Synthesizing Instruction Selection Rewrite Rules from RTL using SMT

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

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

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

Rajsekhar Setaluri
Stanford University
Stanford, CA, USA
setaluri@stanford.edu

Nestan Tsiskaridze Bullock
Stanford University
Stanford, CA, USA
nestan@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*—Creating a compiler for an instruction set architecture (ISA) requires a set of rewrite rules describing how to translate from the compiler's intermediate representation (IR) to the ISA. We address this challenge by synthesizing rewrite rules from a register-transfer level (RTL) description of the target architecture (with minimal annotations about its state and the ISA format), together with formal IR semantics, by constructing SMT queries where solutions represent valid rewrite rules.**

We evaluate our approach on multiple architectures, supporting both integer and floating-point operations. We synthesize both integer and floating-point rewrite rules from an intermediate representation to various reconfigurable array architectures in under 1.2 seconds per rule. We also synthesize integer rewrite rules from WebAssembly to RISC-V with both standard and custom extensions in under 4 seconds per rule, and we synthesize floating-point rewrite rules in under 8 seconds per rule.

I. Introduction

The end of Moore's law and Dennard scaling means that processor performance will not continue to increase exponentially due to improvements in process technology. Future performance increases will instead be due to the increased efficiency of domain-specific architectures and accelerators. In their Turing Award lecture, John Hennessy and David Patterson envision such a future; they predict that these innovations will lead to a new golden age of computer architecture [33]. In order to realize this vision, there must be a corresponding golden age of software tools, programming models, and compilers to design and program specialized architectures [55].

Every new instruction set architecture (ISA) must be accompanied by a set of rewrite rules to be used in code generation. These rules describe how to transform a compiler's intermediate representation (IR) to the ISA. Crafting these rules is a labor-intensive task and is often performed by someone other than the ISA designer. Hence, the ISA must be carefully documented to support compiler writers—this too is a tedious, error-prone process. Moreover, changes to an ISA require new documentation and new rewrite rules.

This leads to a world where there are very few ISAs, and design space exploration is limited to microarchitectural details. To perform architectural design space exploration, a working compiler is critical to perform realistic benchmarking. The work in this paper arose in the context of the Agile Hardware Project, where one of the primary goals is to facilitate rapid design space exploration for a coarse-grained reconfigurable array (CGRA) [7]. We found that manually maintaining rewrite rules for a rapidly changing architecture was a constant pain point. This experience led us to develop a method for automatically synthesizing instruction selection rewrite rules, which is the primary contribution of this paper. Our method requires a register-transfer *level* (RTL)[1] description of the target architecture, a description of the architectural state, and a description of the instruction format. This method has made possible the efficient and algorithmic exploration of large design spaces [41], as generation of the rewrite rules can be efficiently performed without a human in the loop.

Even for established ISAs, it is easy to overlook nuances that are obvious to the ISA designers. This can lead to inefficiencies in compiled code. For example, the RISC-V ISA does not include equals or not-equals instructions but documents "pseudo operations" for performing them using a subtract and an unsigned less-than (`(x - y) < 1` and `0 < (x - y)` respectively) [2]. Similarly, there are no instructions for less-than-or-equals or greater-than-or-equals, each of which can

[1]Not to be confused with Register Transfer *Language*

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_20

also be implemented as two instruction sequences using a less than and an xor (`(y < x) ^ 1` and `(x < y) ^ 1`, respectively); however, these sequences are not documented.

Using the architecture's RTL, we synthesize rewrite rules by constructing first-order logic queries whose solutions, obtained using a satisfiability modulo theories (SMT) solver, represent instruction selection rewrite rules. Additionally, we propose a methodology for abstracting complex operations, such as floating point operations, which proved too costly for previous SMT-based approaches [12]. While some prior work [18], [19], [49], [12] tackled similar problems, they used manually-defined ISA specifications in the form of enumerated lists of instructions with their parameters and semantics. Using RTL directly has the benefit of avoiding this manual specification step. This is particularly important when doing design space exploration, as it is difficult to maintain both the RTL and a corresponding formal specification for a rapidly changing design. ISA specifications also do not typically capture the instruction format or the instruction decode logic, both of which are needed for an end-to-end correctness argument. In addition to these benefits, using the RTL directly also presents unique challenges which we address. Our main contributions are as follows:

- Formalization of the correctness criteria for a general class of rewrite rules between arbitrary IRs and RTL-based architectures.
- A technique for supporting *parametric* rewrite rules.
- A method for abstracting operations whose semantics are either unknown or too complex to model efficiently (e.g., floating-point operations).
- A methodology for efficiently encoding and solving the rewrite rule synthesis problem using SMT.

In our evaluation, we synthesize rewrite rules from CoreIR (an IR designed for RTL) [16] to a family of CGRAs. We also synthesize rewrite rules from WebAssembly to various RISC-V architectures. We target both the base RISC-V ISA and a number of extensions, including extensions with floating-point operations. All of these tasks can be done in seconds. Additionally, we are able to synthesize short multi-instruction sequences for pseudo-operations such as those mentioned above (whether officially documented or not). These take at most 90 seconds to synthesize.

The rest of this paper is organized as follows. Section II provides background on compilers, instruction selection, rewrite rules, and SMT. Section III formalizes rewrite rules and describes our encoding of the problem into SMT. Section IV presents case studies highlighting the utility and performance of the tool. Section V covers related work and, finally, Section VI provides future steps to take towards a general, automatically-derived compiler.

II. Background

A. Code Generation

Most compilers share a common structure: a front end which translates a high-level language into an IR, an optimizer which optimizes the IR, and a code-generator which translates the IR into a hardware-specific representation (which then may be further optimized for the target architecture). The code generation stage typically involves instruction selection, scheduling, resource assignment, and assembly.

There has been significant work devoted to developing instruction selection algorithms [29], [25], [26], [45], [20], [4], [24], [23], [10] that use a set of pre-defined rewrite rules to translate IR programs to architectural instructions. These rewrite rules are dependent on the target ISA and are usually constructed manually. In this paper, we automatically synthesize rewrite rules from the RTL of target architectures;

Notation	**Meaning**
$BV_{[n]}$	Sort for bitvectors of length n
$+_{[n]}, -_{[n]}, \times_{[n]}, \div_{[n]}$	Arithmetic modulo 2^n
$+_{f[n]}$	n-bit floating-point addition
$x \circ y$	Bitvector concatenation
$x[msb : lsb]$	Bitvector extraction
$\text{ite}(c, x, y)$	If-then-else: if c then x else y
$a[i]$	Read from array a at index i
$a[i] := v$	Result of updating array a at index i with value v
$T = C_1(s_1 : \sigma_1) \mid C_2(s_2 : \sigma_2, s_3 : \sigma_3)$	Algebraic data type T with constructors C_1, C_2, testers is_C_1 and is_C_2, and selectors s_i of sort σ_i

TABLE I: Theory-specific notation.

B. Logical Setting

We work in the setting of many-sorted logic (see e.g., [21], [54]). Let S be a set of *sort symbols* (sorts in this setting play a role similar to types in type theory). For every sort $\sigma \in S$, we assume an infinite set of variables of that sort. We assume the usual definitions of terms, literals, formulas, and interpretations, and use $\models$ to denote the satisfiability relation between interpretations and formulas. We write $e\{x \mapsto t\}$ for the result of simultaneously replacing each occurrence of x in e by t. If $\mathbf{x_1}$ and $\mathbf{x_2}$ are two vectors of variables, we write $\mathbf{x_1} :: \mathbf{x_2}$ to denote their concatenation. A term of the form $\text{ite}(\varphi, t_1, t_2)$ is an if-then-else operator, whose meaning is the same as t_1 in an interpretation I where $I \models \varphi$, and the same as t_2 otherwise.

