ROVER: RTL Optimization via Verified E-Graph Rewriting

Samuel Coward, Theo Drane, and George A. Constantinides, *Senior Member, IEEE,*

Abstract—Manual RTL design and optimization remains prevalent across the semiconductor industry because commercial logic and high-level synthesis tools are unable to match human designs. Our experience in industrial datapath design demonstrates that manual optimization can typically be decomposed into a sequence of local equivalence preserving transformations. By formulating datapath optimization as a graph rewriting problem we automate design space exploration in a tool we call ROVER.

We develop a set of mixed precision RTL rewrite rules inspired by designers at Intel and an accompanying automated validation framework. A particular challenge in datapath design is to determine a productive order in which to apply transformations as this can be design dependent. ROVER resolves this problem by building upon the e-graph data structure, which compactly represents a design space of equivalent implementations. By applying rewrites to this data structure, ROVER generates a set of efficient and functionally equivalent design options. From the ROVER generated e-graph we select an efficient implementation. To accurately model the circuit area we develop a theoretical cost metric and then an integer linear programming model to extract the optimal implementation. To build trust in the generated design ROVER also produces a back-end verification certificate that can be checked using industrial tools.

We apply ROVER to both Intel-provided and open-source benchmarks, and see up to a 63% reduction in circuit area. ROVER is also able to generate a customized library of distinct implementations from a given parameterizable RTL design, improving circuit area across the range of possible instantiations.

Index Terms—hardware optimization, design automation, datapath design, computer arithmetic

I. Introduction

IN recent years many new domain specific languages and tools have allowed hardware engineers to write designs at different levels of abstraction [1], [2]. Even with these developments, Register Transfer Level (RTL) design using hardware description languages such as Verilog still dominates industry and much of academia. Despite reaching maturity, logic and high-level synthesis tools are limited in their design space exploration and are unable to match skilled engineers. The hardware design space is large and mostly unexplored due to strict correctness requirements and slow debug time frames. The numerous optimization objectives and constraints present a challenge for both humans and automated systems. Automatic datapath synthesis research has focused on heuristic search and statistical methods [3]–[6] or deployed machine learning [7]. Automatic datapath synthesis can expand design

S.Coward and T.Drane are with the Intel Numerical Hardware Group. E-mail: samuel.coward@intel.com

G.A.Constantinides and S.Coward are with Imperial College London, Department of Electronic and Electrical Engineering, London, UK
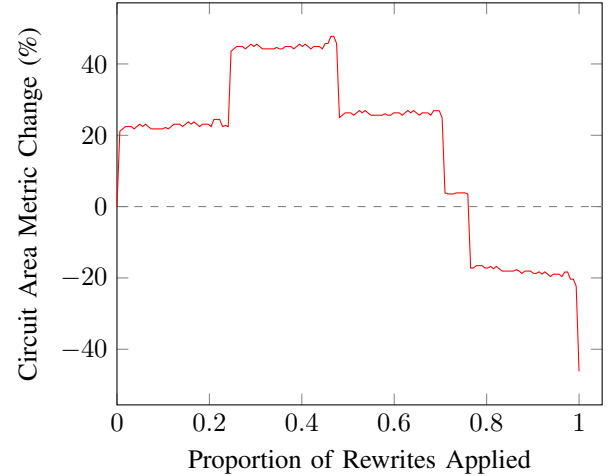
Fig. 1. Progression of design cost throughout RTL rewriting for the Weight Calculation benchmark (described in Section VII). We plot the percentage change in the circuit area metric compared to the original design at every point in the rewrite chain. The area metric may converge non-monotonically.

space exploration resulting in better quality circuit designs. It may also improve productivity reducing the engineering effort required to produce an optimized implementation.

Inspired by traditional compiler optimization techniques and previous work on RTL optimization [3], [8], we observe that manual datapath optimization at RTL can be described in terms of local equivalence-preserving transformations. Skilled engineers learn such transformations through experience and discover patterns or sequences of valuable transformations. Often these optimizations can be generalized, facilitating their application more widely. Automating transformation-driven hardware optimization is complex since it is often necessary to apply several "bad" transformations before an ultimately beneficial transformation can be applied. Figure 1 illustrates an example where it is necessary to initially apply transformations that increase circuit area cost via operator duplication or replacement, but eventually lead to subsequent area saving transformations such as arithmetic simplification or clustering, providing a net area reduction. This is a challenge faced by traditional rewriting techniques [9].

In order to meet the automation objective, we leverage recent advances in e(quivalence)-graph rewriting and equality saturation, bringing them to the RTL optimization problem. By representing combinational RTL as a dataflow graph we can exploit properties of e-graphs that make them a promising technology for hardware design. Firstly, in e-graph rewriting the order in which transformations are applied is unimpor-
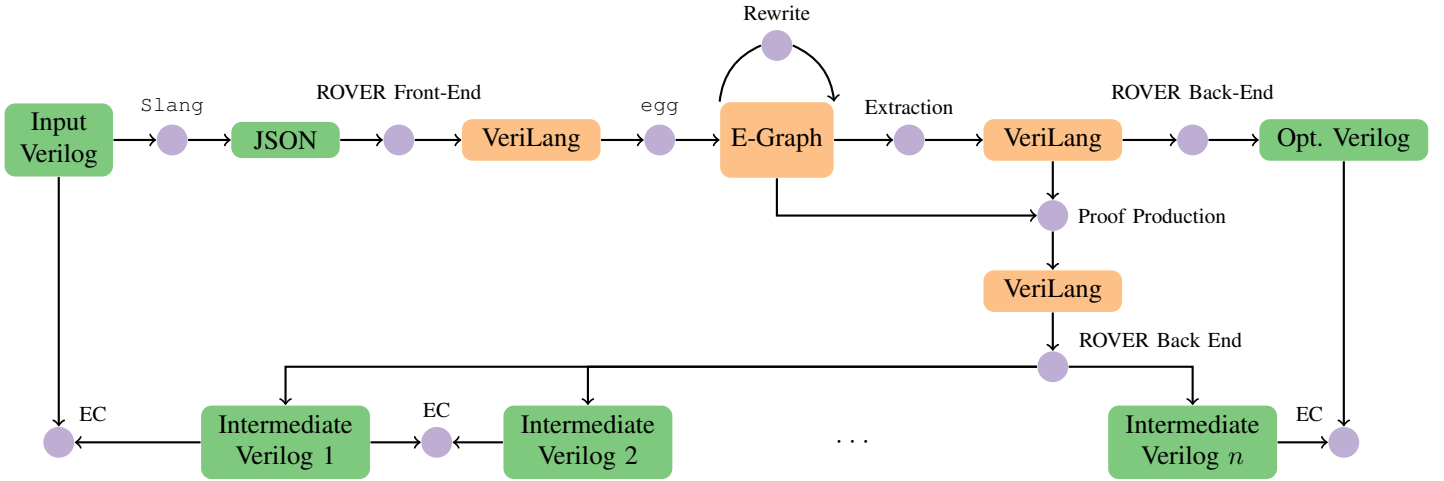
Fig. 2. Flow diagram describing the operation of ROVER. The intermediate RTL designs are formally verified to be functionally equivalent using a commercial equivalence checker (EC) forming a chain of reasoning. The orange boxes denote the novel contributions.

