to loop free patterns. Relaxing this constraint and using other instruction selection algorithms would be an interesting research avenue.

We believe this research area is fertile ground and hope our work inspires and enables future research endeavors towards the goal of automatically generating compilers for emerging domain-specific architectures.

References

[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 934–950. Springer, 2013.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 319–336. Springer, 2017.

[4] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, et al. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[5] Eli Bendersky. *A deeper look into the LLVM code generator, Part 1*, Feb 2013.

[6] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 300–313, 2018.

[7] R. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):173–190, 1980.

[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[9] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Synthesizing instruction selection rewrite rules from RTL using SMT. In *Conference on Formal Methods in Computer-Aided Design (FMCAD)*, page 139, 2022.

[10] Cristina David and Daniel Kroening. Program synthesis: Challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150403, 2017.

[11] Joao Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. *ACM Sigplan Notices*, 45(1):403–416, 2010.

[12] Caleb Donovick, Ross Daly, Jackson Melchert, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. Peak: A single source of truth for hardware design and verification. *arXiv preprint arXiv:2308.13106*, 2023.

[13] Helmut Emmelmann, F.-W. Schröer, and Rudolf Landwehr. BEG: A generator for efficient back ends. *ACM Sigplan Notices*, 24(7):227–237, 1989.

[14] Herbert Enderton and Herbert B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[15] Christopher W. Fraser and David R. Hanson. *A retargetable C compiler: Design and implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[16] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):213–226, 1992.

[17] Mahadevan Ganapathi. *Retargetable code generation and optimization using attribute grammars*. PhD thesis, 1980. AAI8107834.

[18] Mahadevan Ganapathi and Charles N. Fischer. Description-driven code generation using attribute grammars. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 108–119, New York, NY, USA, 1982. Association for Computing Machinery.

[19] Marco Gario and Andrea Micheli. PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*, 2015.

[20] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 231–254, New York, NY, USA, 1978. Association for Computing Machinery.

[21] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[22] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[23] Roger Hoover and Kenneth Zadeck. Generating machine specific optimizing compilers. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–229, 1996.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Communications of the ACM*, 61(9):50–59, 2018.

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

[26] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on DAGs. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 45–54, 2008.

[27] Raúl López-Rueda and Santiago Escobar. Canonical narrowing with irreducibility and SMT constraints as a generic symbolic protocol analysis method. In *Rewriting Logic and Its Applications: 14th International Workshop, WRLA 2022, Munich, Germany, April 2–3, 2022, Revised Selected Papers*, pages 45–64. Springer, 2022.

[28] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A Horowitz, Pat Hanrahan, and Priyanka Raina. APEX: A framework for automated processing element design space exploration using frequent subgraph analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 33–45, 2023.

[29] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[30] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application BURS theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 294–308, 1988.

[31] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. *ACM SIGARCH Computer Architecture News*, 45(2):389–402, 2017.