A *theory* $\mathcal{T}$ assigns meaning to certain theory-specific symbols by fixing a class of allowable interpretations (e.g., it may fix the meaning of the symbol '+' to be the addition function). A formula φ is *$\mathcal{T}$-satisfiable* (resp., *$\mathcal{T}$-unsatisfiable*, *$\mathcal{T}$-valid*) if it is satisfied by some (resp., no, all) interpretations in $\mathcal{T}$. The satisfiability modulo theories (SMT) problem is simply the question of determining $\mathcal{T}$-satisfiability of a formula for some given theory $\mathcal{T}$. SMT solvers solve this problem for a standard set of useful theories (and their combinations).

Some examples of common theories supported by SMT solvers include fixed-width bit-vectors, arrays, integer and floating-point arithmetic, uninterpreted functions, and algebraic data types. Table I lists some notation from these theories that we will use in illustrative examples below. A more thorough introduction to SMT can be found in [9].

III. Synthesizing Rewrite Rules

Rewrite rules are a key component in instruction selection, as they indicate the options for how to transform one or more IR instructions into one or more architecture-specific instructions. In this section, we show how to formalize and solve the rewrite rule synthesis problem using SMT.

A. Intermediate Representation Formalization

An intermediate representation (IR) includes a collection of instructions which can be composed together in various ways to represent programs. IR instructions can be represented in many ways, including as graphs or as functions. Here, we represent IR instructions as SMT formulas. The formulas encode how an instruction's inputs are transformed into a set of outputs. Formally, let $\mathbf{x} = (x_1 : \sigma_1, \ldots, x_k : \sigma_k)$ be a vector of variables. Then, the tuple $\mathbf{IR}(\mathbf{x}) = (IR_1(\mathbf{x}), \ldots, IR_l(\mathbf{x}))$ is an IR instruction with k inputs (each x_i is an input) and l outputs (represented by each IR_j). The value of output j for a given concrete input $(c_1, \ldots, c_k)$ is given by constructing the formula $IR_j(c_1, \ldots, c_k)$ and then evaluating it using the semantics of the theory operations in the formula. For example, an 8-bit adder with two outputs, the sum and the carry-out, and inputs x_1 and x_2 of sort $BV_{[8]}$ could be represented as:

$$(x_1 +_{[8]} x_2,\ (0 \circ x_1 +_{[9]} 0 \circ x_2)[8:8]).$$

For the concrete input $(11111111, 00000001)$, the outputs are 00000000 and 1, respectively.

A formula-tuple $\mathbf{IR}$ need not represent only a single instruction. A complex operation or *pseudo-instruction* can be represented as a composition of other instructions. Our SMT representation can easily accommodate composition: if an output IR_j from $\mathbf{IR}_1$ is connected to an input x_i in $\mathbf{IR}_2$, then the composition is simply the result of substituting IR_j for x_i in $\mathbf{IR}_2$, i.e., $\mathbf{IR}_2 \{ x_i \mapsto IR_j \}$. Below, we assume that $\mathbf{IR}$ represents some IR program (comprising one or more IR instructions) that we wish to find a rewrite rule for.

B. Architecture Formalization

An architecture is a circuit that is parameterized by a single architectural instruction value (separate from and not to be confused with the IR instructions mentioned above), which indicates how other inputs and existing states are transformed into outputs and next states. As above, we represent an architecture as a tuple of SMT formulas. The instruction itself is an input to the architecture, which we assume can be modeled as a variable *inst* of sort τ. We further let $\mathbf{y} = (y_1 : \tau_1, \ldots, y_m : \tau_m)$ be a vector of variables with sorts in Σ, where τ_i is the sort of the architecture's i'th input. The tuple $\mathbf{Arch}(inst, \mathbf{y}) = (Arch_1(inst, \mathbf{y}), \ldots, Arch_n(inst, \mathbf{y}))$ is an architecture with $m+1$ inputs and n outputs. As an example, consider an 8-bit ALU with 4 operations. An input *inst* of sort $BV_{[2]}$ selects which operation to perform on two other inputs, y_1 and y_2, both of sort $BV_{[8]}$. Its single output is also of sort $BV_{[8]}$. For this example, $\mathbf{Arch}$ could be:

$$\begin{aligned}&(\mathrm{ite}(inst = 00,\ y_1 -_{[8]} y_2,\\ &\quad \mathrm{ite}(inst = 01,\ y_1 +_{[8]} y_2,\\ &\qquad \mathrm{ite}(inst = 10,\ y_1 *_{[8]} y_2,\ y_1 \div_{[8]} y_2)))).\end{aligned}$$

States. Architectures with states can be modeled by including current state values as inputs and next state values as outputs. Suppose $\mathbf{z} = (z_1 : \omega_1, \ldots, z_p : \omega_p)$ are variables representing the states. Then, we can represent the architecture as:

$$\begin{aligned}&\mathbf{Arch}(inst, \mathbf{y}, \mathbf{z}) =\\ &\quad (Arch_1(inst, \mathbf{y}, \mathbf{z}), \ldots, Arch_n(inst, \mathbf{y}, \mathbf{z}),\\ &\quad Arch_{n+1}(inst, \mathbf{y}, \mathbf{z}), \ldots, Arch_{n+p}(inst, \mathbf{y}, \mathbf{z})),\end{aligned}$$

where $Arch_{n+i}$ are formulas that encode the next-state function for the i^{th} state variable. An example with states appears in Section III-C, below.

Composing Architectures. A rewrite rule for an IR program might require more than one instruction at the architectural level. Fortunately, as was the case for IR programs, it is straightforward to compose multiple architectures using our SMT representation. Let $\mathbf{Arch}_1(inst_1, \mathbf{y_1}, \mathbf{z_1})$ and $\mathbf{Arch}_2(inst_2, \mathbf{y_2}, \mathbf{z_2})$ be two architectures with m_1 and m_2 inputs, p_1 and p_2 states, and n_1 and n_2 outputs, respectively, and suppose that output i of $\mathbf{Arch_1}$ is passed into input j of $\mathbf{Arch_2}$. Let $inst = (inst_1, inst_2)$, $\mathbf{y} = \mathbf{y_1} :: (y_{2,1}, \ldots, y_{2,j-1}, y_{2,j+1}, \ldots, y_{2,m_2})$, $\mathbf{z} = \mathbf{z_1} :: \mathbf{z_2}$, and $\mathbf{y_2'} = \mathbf{y_2} \{ y_{2,j} \mapsto Arch_{1,i}(inst_1, \mathbf{y_1}, \mathbf{z_1}) \}$. Then, the composition is:

$$\begin{aligned}&\mathbf{Arch}(inst, \mathbf{y}, \mathbf{z}) =\\ &(Arch_{1,1}(inst_1, \mathbf{y_1}, \mathbf{z_1}), \ldots, Arch_{1,n_1}(inst_1, \mathbf{y_1}, \mathbf{z_1}),\\ &Arch_{2,1}(inst_2, \mathbf{y_2'}, \mathbf{z_2}), \ldots, Arch_{2,n_2}(inst_2, \mathbf{y_2'}, \mathbf{z_2}),\\ &Arch_{1,n_1+1}(inst_1, \mathbf{y_1}, \mathbf{z_1}), \ldots, Arch_{1,n_1+p_1}(inst_1, \mathbf{y_1}, \mathbf{z_1}),\\ &Arch_{2,n_2+1}(inst_2, \mathbf{y_2'}, \mathbf{z_2}), \ldots, Arch_{2,n_2+p_2}(inst_2, \mathbf{y_2'}, \mathbf{z_2})).\end{aligned}$$