tant, allowing the e-graph to capture early transformations that initially degrade the design but potentially enable later beneficial optimizations. Secondly, e-graphs are designed to explore equivalent implementations, and RTL optimization typically maintains functionality producing bit-identical implementations. Lastly, the e-graph maintains the complete history of all designs it has explored, which allows us to decompose formal verification into a sequence of equivalence checks.

In this paper, we address the following problem. Given an RTL implementation R, we seek a functionally equivalent implementation R' that minimizes some cost, typically area or delay. Two implementations R and R' are equivalent, $R \cong R'$, iff for all inputs they produce identical outputs. The resulting optimized implementation R' is passed to an industrial logic synthesis tool, producing a netlist from which we can extract relevant circuit quality metrics. ROVER's optimization objective is to generate RTL that the logic synthesis tool can synthesize into the most efficient circuit representation. This means that ROVER must capture and model the downstream logic synthesis capabilities. Figure 2 provides an overview of the tool flow.

The primary contributions of this work are:

- application of e-graph rewriting to RTL datapath optimization,
- a multi-bitwidth and multi-signage rewrite set that enables datapath design space exploration capturing the connection between optimal architecture selection and bitwidth,
- an automated method to generate necessary and sufficient conditions for RTL rewrites using an equivalence checker,
- a robust method to verify the correctness of the generated RTL based on problem decomposition.

An initial application of e-graphs to general datapath optimization was presented at the 29th Symposium on Computer Arithmetic [10] as a preliminary version of this work. Here we extend the conference paper by supporting signed arithmetic, providing semantics for ROVER's intermediate language, and introducing novel methods for rewrite condition generation

and formal verification not present in the conference version.

In the next section we provide the necessary background on datapath optimization and e-graphs. In Section III we describe the intermediate language and supported subset of Verilog. Then we describe the rewrites that encode the optimizations and allow the e-graph to grow in Section IV. In Section V we describe how the optimal design is extracted from the generated e-graph. The verification methodology is described in Section VI. In the final two sections we present results.

II. BACKGROUND

A. Datapath Synthesis

Datapath synthesis is the process of generating gate level netlists from higher-level arithmetic circuit designs expressed in RTL. Zimmermann decomposes this process into three steps: RTL extraction of arithmetic operations, followed by high-level arithmetic optimizations, and finally netlist generation [11]. Such datapath optimization engines form a core component of all logic and high-level synthesis tools and are essential for generating state-of-the-art circuit designs.

Logic synthesis tools implement a range of hardware-specific optimizations, detecting opportunities to merge particular operator sequences and exploit redundant number representations [11]. Synopsys Design Compiler provides datapath coding guidelines, which describe how designers can best exploit the synthesis tool's capabilities [12]. Of particular relevance is the front-end logic synthesis pass that performs datapath extraction, which clusters operators into datapath blocks [11]. Extracting larger clusters enables more effective downstream optimization. Datapath clustering can be prevented by datapath leakage in a design, where a designer, possibly intentionally, truncates an arithmetic operation. A key objective of ROVER is to enable logic synthesis datapath extraction to form larger datapath blocks, which in turn results in more efficient circuit implementations.

In this work we use a rewrite driven approach to the datapath optimization problem. The most relevant prior academic work is from Verma, Brisk and Ienne [3], who automatically apply

dataflow transformations to combinational circuit designs. This work was inspired by the observation that ASIC logic synthesis tools could effectively deploy carry-save representation when presented with consecutive arithmetic operations, generating optimized netlists. However, when given arithmetic blocks separated by additional logical operations, the tool was not able to move the logical operations to facilitate optimal clustering. To address this issue, the authors designed specific logic arithmetic interchange rewrites that produced circuit designs, which when passed to logic synthesis could maximally cluster arithmetic operators together. By leveraging and extending these rewrites in the e-graph optimization framework, we can reproduce and extend results from this paper [9]. In addition to the work of Verma, Brisk and Ienne, a general purpose and verified RTL rewriting framework has been developed by Carl Seger and collaborators [13], [14]. The Voss II framework provides a design visualization environment and proposes a more interactive design space exploration approach, with little emphasis on automation. Datapath rewriting has also been applied to specific design challenges such as the design of large bitwidth multipliers [15]. In datapath verification, rewriting has proven to be an invaluable technique [16]–[18].

Although we target general datapath RTL optimization, there are several problem domains that have received particular attention and can be captured in the framework we present. One such instance is the multiple constant multiplication (MCM) problem [19]–[21], where the design problem is as follows: given a set of integer coefficients $\{a_1, ..., a_n\}$, find an optimal circuit producing all the outputs $a_i \times x$ for a variable input x. The challenge presents many non-obvious operator sharing opportunities and is beyond the reach of existing logic synthesis tools. These problems are usually solved by hand or with bespoke tools, which represent constants using a fixed representation [20] such as Canonical Signed Digit (CSD) [22]. Alternative approaches deploy adder graph algorithms [19] or have encoded the problem as an integer linear programming problem [23]–[25] or as Boolean satisfiability problem [26]. Owing to the generality of the e-graph rewriting framework, such MCM optimizations are another class of methods that are automatically subsumed within our ROVER framework as a special case. Note that bespoke MCM tools will outperform ROVER on complex MCM problems, as we shall see in Section VII.

B. E-Graphs

An e-graph is a data structure developed in the theorem proving community [27]. E-graphs provide a compact representation of equivalence classes (e-classes) of expressions. An e-graph represents a set of equivalent expressions, where nodes represent variables, constants or functions clustered together in e-classes. Edges represent operator inputs and connect nodes to e-classes, as shown in Figure 3. In this way, a small number of nodes can represent exponentially many more expressions. Figure 3 contains a series of e-graphs, demonstrating how multiple equivalent expressions can be represented.

E-graphs are grown using a technique called equality saturation [28]–[30], where rewrites that define equivalent expressions are applied to the e-graph. E-graphs are based on the

(a) Initial e-graph contains $(2 \times x) >> 1$

(b) Apply $x \times 2 \to x << 1$
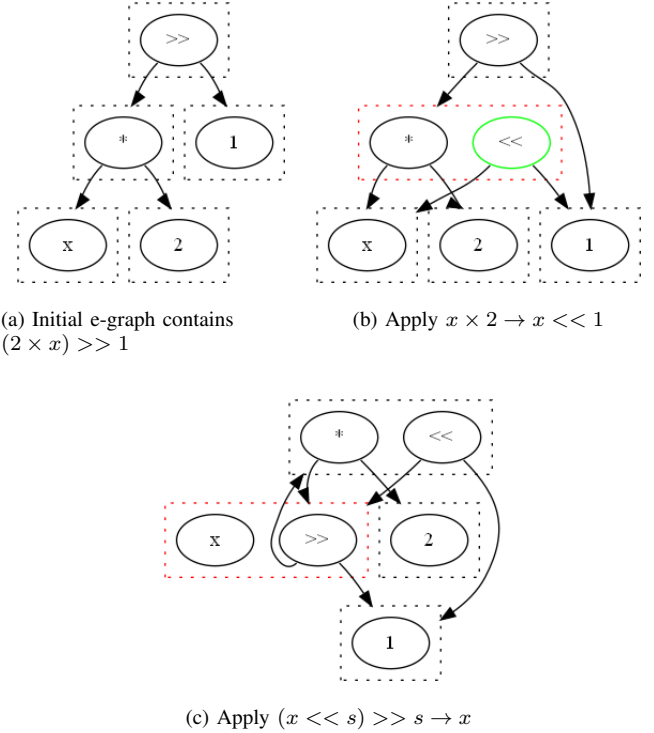
(c) Apply $(x << s) >> s \to x$

Fig. 3. E-graph rewriting for standard integer arithmetic. Dashed boxes represent e-classes of equivalent expressions. Green nodes represent newly added nodes. Red dashed boxes highlight which e-class has been modified.

theory of uninterpreted functions therefore an operator, e.g. addition, only gains any meaning via rewrites defined on that operator. For example $x + x \to 2 \times x$ tells the e-graph that $x + x$ is equivalent to $2 \times x$. Given a set of rewrites, rewriting opportunities are detected via a process known as e-matching that identifies expressions in the e-graph that match the left-hand side patterns [31], [32]. A key differentiating property of e-graphs is constructive rewrite application. More formally, a rewrite is defined as a pair of expressions (lhs, rhs), such that when an expression syntactically matching the lhs is discovered in the e-graph, the rhs is added to the matched e-class in the e-graph. The left-hand side is not destroyed and remains in the e-class. This means that the e-graph grows monotonically and application of one rewrite does not remove alternative rewriting opportunities. Figure 3 shows an e-graph before (Figure 3a) and after (Figure 3b) application of a constructive rewrite that adds a new node to the e-graph. The final rewrite applied to produce Figure 3c, $(x << 1) >> 1 \to x$, adds no new nodes to the e-graph, since the right-hand side expression is already contained within the e-graph. In this case, rewriting merges two existing equivalence classes, which, in this instance, leads to a loop in the e-graph.

Traditional rewrite engines suffer from the phase-ordering problem, which describes how the order of rewrite application can impact the final outcome [33]. Consider the following two ways to rewrite the initial expression from Figure 3a.

$$\text{Order 1: } (x \times 2) >> 1 \to (x << 1) >> 1 \to x$$
$$\text{Order 2: } (x \times 2) >> 1 \to (x + x) >> 1$$

A destructive rewriting process leads to two different endpoints depending on the order of rewrite application, hence the phase-ordering problem. This problem is avoided entirely by constructive rewrite application [28]. Applying the rewrite in "Order 2", the $x \times 2$ expression is retained and can be matched again, facilitating the application of "Order 1". This will prove to be a particularly valuable property for hardware design.

A general purpose and reusable e-graph library, `egg` [28], was recently released and has fueled a new wave of e-graph research. In addition to its usability, `egg` provides innovations in e-graph performance and numerous analysis features. In this work we will exploit the ability to write conditional and dynamic rewrite rules as well as the e-class analysis framework [28]. The validity of conditional rules can be determined at runtime based on the specific values matched by the left-hand side pattern. The dynamic rewrite rules construct, at runtime, the right-hand side of a rewrite having matched a pattern. The e-class analysis feature allows users to attach additional information to an e-class, enabling program analysis techniques [34]. Since e-graphs grow monotonically, they usually reach a fixed point called saturation, where no further rewrite applications add additional information to the e-graph.

Proof production was recently added to `egg` allowing users to extract a rewrite sequence mapping one expression to an equivalent expression in the e-graph [35]. This enables translation validation techniques to be applied to e-graph applications. Translation validation is a compiler technique to verify the correctness of a compiler's output [8]. The verification problem is broken down into a sequence of sub-problems, verifying each step of the transformation. The proof production feature has been leveraged to develop an RTL verification assistant [17].

E-graphs can be found in widely used SMT solvers such as Z3 [36]. More recently, `egg` has helped to automate numerical stability improvement in the Herbie tool [37] and synthesis smaller and more efficient rewrite sets via the Ruler tool [38]. In the hardware domain, there is growing interest, with Ustun, Yu and Zhang advocating e-graph rewriting [39]. Previous datapath research has explored alternative implementations of large multipliers on FPGAs, where different levels of decomposition were efficiently explored via equality saturation [15]. ROVER tackles the more general ASIC RTL optimization problem, maximally exploiting logic synthesis capabilities.