C. Rewrite Rule Formalization

A rewrite rule defines how a specific IR program can be implemented using one or more instructions of a particular architecture. We start with a simple but incomplete definition of a rewrite rule and incrementally build up a definition with more generality and sophistication. The simplest rewrite rule is a tuple $(\mathbf{IR}, \mathbf{Arch}, inst_c)$, where $\mathbf{IR}$ is an IR program, $\mathbf{Arch}$ is an architecture (without states for now), and $inst_c$ is a concrete constant (i.e., a constant that maps to a particular domain value, like 0 or 1) of sort τ. We say such a tuple is a *valid* rewrite rule if the following formula is well-formed and T-valid:

$$\forall \mathbf{x}.\ \mathbf{Arch}(inst_c, \mathbf{x}) = \mathbf{IR}(\mathbf{x}) \qquad (1)$$

Note that well-formedness requires that $\mathbf{Arch}$ and $\mathbf{IR}$ have the same number of inputs and outputs and that corresponding inputs and outputs have the same sort. As an example, take again the sum output of the IR program given in Sec. III-A, that is, $\mathbf{IR} = (x_1 +_{[8]} x_2)$, and suppose $\mathbf{Arch}$ is as given in Section III-B. Then, (1) holds when $inst_c = 01$, and so $(\mathbf{IR}, \mathbf{Arch}, 01)$ is a valid rewrite rule. In practice, things can be more complicated in several ways, which we address next.

Bindings. One problem with (1) is that the inputs and outputs of the IR rarely match those of the architecture. A more general rewrite rule is $(\mathbf{IR}, \mathbf{Arch}, inst_c, \boldsymbol{b}^{in}, \boldsymbol{b}^{out})$, where $(\boldsymbol{b}^{in}, \boldsymbol{b}^{out})$ is a pair of formula tuples, called a *binding*, that specifies how to map between the inputs and outputs of the two formulas. The rewrite rule is valid if the following formula is well-formed and valid:

$$\forall \mathbf{x}.\, \boldsymbol{b}^{out}(\mathbf{Arch}(inst_c, \boldsymbol{b}^{in}(\mathbf{x}))) = \mathbf{IR}(\mathbf{x}). \qquad (2)$$

Here, well-formedness means $\boldsymbol{b}^{in}(\mathbf{x}) = (b_1^{in}(\mathbf{x}), \ldots, b_m^{in}(\mathbf{x}))$, where each $b_i^{in}(\mathbf{x})$ has sort τ_i. We also require $\boldsymbol{b}^{out}(\mathbf{w}) = (b_1^{out}(\mathbf{w}), \ldots, b_l^{out}(\mathbf{w}))$, where $\mathbf{w} = (w_1, \ldots, w_n)$, the sort of each w_i matches $Arch_i$, and the sort of each b_j^{out} matches IR_j. As an example, consider $\boldsymbol{b}^{in} = (x_2, x_1)$ and $\boldsymbol{b}^{out} = (w_2)$. This binding swaps the two IR inputs and only uses the second architecture output.

Another complexity with bindings is that sometimes it is necessary to map the IR inputs to only a subset of the architecture inputs (for example, mapping a unary IR operation to an ISA supporting only binary operations). The extra inputs which do not correspond to any IR input must not have any effect on the output. To model this, we extend $\boldsymbol{b}^{in}$ so that, in addition to $\mathbf{x}$, it also takes additional arguments $\mathbf{y} = (y_1 \ldots y_m)$ with sorts $(\tau_1, \ldots, \tau_m)$. The idea is that the binding can choose to simply map some variable y_i to an extra architecture input. With this extension, we can write the new rewrite rule formula as follows:

$$\forall \mathbf{x}, \mathbf{y}.\, \boldsymbol{b}^{out}(\mathbf{Arch}(inst_c, \boldsymbol{b}^{in}(\mathbf{x}, \mathbf{y}))) = \mathbf{IR}(\mathbf{x}). \qquad (3)$$

Finally, we can handle the full generality of architectures with states by including these in the binding as well, where $\boldsymbol{b}^{in}$ is extended to be a function of sort $(\sigma_1 \ldots \sigma_k, \tau_1 \ldots \tau_k, \omega_1 \ldots \omega_p) \rightarrow (\tau_1 \ldots \tau_m, \omega_1 \ldots \omega_p)$, and $\boldsymbol{b}^{out}$ also takes an additional p inputs of sort $\omega_1, \ldots, \omega_p$.

$$\forall \mathbf{x}, \mathbf{y}, \mathbf{z}.\, \boldsymbol{b}^{out}(\mathbf{Arch}(inst_c, \boldsymbol{b}^{in}(\mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{x}). \qquad (4)$$

As an example, consider a simple architecture which either multiplies its inputs and accumulates the result into a register file z (represented by an array variable) at index 0 while outputting the product or performs a subtraction, both outputting the result and storing it at index 1 of the register file. Assume the instruction is of sort $BV_{[1]}$, and the other inputs are of sort $BV_{[8]}$. All operators use 8-bit arithmetic (so we will omit the [8] subscript to ease readabilty). The formula for the architecture is then:

$$\mathbf{Arch}(inst,\ y_1,\ y_2,\ z) = (\text{ite}(inst = 0,\ y_1 * y_2,\ y_1 - y_2),$$
$$\text{ite}(inst = 0,\ z[0] := z[0] + (y_1 * y_2),\ z[1] := y_1 - y_2))$$

Note that the first formula in the $\mathbf{Arch}$ tuple represents the output of the architecture, while the second represents the next state of z. Now, suppose we are searching for a rewrite rule for $\mathbf{IR}(\mathbf{x}) = (x_3 * x_2) + x_1$. One valid rule is $inst_c = 0$, $\boldsymbol{b}^{in}(\mathbf{x},\ \mathbf{y},\ z) = (x_3,\ x_2,\ z[0] := x_1)$, and $\boldsymbol{b}^{out}(\mathbf{w}) = w_2[0]$ (note that w_2, represents the second input to $\boldsymbol{b}^{out}$, which corresponds to the register file state). This rule represents a solution using $inst_c = 0$ when x_1 is the value of $z[0]$, x_2 drives the y_2 input, and x_3 drives the y_1 input. The result is stored at index 0 of the (next state value of the) register file.

D. Rewrite Rule Synthesis

We next formalize the problem of synthesizing rewrite rules. We assume that we are given $\mathbf{IR}$ and $\mathbf{Arch}$ representing an IR program and an architecture, respectively. We must find $inst_c$, $\boldsymbol{b}^{in}$, and $\boldsymbol{b}^{out}$. Starting from (4), we can simply replace $inst_c$, $\boldsymbol{b}^{in}$, and $\boldsymbol{b}^{out}$ with variables to get a (second-order) formula. It is also useful to make the bindings a function of the instruction, as we explain below. Thus, we have:

$$\exists\, inst, \boldsymbol{b}^{in}, \boldsymbol{b}^{out}.\, \forall \mathbf{x}, \mathbf{y}, \mathbf{z}.$$
$$\boldsymbol{b}^{out}(inst, \mathbf{Arch}(inst, \boldsymbol{b}^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{x}). \qquad (5)$$

If (5) holds, then there exists a valid rewrite rule.