III. INTERMEDIATE REPRESENTATION

To facilitate RTL exploration via e-graph rewriting, we have developed an intermediate language, VeriLang, along with a parser and generator for translation to and from Verilog/SystemVerilog [42]. Since e-graphs work with expressions, VeriLang is a nested S-expression language in Common Lisp [43]. A formal description is given in Grammar 1.

As an example, in VeriLang, an 8-bit unsigned addition, stored in a 9-bit result would be expressed as:

$$(+ \ 9 \ 8 \ \text{unsign} \ x \ 8 \ \text{unsign} \ y). \tag{1}$$

TABLE I
VERILANG OPERATORS INCLUDING THE ARCHITECTURE USED FOR THEORETICAL COST ASSIGNMENT. OPERATORS ABOVE THE DASHED LINE ARE THOSE THAT DIRECTLY TRANSLATE FROM VERILOG, WHILST THOSE BELOW ARE CUSTOM OPERATORS THAT ALLOW VERILANG TO EXPRESS MORE OPTIMIZATIONS.

Operator	Symbol	Arity	Architecture
Add/Sub	+/-	2	Prefix Adder (PA)
Negation	-	1	PA [40]
Multiplication	$\times$	2	Booth Radix-4 [41]
Reduce	$\&, \|, \hat{}$	1	Log Tree
Inverse Reduce	$\sim \&, \sim \|, \sim \hat{}$	1	Log Tree
Shifting	$\ll, \gg$	2	Mux Tree
Multiplexer	$\cdot ? \cdot : \cdot$	3	Mux Gates
Concat/Repl	$\{, \}$	n	Wiring
Comparison	$==, !=$ $<, \leq$ $>, \geq$	2	PA
Range Select	slice	1	Wiring
Sum	SUM	n	CSA and PA
Muxed Mult Array	MUXAR	3	Reduction and PA
Fused Mult-Add	FMA	3	Booth Radix-4

```
term     ::=  (op width [arg] ... [arg])
         |    var  |  int
arg      ::=  width signage term
width    ::=  var  |  int
signage  ::=  var  |  unsign  |  sign
```

Grammar 1. VeriLang grammar definition. The terminal variable var is a symbol drawn from a set of expression variables, and op is an operation from the supported set of VeriLang operators as described in Table I.

To provide VeriLang semantics, we first specify two functions.

$$[\![\cdot]\!] : \text{term} \rightarrow \mathbb{Z} \tag{2}$$

$$\cdot_{\cdot,\cdot} : \mathbb{Z} \times \mathbb{N} \times \{\text{unsign}, \text{sign}\} \rightarrow \mathbb{Z} \tag{3}$$

We then define the semantics in terms of integer arithmetic:

$$[\![(op \ w \ w_1 \ s_1 \ t_1 \ \ldots \ w_n \ s_n \ t_n)]\!] = \tag{4}$$

$$([\![op]\!] \ [\![t_1]\!]_{w_1,s_1} \ \cdots \ [\![t_n]\!]_{w_n,s_n})_{w,\text{unsign}} \tag{5}$$

where $[\![op]\!]$ denotes the standard interpretation of op acting on integers and for $k \in \mathbb{Z}$, $w \in \mathbb{N}$ and $s \in \{\text{unsign}, \text{sign}\}$,

$$k_{w,s} = \begin{cases} k \mod 2^w, & \text{if } s == \text{unsign} \\ 2(k \mod 2^{w-1}) - (k \mod 2^w), & \text{else.} \end{cases} \tag{6}$$

This is a valid model of bitvector arithmetic under the least positive residue definition of modulus.

$$\cdot \mod \cdot : \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{N} \tag{7}$$

Under these semantics, (1) has the following interpretation:

$$\left(+ \ ([\![x]\!] \mod 2^8) \ ([\![y]\!] \mod 2^8)\right) \mod 2^9.$$

Type annotations are essential, since Verilog is a context determined language. The signage of an operator is determined by the signage of its input operands. For this reason we do not include a signage annotation for the output of an operator in VeriLang. The bitwidth of an operator is determined by the bitwidth of the largest operand, *including the left-hand side of an assignment* [42]. Therefore we do include a bitwidth annotation for the output of an operator in VeriLang. Only the subset of VeriLang expressions comprised of concrete instances
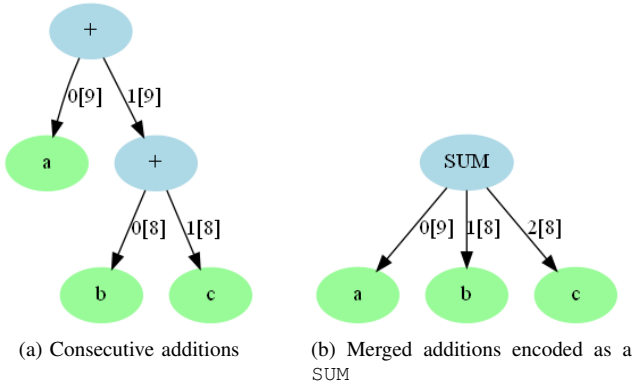
(a) Consecutive additions (b) Merged additions encoded as a SUM

Fig. 4. Edge labels show the operand's index and bitwidth in square brackets.

of the `width` and `signage` type parameters, meaning these cannot be variables, can be translated to synthesizable Verilog.

Since e-graph rewriting is based on the theory of uninterpreted functions, operators take on meaning via rewrites that define equivalent implementations. VeriLang is designed with rewrites in mind, making it simple to express conditional and dynamic rewrites with access to all the relevant parameter values. In Section IV we describe how ROVER's rewrites differentiate between type annotations and variables. Type annotations are also essential for accurate hardware costing, since an 8-bit addition should be cheaper than a 32-bit addition.

VeriLang currently supports almost all the fundamental Verilog operators, with the exception of less commonly used operators such as trigger ($->$), modulus (%) and power (**), though these could easily be added. In total we support 29 of the Verilog defined operators as shown in Table I, which omits the single gate operators that are also supported.

In addition to the Verilog operators, VeriLang supports a set of custom operators as described in Table I, which capture the optimization capabilities of modern ASIC logic synthesis tools. These additional operators greatly improve correlation between ROVER's cost model and the final circuit cost reported by commercial synthesis tools [11]. The SUM operator encodes how multiple additions can be clustered into a single carry-save adder (CSA) allowing the circuit to deploy fewer expensive carry-propagate adders. These clustering nodes are typically valuable but may not be useful if an intermediate result is required. Figure 4 shows two consecutive additions being reduced to a single SUM node. We include two further merged operators, the familiar Fused Multiply-Add (FMA), which encodes the ability to construct the circuit for $a*b+c$ using a single carry-propagate adder and the Muxed Mult Array, which encodes the synthesis optimization for $a \times b + \bar{a} \times c$ as described in [11]. The Muxed Mult Array will be discussed further in Section IV.

As shown in Figure 2, the input Verilog/System Verilog is first parsed by the open-source `slang` parser [44], generating a JSON representation. The ROVER front-end then translates this JSON representation into a VeriLang expression. From this VeriLang expression `egg` generates an initial e-graph, where each e-class contains a single node. In the initial translation phase, we construct a mapping from the original variable names to their corresponding VeriLang expressions.

By attaching the variable name to the corresponding e-classes, we retain information from the original RTL, which we can use during code generation (Section V-C) to improve readability.

IV. REWRITES

A. Specifying Rewrites

Rewrites define local equivalences between two expressions that, when chained, enable architectural exploration. Equivalence is defined as functional equivalence over `terms` in VeriLang. Namely, given `terms` t_1 and t_2, $t_1 \cong t_2$ if and only if for all possible inputs t_1 and t_2 produce identical outputs under the semantics of VeriLang. A rewrite is defined as a transformation from a `term` to a `term`. Note that rewrite pattern `terms` may contain free variable bitwidth and signage type parameters. This is analogous to using parameterizable bitwidths in Verilog as opposed to concrete integer values.

Via the e-matching process described in Section II-B, `egg` matches a `term` in the e-graph returning a `map` that is an assignment of (some of the) variables in the `term` to concrete values. A partial evaluation of a `term` with respect to a `map` produces a new `term`, $[\![\cdot]\!] : \texttt{term} \times \texttt{map} \to \texttt{term}$. As a first example, we describe the unconditional commutativity rewrite that is always valid. Later we will give an example of a conditional rewrite. Commutativity of addition is defined as:

$$\overbrace{(+\; w\; w_a\; s_a\; a\; w_b\; s_b\; b)}^{lhs} \to \overbrace{(+\; w\; w_b\; s_b\; b\; w_a\; s_a\; a)}^{rhs}.$$

If applied to an e-graph containing (1), the e-matching process would return a `map`,

$$m = \begin{cases} w & \mapsto 9 \\ w_a & \mapsto 8 \\ s_a & \mapsto \texttt{unsign} \\ w_b & \mapsto 8 \\ s_b & \mapsto \texttt{unsign} \end{cases} \tag{8}$$

Note that m is a partial function because it does not provide any assignment for variables a and b. This approach differs from other e-graph based applications, in that a single rewrite encodes a rewrite over many distinct types. Previous work encoded types in the operator name itself e.g. $+_{16}$ and $\times_{32}$ [15], but in our setting this is impractical due to the number of operators we would have to support. The partially evaluated `term`, $[\![rhs]\!]_m$ is then added to the e-graph, where

$$[\![rhs]\!]_m = \texttt{(+ 9 8 unsign b 8 unsign a)}.$$

For this simple commutativity example, the rewrite is valid anywhere that it matches. However, the set of RTL rewrites for which this statement holds is small. To enable meaningful RTL transformations, we define a set of conditionally applied rewrites specified as a triple (`cond`, `term`, `term`), where

$$\texttt{cond} : \texttt{map} \to \texttt{Bool}.$$

The condition is checked each time the left-hand side `term` of a rewrite is matched. The partially evaluated right-hand side is only added to the e-graph if the condition returns true.

```
wire   [7:0] A, B, C;
wire   [7:0] add_8bit;
wire   [8:0] add_9bit, add_right;
wire   [9:0] left1, left2, right;

assign add_8bit  = A + B; // carry-out discarded
assign left1     = add_8bit + C;

assign add_9bit  = A + B; // carry-out retained
assign left2     = add_9bit + C;

assign add_right = B + C;
assign right     = A + add_right;
```

Fig. 5. Verilog associativity rewriting example. Signals `left1` and `right` are functionally distinct, because the carry-out is discarded in computing `add_8bit`, therefore `left1`$\nrightarrow$`right`. The signals `left2` and `right` are functionally equivalently, therefore it is valid to rewrite `left2`$\rightarrow$`right`.

That is, the condition for correctness of a conditional rewrite (ϕ, lhs, rhs) is that for any map m:

$$\phi(m) \Rightarrow [\![lhs]\!]_m \cong [\![rhs]\!]_m. \tag{9}$$

In Figure 5, we provide an example, to highlight where the validity of a rewrite can depend on the context. Specifically, the associativity rewrite is valid in the case where the intermediate signal retains the carry-out of the first addition.

Conditional rewriting allows ROVER to detect all syntactic opportunities to apply a transformation and then filter out those that would be semantically invalid. Such an approach allows ROVER to capture a wide range of RTL transformations without sacrificing correctness. In Section IV-B we describe the construction of the conditions and return to this example to construct a condition for this exact associativity rewrite.

The set of rewrites described in Table II captures optimizations learnt from Intel's Numerical Hardware Group, prior work [3] and logic synthesis documentation [11], [12]. All rewrites include the type annotations described in Section III. We impose no restrictions on the bitwidth and signage parameters in the rewrites, to ensure maximum generality of the rewrites. We omit the bitwidth and signage annotations as well as the conditions in Table II to maintain readability.

ROVER combines both static rewrites, where the right-hand side is known at compile time, and dynamic rewrites, where the right-hand side is constructed at runtime. Dynamic rewrites are particularly useful for constant manipulation, building normal forms and computing sufficient bitwidths.