In order to use (5) for a practical rewrite rule synthesis algorithm, we must additionally specify what kinds of functions are allowed for $\boldsymbol{b}^{in}$ and $\boldsymbol{b}^{out}$. These functions should tell us how to map the inputs and outputs, but should not introduce extra functionality. For non-state inputs to the architecture, we simply require that the binding either pick a variable in $\mathbf{x}$ or pass through the corresponding variable from $\mathbf{y}$.

For state inputs, there are two[2] cases. For programmable states (states with compile-time addresses that can be written and read by instructions, e.g., a register file), we allow the binding to *update* part of the state with a variable in $\mathbf{x}$. This corresponds to a previous instruction storing its result (the input for the current instruction) in the state. We do this by using array variables for these states and allowing the binding to write to the arrays. Other states, such as the accumulators or other non-programmable registers, are passed through unchanged by the binding. Formally, we require:

$$b_i^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}) = \begin{cases} y_i \text{ or } x_j (1 \le j \le k), \text{ if } i \le m, \\ z_{i-m}, \text{ if } i > m \text{ (non-programmable)} \\ update(z_{i-m}, inst, \mathbf{x}), \text{otherwise,} \end{cases}$$

where $update(z, inst, \mathbf{x})$ is one or more array writes to z at indexes specified by one or more fields in $inst$ and with values from the variables in $\mathbf{x}$. The output binding is similar:

$$b_j^{out}(inst, \mathbf{w}) = \begin{cases} w_i\ (1 \le i \le n + p), \text{ or} \\ read(w_i, inst)\ (n + 1 \le i \le n + p), \\ \text{where } w_i \text{ is programmable,} \end{cases}$$

where $read(w, inst)$ is a read from the array w at an index specified by some field of $inst$. Implicit in this formulation is the requirement that instructions must either directly output their result or write them to programmable state in a single a

[2]A third kind of state with computed addresses (like indirect loads and stores), can be handled in a way similar to [12], or by using the computed address from the architecture and the IR in the output bindings.

cycle. Pipeline registers and other micro-architectural state fall into the category of states which cannot be bound. We discuss possible approaches for handling pipelining in Section VI.

We next explain how to solve (5), subject to the constraints on bindings. But first, we introduce two useful generalizations.

Synthesizing Parametric Rewrite Rules. Sometimes, we are interested in finding a *parameterized* rewrite rule that works for a family of IR nodes (for instance, the family of IR instructions that multiply a constant parameter by some input). Rather than having to discover a different rewrite rule for each value of the parameter, we would like to solve the problem once and have it work for all possible values of the parameter. Formally, let $\mathbf{c}$ be a vector of parameters, and let $\mathbf{IR}(\mathbf{c}, \mathrm{x})$ be a family of IR nodes parameterized by $\mathbf{c}$. Using equation (5) as a starting point, the new rewrite formula becomes:

$$\forall \mathbf{c}.\, \exists inst, \boldsymbol{b}^{in}, \boldsymbol{b}^{out}.\, \forall \mathbf{x}, \mathbf{y}, \mathbf{z}. \\ \boldsymbol{b}^{out}(inst, \mathbf{Arch}(inst, \boldsymbol{b}^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{c}, \mathbf{x}). \quad (6)$$

In other words, we would like there to be an appropriate instruction encoding for each value of the parameter c. As it stands, this formulation is not very useful, as it does not tell us how to connect the instruction to the parameter. However, by Skolemizing (6), we get the following:

$$\exists inst, \boldsymbol{b}^{in}, \boldsymbol{b}^{out}.\, \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}. \\ \boldsymbol{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}(inst(\mathbf{c}), \\ \boldsymbol{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{c}, \mathbf{x}). \quad (7)$$

where now, *inst* is a function from $\mathbf{c}$ to instructions.[3]

Abstracting Complex Operations. Complex operations (e.g., floating-point arithmetic) can present a challenge. However, it is often the case that there are identical complex operations in the IR and in the architecture. We can handle such situations by replacing such complex operations with *uninterpreted functions* [13]. We must be careful about how this is done though. If we simply introduce new function symbols in the formulas for the IR and the architecture, they will be implicitly *existentially* quantified when checking for satisfiability, leading to spurious results as the solver can choose *any* interpretation. Hence, introduced function symbols must be *universally* quantified. Formally, let $\mathbf{Arch}^{abs}$ and $\mathbf{IR}^{abs}$ be the abstract versions of $\mathbf{Arch}$ and $\mathbf{IR}$, respectively, where the complex operations are removed and replaced with a vector of function symbols $\mathbf{f}$. Then, building on (7), we get the following formulation for the fully general rewrite rule synthesis formula:

$$\exists inst, \boldsymbol{b}^{in}, \boldsymbol{b}^{out}.\, \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{f}.\, \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, \mathbf{f}) = \\ \boldsymbol{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), \mathbf{f}, \boldsymbol{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (8)$$

[3]Technically, to maintain logical equivalence, $\boldsymbol{b}^{in}$ and $\boldsymbol{b}^{out}$ should also be functions of $\mathbf{c}$, but for simplicity, we omit this, keeping the restrictions on their form introduced above. We also did not find any additional dependency on $\mathbf{c}$ to be needed in practice.

E. Rewrite Rule Synthesis Implementation

Here, we detail several additional considerations required to solve the rewrite rule synthesis problem formalized above in practice. Specifically, we discuss (i) removing second-order quantifiers; (ii) encoding instructions; (iii) formula optimizations; and (iv) solving algorithm optimizations.

Removing Second-Order Quantifiers. Note that $inst$, $\boldsymbol{b}^{in}$, $\boldsymbol{b}^{out}$, and $\mathbf{f}$ are all quantified functions. In order to use an SMT solver, we first need to find an equivalent formulation using only first-order quantification. For the binding functions, this is straightforward. Given the restrictions outlined above, there are only a finite number of possible binding functions.[4] Let $\mathcal{B}$ be the set of all legal bindings $(\boldsymbol{b}^{in}, \boldsymbol{b}^{out})$. Then, formula (8) is equivalent to

$$\exists inst.\, \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{f}. \bigvee_{(\boldsymbol{b}^{in}, \boldsymbol{b}^{out}) \in \mathcal{B}} \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, \mathbf{f}) = \\ \boldsymbol{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), \mathbf{f}, \boldsymbol{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (9)$$

Unfortunately, just satisfying this formula does not tell us which binding to use, so in practice, we also add an *indicator variable* i, whose value indicates which binding was used. Formally, we extend the notion of binding to a triple $(b, \boldsymbol{b}^{in}, \boldsymbol{b}^{out})$, where b is an integer unique to each binding. Then, our formula becomes:

$$\exists inst, i.\, \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{f}. \bigvee_{(b, \boldsymbol{b}^{in}, \boldsymbol{b}^{out}) \in \mathcal{B}} i = b \wedge \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, \mathbf{f}) = \\ \boldsymbol{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), \mathbf{f}, \boldsymbol{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (10)$$

To remove the quantification on $\mathbf{f}$, we can use Ackermannization [3]. For each function symbol f, we replace each instance of f with a fresh variable of the same sort as the return sort of f and add constraints requiring that if the arguments to any two of those instances of f are equal, then the fresh variables representing those instances are equal too. Assume, for ease of presentation, that $\mathbf{f} = (f)$ and f appears only once in $\mathbf{IR}^{abs}$, with arguments $\mathbf{s}$, and once in $\mathbf{Arch}^{abs}$, with arguments $\mathbf{t}$. Then, (10) is equivalent to[5]

$$\exists inst, i.\, \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, f_1, f_2. \bigvee_{(b, \boldsymbol{b}^{in}, \boldsymbol{b}^{out}) \in \mathcal{B}} i = b \wedge \\ (\mathbf{s} = \mathbf{t} \rightarrow f_1 = f_2) \rightarrow \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, f_1) = \\ \boldsymbol{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), f_2, \boldsymbol{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (11)$$

Encoding Instructions. Above, we have assumed a simple instruction model, where instructions are taken from some sort τ. In practice, an architecture may have a variety of

[4]To ensure finiteness, we limit the *update* operation mentioned above to allow no more updates than there are IR inputs.