The first group, bitvector arithmetic identities, contains familiar arithmetic rewrites allowing ROVER to re-arrange and simplify arithmetic expressions. The second group includes transformations more commonly encountered in hardware design, simplifying logical expressions and removing redundant logic. The third class of rewrites, Arithmetic Logic Exchange, are inspired by the work of Verma *et al.* [3] and facilitate the discovery of additional arithmetic clustering opportunities. These opportunities can be missed by logic synthesis as arithmetic operations can be separated by logical operations. The Arithmetic Logic Exchange rewrites allow ROVER to move logic operations over arithmetic operations, enabling larger arithmetic clusters to form. Once clustered together,

these blocks can be effectively optimized by logic synthesis resulting in more optimal circuit designs. We extend prior work on this subject [3], generalizing and expanding the scope.

The Merging Ops rewrites detect certain operator combinations and cluster them into a single custom operator which, as described in Section III, allows ROVER to identify sub-circuits that synthesis tools will specifically optimize [12]. Both the "Merge Additions" and "FMA Merge" rewrites exploit carry-save format to construct a multi-row array which can be reduced using half- and full-adders [22]. Like the `SUM` operator, the `FMA` operator requires a single carry-propagate adder to generate the result $a \times b + c$. The "Merge Mult Array" identifies disjoint multiplier arrays that can be merged. Letting $a[i]$ represent bit i of a and $u = \lceil \log_2 r \rceil$, `MUXAR` in the table denotes the right hand side of the rewrite, where the `SUM` represents array reduction:

$$
\begin{aligned}
\mathrm{MUXAR}(b, a, c) = \\
\mathrm{SUM}((b[0]?a:c) \ll 0, \\
(b[1]?a:c) \ll 1, ..., \\
(b[r-1]?a:c) \ll (r-1)).
\end{aligned}
$$

These rewrites help ROVER to identify the best design to pass onto logic synthesis as they encode downstream logic synthesis optimizations directly in the e-graph.

The remaining class of rewrites, "Constant Expansion", explores alternative representations of constants in hardware with particular attention paid to multiplication of a variable by a constant. These rules generalize MCM optimizations and are valuable where constant manipulation can occur as a sub-problem in a larger design optimization, where a specialist MCM tool is not applicable. We shall encounter such results in Section VII, but will also encounter limitations of a rewriting approach for complex MCM problems. These rules allow ROVER to re-create and generalise results from the MCM literature described in Section II-A. As in previous `egg` implementations, constant folding is implemented as an e-class analysis [28].

B. Synthesizing Rewrite Conditions

As described above, rewrites are encoded as triples (`cond`, `term`, `term`), where the `terms` may contain variable `width` and `signage` parameters. Not all assignments to these parameters produce valid rewrites. Namely, in general, for rows in the table with † conditions, there exist mappings m such that $[\![lhs]\!]_m \ncong [\![rhs]\!]_m$. In this section we describe a solution to the following problem. Given a pair of `terms`, (lhs, rhs), construct a `cond`, ϕ, such that for all maps m,

$$\phi(m) \Leftrightarrow [\![lhs]\!]_m \cong [\![rhs]\!]_m. \tag{10}$$

The sufficiency of ϕ ($\Rightarrow$) is essential because applying a single invalid rewrite introduces a non-equivalent expression into the e-graph, meaning that no design in the e-graph can be trusted. The necessity of ϕ ($\Leftarrow$) ensures that no rewriting opportunities are missed by ROVER. In practice, constructing a ϕ satisfying (10) is challenging. To make progress, we make certain assumptions that simplify the problem, as described below.

TABLE II
ROVER'S BITWIDTH AND SIGNAGE DEPENDENT DATAPATH REWRITES. BITWIDTH AND SIGNAGE PARAMETERS ARE OMITTED HERE. THE $*$ OPERATION REPRESENTS BOTH $\{+, \times\}$. THE RULES ARE CONDITIONALLY APPLIED AS A FUNCTION OF THE BITWIDTH AND SIGNAGE INFORMATION ATTACHED TO EACH OPERAND. THE NECESSARY AND SUFFICIENT CONDITIONS ARE TOO COMPLEX (DENOTED BY †) TO DISPLAY IN COLUMN 4 FOR MOST REWRITES.

Class	Name	Left-hand Side	Right-hand Side	Condition
	Commutativity	$a * b$	$b * a$	True
	Associativity	$(a * b) * c$	$a * (b * c)$	†
	Associativity of Sub	$(a - b) - c$	$a - (b + c)$	†
	Dist Mult over Add/Sub	$a \times (b \pm c)$	$(a \times b) \pm (a \times c)$	†
	Dist Add/Sub over Mult	$(a \times b) \pm (a \times c)$	$a \times (b \pm c)$	†
	Add Zero	$a + 0$	$\texttt{slice}(a)$	†
Bitvector Arithmetic	Mul by Zero	$a \times 0$	0	†
	Mult by One	$a \times 1$	$\texttt{slice}(a)$	True
	Mult by Two	$a \times 2$	$a \ll 1$	True
	Sub to Neg	$a - b$	$a + (-b)$	True
	Sum Same	$a + a$	$2 \times a$	†
	Mult Sum Same	$(a \times b) + b$	$(a + 1) \times b$	†
	Merge Left Shift	$(a \ll b) \ll c$	$a \ll (b + c)$	†
	Merge Right Shift	$(a \gg b) \gg c$	$a \gg (b + c)$	†
	Redundant Sel	$b?a : a$	$\texttt{slice}(a)$	True
Bitvector Logic	Nested Mux Left	$a?(a?b : c) : d$	$a?b : d$	†
	Nested Mux Right	$a?b : (a?c : d)$	$a?b : d$	†
	Sel Left Shift	$e?(a \ll b) : (c \ll d)$	$(e?a : c) \ll (e?b : d)$	†
	Sel Right Shift	$e?(a \gg b) : (c \gg d)$	$(e?a : c) \gg (e?b : d)$	†
	Not over Con	$\sim \{a, b\}$	$\{(\sim a), (\sim b)\}$	†
	Left Shift Add	$(a + b) \ll c$	$(a \ll c) + (b \ll c)$	†
	Add Right Shift	$a + (b \gg c)$	$((a \ll c) + b) \gg c$	†
	Left Shift Mult	$(a \times b) \ll c$	$(a \ll c) \times b$	†
	Sel Add/Mul	$e?(a * b) : (c * d)$	$(e?a : c) * (e?b : d)$	†
Arithmetic Logic	Sel Add Zero Left	$e?(a + b) : c$	$(e?a : c) + (e?b : 0)$	†
Exchange	Sel Add Zero Right	$e?a : (b + c)$	$(e?a : b) + (e?1 : c)$	†
	Sel Mul One Left	$e?(a \times b) : c$	$(e?a : c) \times (e?b : 1)$	†
	Sel Mul One Right	$e?a : (b \times c)$	$(e?a : b) \times (e?1 : c)$	†
	Move Sel Zero	$(b?0 : a) \times c$	$a \times (b?0 : c)$	†
	Concat to Add	$\{a, b\}$	$(a \ll w_b) + b$	†
	Neg Not	$-a$	$(\sim a) + 1$	†
	Merge Additions	$a1 + (a2 + (a3 + ... + an)...)$	$\texttt{SUM}(a1, a2, ..., an)$	†
Merging Ops	Merge Mult Array	$(a \times b) + (c \times (\sim b))$	$\texttt{MUXAR}(b, a, c)$	†
	FMA Merge	$(a \times b) + c$	$\texttt{FMA}(a, b, c)$	†
Constant Expansion	Mult Constant	$c \times x$	$((2 \times (c \gg 1)) \times x) + (c[0] \times x)$	†
	One to Two Mult	$1 \times x$	$(2 \times x) - x$	†