[5]With some abuse of notation, if $P(f)$ is a formula containing f, and f_1 is a variable whose sort matches the return sort of f, we write $P(f_1)$ to mean the result of replacing the application of f in P by f_1.

instructions, each with different components. This can be modeled by letting τ be an *algebraic data type* (ADT), with different constructors for each type of instructions. This also solves the problem of how to handle *inst* as a function of $\mathbf{c}$. Some types of instructions allow *immediate* values to be encoded as part of the instruction. For those instructions, we allow a parameter from $\mathbf{c}$ to appear as the immediate value. This is a very limited type of functional dependence on $\mathbf{c}$, but it is sufficient for modeling the kinds of parametric rewrite rules we are interested in.

To see how this works, consider as an example two formats from the RISC-V integer instruction set (RV32I): (i) R-type: register-register instructions; and (ii) I-type: register-immediate instructions. We can model these using the ADT:

$$\begin{aligned} \mathit{INST} = \; & \mathit{RType}(op : BV_{[7]},\ rd : BV_{[5]},\ \mathit{func}3 : BV_{[3]}, \\ & \quad rs1 : BV_{[5]},\ rs2 : BV_{[5]},\ \mathit{func}7 : BV_{[7]}) \mid \\ & \mathit{IType}(op : BV_{[7]},\ rd : BV_{[5]},\ \mathit{func}3 : BV_{[3]}, \\ & \quad rs1 : BV_{[5]},\ imm : BV_{[12]}) \end{aligned}$$

This could be further refined by declaring *op*, *func*3, etc. as additional data types with limited sets of values. To handle the dependence on parametric values, we add a constraint stating that some immediate value is equal to a parameter. For example, if we want to encode the case where the immediate of *IType* is a constant c, we add the constraint $\mathit{is\_IType}(\mathit{inst}) \wedge \mathit{imm}(\mathit{inst}) = c$. To consider many possible mappings of constants to immediates, we use a disjunction over a set of possibilities as we do with bindings.

Formula Optimizations.

For non-trivial designs, it is too expensive to repeat the architecture and IR formulas for every disjunct in the set of bindings. An alternative is to introduce additional variables for the inputs to and outputs from the architecture and to have the bindings operate only on those variables. For ease of presentation, let's go back to formula (5) and write it as:

$$\begin{aligned} & \exists\, \mathit{inst}.\, \forall\, \mathbf{x}, \mathbf{y}, \mathbf{z}. \\ & \quad \bigvee_{(\boldsymbol{b}^{in}, \boldsymbol{b}^{out}) \in \mathcal{B}} \boldsymbol{b}^{out}(\mathit{inst}, \mathbf{Arch}(\mathit{inst}, \boldsymbol{b}^{in}(\mathit{inst}, \mathbf{x}, \mathbf{y}, \mathbf{z}))) \\ & \qquad = \mathbf{IR}(\mathbf{x}). \quad (12) \end{aligned}$$

This is equivalent to:

$$\begin{aligned} & \exists\, \mathit{inst}.\, \forall\, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{u}, \mathbf{v}, \mathbf{w}. \\ & \left(\bigvee_{(\boldsymbol{b}^{in}, \boldsymbol{b}^{out}) \in \mathcal{B}} (\boldsymbol{b}^{out}(\mathit{inst}, \mathbf{u}) = \mathbf{v} \ \wedge\ \boldsymbol{b}^{in}(\mathit{inst}, \mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{w}) \right) \\ & \qquad \to (\mathbf{Arch}(\mathit{inst}, \mathbf{w}) = \mathbf{u} \wedge \mathbf{v} = \mathbf{IR}(\mathbf{x})). \quad (13) \end{aligned}$$

In practice, it can also be inefficient to include memories and register files in the architecture. An alternative is to remove them and add an additional input for every read port and output for every write port. From the point of view of the rewrite rule synthesis, the problem is equivalent. This is the approach we take in our experiments. For example, the RISC-V register file, which has the property that register 0 always holds 0, can be modeled with two formulae:
One for reads:

$$\text{let } r_1 = \begin{cases} 0 & \text{if } \mathit{rs1} = 0 \\ v_1 & \text{otherwise} \end{cases} \quad r_2 = \begin{cases} 0 & \text{if } \mathit{rs2} = 0 \\ v_1 & \text{if } \mathit{rs1} = \mathit{rs2} \neq 0 \\ v_2 & \text{otherwise} \end{cases}$$
$$\text{in } (r_1, r_2)$$

and one for writes: $(\mathrm{ite}(rd = 0, s, v))$

In the first formula, v_1 and v_2 are the values bound into the register file (or more precisely added as inputs to the architecture). r_1 and r_2 represent the values read from the register file. *rs1* and *rs2* are the read addresses calculated by the architecture from its instruction. Note that this is equivalent to having two reads on an array without an intervening update. However, it massively simplifies the task of generating $\boldsymbol{b}^{in}$, as we do not need to reason about how *rs1* and *rs2* will be derived from *inst*.

In the second formula, *rd* is the write address, s represents the previous state of the written register, and v is the value to be written. Similar to the abstraction of reads, this significantly simplifies the generation of $\boldsymbol{b}^{out}$. These simplifications are possible as we do not care about the full state of the register file. We only care about the two indices which are read and the one index that is written.

Solving Strategy. While some SMT solvers have support for quantified formulas, it is well-known that quantified formulas often lead to performance and robustness problems (and indeed, we observed this in preliminary experiments). We therefore adopt an external technique to solve the final quantified SMT queries, all of which are in *exists-forall* form:

$$\exists\, \mathbf{a}.\forall\, \mathbf{b}.\ \phi(\mathbf{a}, \mathbf{b}) \qquad (14)$$

Our technique is inspired by the counter-example guided synthesis (CEGIS) [51] approach introduced in [15] and more formally described in [27]. The algorithm consists of alternating phases. The algorithm first suggests a solution for $\mathbf{a}$ by simply checking the satisfiability of $\phi(\mathbf{a}, \mathbf{b})$. If $\mathbf{a}_c$ is the value found, it then checks whether this works for all values of $\mathbf{b}$ by checking the satisfiability of $\neg\phi(\mathbf{a}_c, \mathbf{b})$. If this is unsatisfiable, then $\mathbf{a}_c$ is a solution for $\mathbf{a}$ in (14). Otherwise, let $\mathbf{b}_c$ be the satisfying value found. We simply update ϕ to be $\phi(\mathbf{a}, \mathbf{b}) \wedge \phi(\mathbf{a}, \mathbf{b}_c)$ and repeat. Essentially, we thus collect many *sample points*, $\mathbf{b}_c$ with the hope that after enough are collected, it will drive the search to find a value for $\mathbf{a}$ that satisfies (14). We found that in our setting of rewrite rule synthesis, this approach works well.