We have developed an automated condition synthesis flow, shown in Figure 6, that makes ROVER extensible. Developers or design engineers can specify new ROVER rewrite rules as pairs of `terms` and run ROVER's condition synthesis flow to automatically generate a correct `cond`. This allows design engineers to include valuable transformations drawing from their own experience, but avoids the overhead of considering all the scenarios in which the transformation is valid or invalid. The idea is to sample the space of all signages and all small bitwidth combinations, and to build a general rule for validity consistent with the sample taken.

The automated condition synthesis flow deploys program synthesis [45], where a correct condition is learnt from data. Let lhs contain H free bitwidth parameters w_1 to w_H and G free signage parameters s_1 to s_G.

$$M = \{w_1 \mapsto \texttt{w1}, ... w_H \mapsto \texttt{wH}, s_1 \mapsto \texttt{s1}, ... s_G \mapsto \texttt{sG}$$
$$| \ \texttt{wi} \in \{1, ..., 8\} \wedge \texttt{si} \in \{\texttt{unsign}, \texttt{sign}\}\}.$$

We enumerate the entire parameter space, M, constructing VeriLang expressions $[\![lhs]\!]_m$ and $[\![rhs]\!]_m$ for all $m \in M$, and determine, for each $m \in M$, whether these representations are equivalent. ROVER converts both $[\![lhs]\!]_m$ and $[\![rhs]\!]_m$ to Verilog then deploys a commercial RTL equivalence checker

(EC). This enables the re-use of the RTL generation framework (see Section V-C) and defers Verilog semantic interpretation to the commercial tool. Each mapping corresponds to a single lemma, which the EC either proves (true) or disproves (false). These results are stored in a lookup table T such that

$$T(m) = \begin{cases} \text{true,} & \text{if } [\![lhs]\!]_m \cong [\![rhs]\!]_m \\ \text{false,} & \text{else.} \end{cases} \quad (11)$$

The lookup table T, represents the data from which ROVER learns a condition. The objective is to determine a condition, ϕ, that can be extrapolated beyond the domain M. To achieve this ROVER fits a decision tree classifier [46] to determine a predicate, ϕ, such that

$$\forall m \in M, \ \phi(m) = T(m). \quad (12)$$

ROVER uses Python's sklearn library implementation to fit a decision tree classifier. The classifier learns based on Boolean
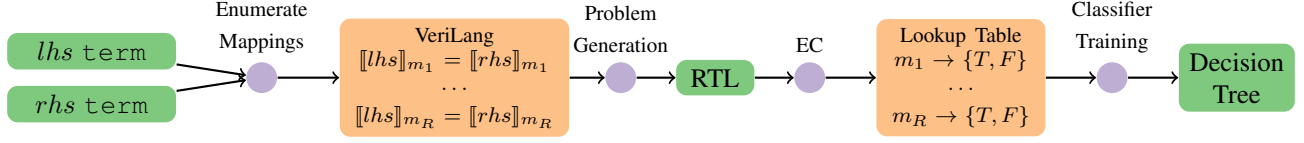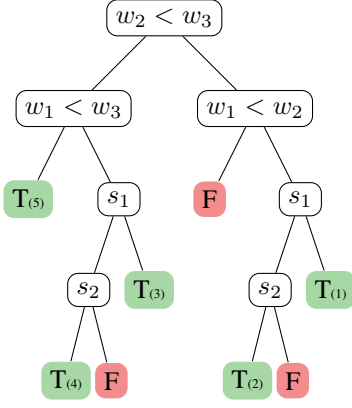
Fig. 6. Flow diagram for the automated process of synthesizing rewrite conditions. The output is a decision tree that is translated into a Boolean expression.

$$(+\ w_3\ w_2\ s_2\ (+\ w_2\ w_1\ s_1\ \mathbf{a}\ w_1\ s_1\ \mathbf{b})\ w_1\ s_1\ \mathbf{c}) \rightarrow$$
$$(+\ w_3\ w_1\ s_1\ \mathbf{a}\ w_2\ s_2\ (+\ w_2\ w_1\ s_1\ \mathbf{b}\ w_1\ s_1\ \mathbf{c}))$$

$$
\begin{aligned}
\phi\ =\ & \\
(1)\quad & (w_2 < w_3 & \wedge\quad & w_1 < w_2 & \wedge\quad & s_1) & & \vee \\
(2)\quad & (w_2 < w_3 & \wedge\quad & w_1 < w_2 & \wedge\quad & !s_1 & \wedge\quad !s_2) & \vee \\
(3)\quad & (!(w_2 < w_3) & \wedge\quad & w_1 < w_3 & \wedge\quad & s_1) & & \vee \\
(4)\quad & (!(w_2 < w_3) & \wedge\quad & w_1 < w_3 & \wedge\quad & !s_1 & \wedge\quad !s_2) & \vee \\
(5)\quad & (!(w_2 < w_3) & \wedge\quad & !(w_1 < w_3))
\end{aligned}
$$