IV. Evaluation

We evaluate the above approach for rewrite rule synthesis by showing the ability to efficiently synthesize rewrite rules in two settings. First, we synthesize rewrite rules from the CoreIR intermediate representation to different CGRA processing elements and, second, from the WebAssembly intermediate representation to RISC-V with extension.

We implement the architectures in the Magma hardware description language [1], [55]. We chose Magma as it has first class support for formal analysis through its associated “hwtypes” library [37], whose semantics match those of the SMT-LIB theory of bitvectors. We construct an SMT formula for the architecture by tracing the inputs of the circuit and the outputs of the architectural state to the outputs of the circuit and the inputs of its architectural state. While Magma is convenient, it is not essential; any HDL could be used to generate a formal model. We specify IRs directly in SMT using pysmt [27] and use Boolector [43] as the SMT solver. Additionally, we implement minimal compilers which apply the synthesized rules in order to compare to existing hand-coded tools. Details of our full experimental set up and more results can be found in the appendix.

A. *Rewrite Rules for CGRAs*

Our first case study targets CGRAs, style of spatial architecture similar to FPGAs which have been of increasing interest to both academia and industry. CGRAs differ from FPGAs by employing larger processing elements (PEs) instead of lookup tables (LUTs). Further, CGRAs typically have more restricted word-level routing networks rather than bit-level routing networks [39]. We evaluate our ability to synthesize rewrite rules for such architectures by synthesizing rewrite rules from CoreIR to four different PEs. We chose CoreIR as a source IR as it is formally specified [16], [40].

1) CGRA Processing Element Implementation: We use four versions (PE-A, PE-B, PE-C, PE-F) of an internally developed 16-bit processing element. PE-A contains a two-input ALU that can perform bit-wise operations, comparisons, shifts, addition, and multiplication, along with a lookup table for Boolean operations. Each ALU input can be driven by an external signal or a local immediate constant. PE-F adds 16-bit floating point (bfloat16) addition and multiplication to PE-A. We then extend PE-A with operations commonly occurring in image processing applications. PE-B extends PE-A with absolute difference (`|x-y|`), and PE-C extends PE-B with fused multiply-add with an immediate constant (`x*const + y`). Generating such a collection of similar architectures is a common practice when doing design space exploration. Our synthesis method, combined with a tool such as VTR [42] to perform place and route, could enable a designer to evaluate a large design space on real benchmarks.

2) Rewrite Rule Synthesis: We evaluate our ability to synthesize rewrite rules for CoreIR’s 16-bit integer instructions (`i16`), Boolean instructions (`i1`), and floating point instructions using Bfloat16 [17] (bfloat16).

The times to derive these rewrite rules are shown in Figure 1. Note that while most CoreIR operations can be mapped to the base PE, some can only be mapped to one or more of the variants. Each rule for the integer PEs can be found within 1.1 seconds. Additionally, the floating-point instructions can be found for PE-F within 1.2 seconds.

In Table II, we show the total time in seconds spent synthesizing rewrite rules (a SAT result) or proving that no

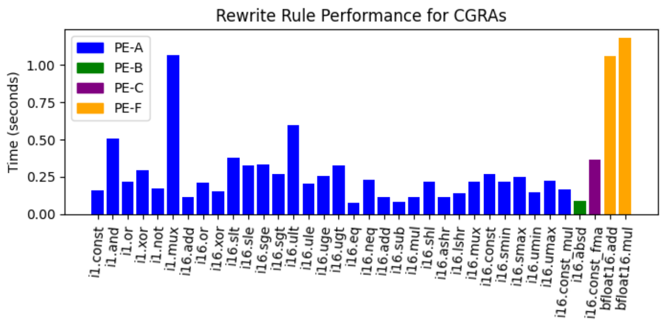

Fig. 1: The median time over 10 runs needed to derive a rewrite rule for various CoreIR operations to different PE architectures.

	PE-A	PE-B	PE-C	PE-F
UNSAT (s)	0.81	0.74	0.34	118.09
SAT (s)	8.63	10.15	11.06	91.49
Total (s)	**9.44**	**10.88**	**11.40**	**209.58**

TABLE II: Total time generating SAT results and UNSAT results, for each PE design.

rewrite exists (an UNSAT result, potentially due to the lack of a matching abstraction) for each PE design. Targeting the integer PEs is extremely fast, taking less than 12 seconds per design to generate a full set of rewrite rules. The process is "slow" for PE-F requiring about 3.5 minutes. However, this time is trivial compared to the time it would take to manually write these rules.

B. *Rewrite Rules for RISC-V*

Our second case studies shows how our technology can be used to synthesize rewrite rules from WebAssembly targeting RISC-V processors. WebAssembly is an intermediate representation designed to be a target for web applications. The IR itself has formally-defined semantics for each operation, making it suitable for our method.

We extract the post-instruction-fetch portion of the processor in order to give it the appearance of having an instruction input. Further, we replace the register file with the simplified model described in Section III-E. These transformations require only a handful of lines of boilerplate python for each architecture. Additionally, we construct specifications of instruction formats as ADTs and provide any necessary annotations for the register file (i.e., which registers have special semantics, like register 0 in RISC-V).

1) RISC-V Implementation: In addition to implementing a processor for the base RV32I ISA, we implement processors for the RV32IM and RV32IF standards. The "M" extension adds instructions for multiplication, division, and remainder. The "F" extension adds support for floating point operations. Full details can be found in the RISC-V manual [2]. In addition to these standard extensions, we define our own extension RV32X, which adds common bit-counting operations, which are defined in WebAssembly. Specifically: count-leading-zeros

Instruction	RV32I	RV32IM	RV32IX	RV32IF
`i20.const`	0.3	10.4	1.8	4.2
`i32.le_s`	2.2	27.3	3.7	80.1
`i32.ge_s`	1.6	30.8	4.5	71.7
`i32.le_u`	1.6	25.7	4.7	75.1
`i32.ge_u`	2.4	18.1	2.2	51.2
`i32.eq`	2.1	23.5	3.3	22.3
`i32.ne`	2.2	6.4	1.2	9.9

TABLE III: Median SMT performance in seconds for synthesizing two sequential instructions for `i20.const` and comparison instructions.

(`i32.clz`), count-trailing-zeros (`i32.ctz`), and population count (`i32.popcnt`).

2) Rewrite Rule Synthesis: We evaluate our ability to synthesize rewrite rules for WebAssembly's 32-bit integer instructions (`i32`) and a subset of floating point instructions (`float`). The integer instructions also include pop-count, count-leading-zeros, and count-trailing-zeros.

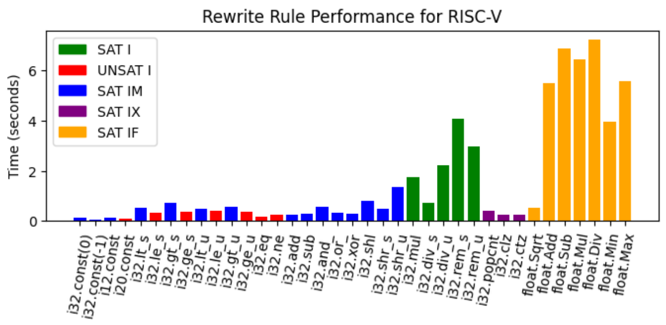

Fig. 2: Time needed to synthesize a single RISC-V instruction for each RISC-V Architecture. SAT means a rewrite rule was discovered. UNSAT means there is provably no single instruction rewrite that is possible. Reported times are the median result over 10 runs.

In Figure 2, either the time to synthesize a rewrite rule (SAT) or the time to prove that a rewrite rule does not exist (UNSAT) is shown for each IR instruction. Synthesis for RV32I succeeds in finding all instructions executable as a single instruction on the target architecture. For the integer processors, all rules are discovered within 4.1 seconds, with most only taking a few hundred milliseconds. Proving that rewrite rules do not exist is also possible within 4.1 seconds. For RV32IF, all the rules are found within 22 seconds, with most taking less than 8 seconds. Proving that particular rules like `i32.rem_s` are not possible takes up to 38 seconds. RV32IF contains many floating point instances, each requiring an expensive new universally quantified variable (explained in Section III-E). This can mostly explain the higher time compared to the other architectures.

Some comparison instructions are impossible to implement in a single instruction (a fact verified by our method), so we searched for sequences of two instructions, by composing two architectures as described in Section III-B. The times to find rewrites for these comparison operations for each of the RISC-V architectures are shown in Table III. We are able synthesize these rules for RV32I and RV32IX in a few seconds. For RV32IM and RV32IF, which are significantly more complex circuits, synthesis times are under 31 and 81 seconds, respectively. We note that verifying a rewrite rule can be done nearly instantaneously (well under a second for any rule we discovered). Therefore, given the knowledge that RV32I is a subset of RV32IF, one could simply verify that rules generated for RV32I work for RV32IF in order to avoid the longer synthesis times which arise from the complexity of floating point.

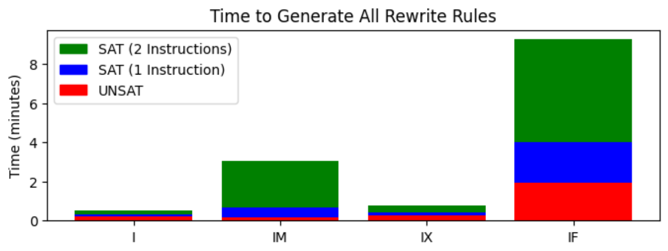

Fig. 3: Total time to generate single instruction SAT results, 2 instruction SAT results, and UNSAT results of 37 rewrite rules for each RISC-V architecture.

Similar to Table II, in Figure 3 we show the time spent synthesizing rewrite rules or proving no rewrite rule exists for each RISC-V architecture. This includes the time for proving that the instructions in Table III cannot be accomplished in one instruction, and the time for synthesizing each two-instruction rule. Results from targeting RV32I and RV32IX are fast, each taking less than a minute. Results targeting RV32IM and RV32IF are slower at around 3 minutes and 9 minutes, respectively, but this is still significantly faster than manually writing these rules.

V. Related Work

In recent years, many new techniques and tools have been developed for synthesis based on SAT and SMT solving [30], [31], [34], [50], [52], [51], [53]s. In the Sketch language, for example, a programmer provides a specification and a partial program with "holes" [52], [51]. Sketch attempts to fill these holes so that the complete program matches the specification. However, due to the nuances of targeting RTL, we found that a direct encoding into SMT formulas was more flexible and convenient than using an existing program synthesis system. One promising approach is Syntax-guided synthesis (SyGuS) [5], [6], [48], in which a program must be synthesized within a given grammar to meet a given specification (the grammar and specification are given using a variant of the SMT-LIB language [8]). Exploring possible uses of SyGuS in this context is an interesting avenue for future work.

Perhaps more relevant is the work of Dias and Ramsey [18], [19], [49], who, in their 2006 work, propose a system to synthesize rewrite rules using an ISA specification where each instruction is specified as a distinct formula. They use a pattern-matched syntax tree to synthesize these rules. In contrast, we use SMT to find all equivalences. Further, we use the RTL directly rather than a manually specified enumeration of instructions. This distinction is especially important during design space exploration, when automating as much as possible is crucial.

More recently, Buchwald, Fried, and Hack proposed a system which, like the work of Dias and Ramsey, synthesizes rewrite rules using an enumeration of an ISA's instructions [12]. However, instead of using pattern matching they leverage SMT to find rewrite rules for integer instructions. They notably lack support for floating-point, which we can handle efficiently. One interesting contribution of their work is the ability to synthesize control flow instructions by modeling them as a set of Boolean functions which indicate which branch target was taken. Applying a similar method in our approach is an interesting avenue for future work.

VI. Discussion and Future Work

Our technique for rewrite rule synthesis is a step towards automatically synthesizing a complete code generator from an RTL description of the target architecture. Future work includes two directions: synthesizing more kinds of rewrite rules, and targeting more expressive RTL. Pipelined architectures could leverage unpipelining [38] or unrolling [11] (with side conditions to ensure progress) to generate a model with the desired properties. Alternatively, if the RTL is derived from a high-level language, we could capture the synthesized design before micro-architectural details are added.

Architects often explore many alternatives when designing new hardware. This is often done incrementally. They propose a design change, implement it, then reevaluate the efficiency. A major impediment to design space exploration is implementing the software changes needed to compile the application to the new accelerator. The work in this paper enables automatically deriving part of the code generator and is one step towards the goal of eventually building a complete system for rapid and automated design space exploration.

References

[1] Magma. https://github.com/phanrahan/magma.

[2] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2. RISC-V Foundation, 2017.

[3] W. Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundation of Mathematics*, 1954.

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

[5] Rajeev Alur, Rastislav Bodik, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. To Appear in Marktoberdrof NATO proceedings, 2014. http://sygus.seas.upenn.edu/files/sygus_extended.pdf, retrieved 2015-02-06.

[6] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.

[7] 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.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[9] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018.

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

[11] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.

[12] 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.

[13] Jerry R Burch and David L Dill. Automatic verification of pipelined microprocessor control. In *International Conference on Computer Aided Verification*, pages 68–80. Springer, 1994.

[14] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[15] Chih-Hong Cheng, Natarajan Shankar, Harald Ruess, and Saddek Bensalem. EFSMT: A logical framework for cyber-physical systems. *CoRR*, abs/1306.3456, 2013.

[16] Ross Daly and Lenny Truong. Invoking and linking generators from multiple hardware languages using coreir. In *Proceedings of the 1st Workshop on Open-Source EDA Technology*, 2018.

[17] Jeffrey Dean, Greg S Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, et al. Large scale distributed deep networks. 2012.

[18] Joao Dias and Norman Ramsey. Converting intermediate code to assembly code using declarative machine descriptions. In *International Conference on Compiler Construction*, pages 217–231. Springer, 2006.

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

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

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

[22] Martin Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*.

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

[24] 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.

[25] Mahadevan Ganapathi. *Retargetable Code Generation and Optimization Using Attribute Grammars*. PhD thesis, 1980. AAI8107834.

[26] 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.

[27] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT workshop*, 2015.

[28] Philip B Gibbons and Steven S Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 11–16, 1986.

[29] 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.

[30] 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.

[31] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017.

[32] Christopher G Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

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

[34] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224, 2010.

[35] Ron Kimmel. Demosaicing: image reconstruction from color ccd samples. *IEEE Transactions on image processing*, 8(9):1221–1228, 1999.

[36] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[37] Caleb Donovick Leonard Truong. hwtypes. https://github.com/leonardt/hwtypes.

[38] Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Proceedings of the 33rd annual Design Automation Conference*, pages 558–563, 1996.

[39] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.

[40] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G Daly, Dillon Huff, and Pat Hanrahan. Cosa: Integrated verification for agile hardware design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.

[41] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Clark Barrett, Mark Horowitz, Pat Hanrahan, and Priyanka Raina. Automated design space exploration of cgra processing element architectures using frequent subgraph analysis. *arXiv preprint arXiv:2104.14155*, 2021.

[42] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai Min Wang, Mohamed ElDafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. Vtr 8: High performance cad and customizable fpga architecture modelling. *ACM Trans. Reconfigurable Technol. Syst.*, 2020.

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

[44] Mark Nixon and Alberto Aguado. *Feature extraction and image processing for computer vision*. Academic press, 2019.

[45] 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.

[46] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.

[47] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[48] Mukund Raghothaman and Abhishek Udupa. Language to Specify Syntax-Guided Synthesis Problems. *CoRR*, abs/1405.5590, 2014.

[49] Norman Ramsey and Joao Dias. Resourceable, retargetable, modular instruction selection using a machine-independent, type-based tiling of low-level intermediate code. *ACM SIGPLAN Notices*, 46(1):575–586, 2011.

[50] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. *SIGPLAN Not.*, 48(4):305–316, March 2013.

[51] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, 2013.

[52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

[53] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):497–518, 2013.

[54] Cesare Tinelli and Calogero G. Zarba. Combining decision procedures for sorted theories. In Jóse Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence*, pages 641–653, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[55] Lenny Truong and Pat Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA*, volume 136 of *LIPIcs*, pages 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[56] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.

[57] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383, 2020.

[58] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312, 2011.

Appendix

In addition to showcasing the efficiency of generating instruction rewrite rules, we wrote two compilers, one targeting the CGRA PEs, and one targeting the RISC-V processors, that showcase the synthesized rewrite rules can be used in real application compilation. The CGRA rewrite rule synthesis and compiler are actively being used in production in our lab's efforts to design and run applications on our CGRA [7].

1) CGRA Compilation Results: We apply our synthesized rewrite rules to a number of image processing applications written in the domain-specific language Halide [47]. Halide is generally amenable to hardware acceleration [46], [57], making it a suitable source language to target our PE designs. The standard Halide compiler first lowers the program to its internal IR consisting of multiple computational kernels and structured for-loops [47]. Each kernel is further lowered to a dependency graph of CoreIR instructions. Our instruction selector then applies the synthesized rewrite rules to transform each kernel into a graph of PE instructions.

We selected four typical image processing applications: (1) Gaussian blur, an algorithm for blurring an image using convolution with a Gaussian kernel [44]; (2) A bfloat16 version of Gaussian blur; (3) Harris corner detection, which finds sharp corners of objects in images [32]; and (4) A complete camera pipeline, which is representative of end-to-end processing of raw sensor data to a final image [35]. The camera pipeline includes kernels for hot pixel suppression, demosaicing, and color correction. These applications have 3, 3, 10, and 14 distinct kernels, respectively, and the number of operations within a kernel range from just a single operation to almost 200.

We compare our synthesized rewrite rules to an existing hand-coded set for PE-F. Table IV shows the instruction counts for each application with each set of rewrite rules. The code sizes for the synthesized rewrite rules are the same or better than the sizes of those from the hand-coded rules. For Harris, which contains the `i16.umin` and `i16.smin` operations, the hand-coded result uses 2 instructions[6] (`i16.lte` and `i16.mux`), but since we synthesize rewrite rules directly for `i16.umin` and `i16.smin`, the instruction selector produces more efficient code. Additionally, our instruction selector works for the new PE variants automatically. It uses the `i16.absd` instruction in both PE-B and PE-C, reducing the total instructions for Camera. Similarly, it leverages the `i16.const-fma` instruction in PE-C, greatly reducing the total instructions for Gaussian and slightly reducing them for Camera. This example demonstrates that it is easy to extend PEs with new instructions and automatically update the set of valid rewrite rules.

2) RISC-V Compilation Results: We also show that we can compile branch-free C programs using our synthesized rewrite rules. This approach has been used to evaluate other code generators [50], [30]. Specifically, we compile 25 Hacker's Delight [56] programs. We use C implementations from Gulwani et al. [30].

	Hand-Coded	Synthesized			
Application	PE-F	PE-F	PE-A	PE-B	PE-C
Gaussian i16	20	20	20	20	12
Gaussian bfloat16	20	20	N/A	N/A	N/A
Harris	116	109	109	108	108
Camera	343	338	338	309	308

TABLE IV: The number of PE instructions required for four Halide applications: Camera, Gaussian integer, Gaussian bfloat16, and Harris. The applications are compiled using both the hand-coded rewrite rules and the synthesized ones.

Benchmark	Synthesized	gcc -O0	gcc -O1
P1	3	16	3
P2	3	16	3
P3	3	16	3
P4	3	16	3
P5	3	16	3
P6	3	16	3
P7	5	19	4
P8	5	19	4
P9	4	20	4
P10	4	24	5
P11	4	22	4
P12	5	23	5
P13	5	22	4
P14	5	25	5
P15	5	25	5
P16	10	29	6
P17	6	23	5
P18	4*	36	7
P19	6	35	7
P20	9	35	8
P21	25	50	13
P22	26	39	11
P23	32	50	15
P24	18	50	12
P25	27	72	19

TABLE V: Number of RISC-V RV32IM instructions on 25 Hacker's Delight programs (P1-P25). We show our system versus `gcc` with two levels of optimization. *The compilation of P18 to WebAssembly generated a `i32.popcnt` and hence could only be compiled to RV32IX.

We compile C to stack-machine WebAssembly byte code using Emscripten [58] (using `emcc -Os`). We then transform the resulting code into a basic block by abstract interpretation on a virtual stack [22], implemented with a modified WebAssembly interpreter.

We apply type legalization [36] to decompose `i32` constants into `i12` and `i20` constants. These bit-widths are chosen as they are the bit-width of immediate fields in the RISC-V ISA. Instruction selection is then applied using the synthesized rewrite rules. Next, we perform basic instruction scheduling and register allocation, and finally we assemble the instructions into RISC-V byte code [28], [14].

We compare the code we generate to that produced

[6]We are unsure at this point whether this is a result of the architecture being updated without a corresponding update to the hand-coded rewrite rules or whether this rule was just overlooked by the original author of the tool. In any event this sort of mistake motivates this paper.

by `gcc` (`riscv64-unknown-elf-gcc -march=rv32g -mabi=ilp32`). The `gcc -O0` option uses the stack to store intermediates so our code size is better, while `gcc -O1` uses multiple basic blocks to decrease code size, which we do not support.

Table V shows a comparison of the number of instructions generated from our compiler versus a RISC-V `gcc` compiler for each Hacker's Delight program. For P21-P25, `gcc -O1` generates small code size by using branching code, an optimization we do not implement. P18 uses a `i32.popcnt` in the generated WebAssembly. When targeting RV32IX we can leverage the custom instructions to compile program P18 using only 4 instructions.