Fig. 7. A decision tree classifier, which determines whether the restricted associativity of addition rewrite (shown above the tree) is valid (T) or invalid (F). The right/left branch is taken if the condition is true/false. The s_i nodes evaluate to true when $s_i ==$ unsign. The decision tree corresponds to the sum of product Boolean expression displayed at the bottom of the tree, where each product corresponds to a particular T leaf.

features (13)-(18).

$$
\begin{aligned}
i = 1 \dots m, \quad & s_i == \text{unsign} & (13) \\
i, j, k = 1 \dots n, i \neq j \neq k, \quad & w_i == w_j & (14) \\
& w_i < w_j & (15) \\
& w_i \pm 1 < w_j & (16) \\
& w_i + w_j < w_k & (17) \\
& w_i + 2^{w_j} < w_k & (18)
\end{aligned}
$$

These features are relevant for the operators supported in VeriLang. For example, (15) indicates whether an addition of w_i-bit integers stored in a w_j-bit signal will retain a carry-out. Similarly, (17) relates to a multiplication of a w_i-bit integer and a w_j-bit integer stored in a w_k-bit signal. Lastly, (18) relates to a w_i-bit integer left-shifted by a w_j-bit integer stored in a w_k-bit signal.

Starting from depth one, ROVER incrementally increases the maximum decision tree depth during the fitting procedure until the generated classifier satisfies (12), corresponding to

zero classification error on the training set. In Figure 7, we take a restricted associativity of addition rewrite as an example, where we force the variables a, b and c to have identical bitwidth and signage parameters. This rewrite contains $H = 3$ free bitwidth parameters and $G = 2$ free signage parameters. The procedure shown in Figure 6 generates $|M| = 8^3 \times 2^2 = 2048$ equivalence checks. The equivalence check results are used to train a decision tree classifier, which achieves perfect classification accuracy at depth four. The resulting decision tree is shown in Figure 7, where each T (F) leaf corresponds to valid (invalid) rewrite instances.

The decision tree is converted to a Boolean expression in sum of product form, yielding a ϕ that satisfies (12), where only the leaves that are classified as true are retained. The sum of product expression corresponding to the example decision tree is shown in Figure 7. The minimum depth classifier satisfying (12) corresponds to a condition with the minimal number of products. Even for a relatively simple rewrite such as the unrestricted associativity of addition, there are $H = 5$ free bitwidth parameters and $G = 4$ free signage parameters. As a result, the fitting process described above generates a depth 9 decision tree classifier.

Via the e-matching process egg searches the e-graph for expressions matching the left-hand side of a given rewrite, returning a mapping m. ROVER evaluates the synthesized cond, $\phi(m)$, to determine whether the rewrite can be applied or not. ϕ is guaranteed to be necessary and sufficient if the mapping returned by the e-matching process $m \in M$. For example, applying the rewrite described in Figure 7 to an e-graph corresponding to the Verilog shown in Figure 5, e-matching returns two maps m_1 and m_2 corresponding to the expressions for left1 and left2 respectively.

$$
m_1 = \begin{cases}
w_3 & \mapsto 9 \\
w_2 & \mapsto 8 \\
s_2 & \mapsto \text{unsign} \\
w_1 & \mapsto 8 \\
s_1 & \mapsto \text{unsign}
\end{cases}
\qquad
m_2 = \begin{cases}
w_3 & \mapsto 9 \\
w_2 & \mapsto 9 \\
s_2 & \mapsto \text{unsign} \\
w_1 & \mapsto 8 \\
s_1 & \mapsto \text{unsign}
\end{cases}
$$

Evaluating the cond, ϕ, shown in Figure 7

$$\phi(m_1) = \text{false} \qquad \phi(m_2) = \text{true}. \qquad (19)$$

This agrees with the validity statements made in Figure 5.

Since ROVER supports Verilog with signals exceeding 8-bit integers (the limit of the training data), we extrapolate by assuming that the predicate, ϕ, learnt on training data is valid for the entire domain of feasible bitwidths, which is an infinite space. Even if this assumption is incorrect, false positives, which we did not observe in practice, are detected by the back-end verification, described in Section VI, preventing ROVER from delivering functionally incorrect RTL.

V. EXTRACTION AND BACK-END

ROVER applies rewrites to the e-graph until saturation (defined in Section II-B) or a user defined iteration limit is reached. The final e-graph contains a set of valid implementations. The extraction process selects a set of e-classes to implement and within these e-classes chooses the best node to implement that particular e-class. ROVER selects the minimum area design according to a theoretical area metric.

A. Cost Model

The theoretical area metric estimates, per operator, the number of two-input gates required to build that operator, as a function of the input and output parameters. For most logical operators the cost metric is fairly simple, but for the arithmetic operators we fix a particular architecture from amongst the various possibilities. These architecture choices are described in Table I and are representative of operator architectures implemented by commercial synthesis tools [11]. When at least one operand is constant we use different constant specific costs, as logic synthesis propagates constants throughout a circuit to reduce the number of gates, e.g. constant multiplication.

Having assigned a cost to each operator, the objective is to minimize the sum of the operator costs. Note that by computing theoretical costs for the merging operators, SUM, MUXAR and FMA downstream synthesis optimizations are encoded directly in the cost model. The theoretical cost metric allows ROVER to efficiently evaluate alternative designs in the e-graph. Commercial ASIC high-level synthesis (HLS) tools use call-outs to logic synthesis engines to evaluate different circuit designs [2]. Such an approach is more computationally intensive thus limiting design space exploration. In Section VIII, we evaluate the effectiveness of the theoretical cost metric.

B. Common Sub-Expression Aware Extraction

An accurate circuit area model must correctly account for common sub-expressions. For example a circuit to generate $(a + b) \times (a + b)$ should be costed as `let c = a + b in` $c \times c$. Such a requirement makes extraction a global problem, since an optimal e-node implementation for a given e-class is no longer local, instead it may depend on implementation choices made in other e-classes. The default greedy extraction method in egg fails to account for common sub-expression re-use, therefore yielding sub-optimal solutions. The common sub-expression problem has been solved by casting extraction as an integer linear programming (ILP) problem [47].

Let $\mathcal{N}$ denote the set of all nodes, $\mathcal{C}$ denote the set of all e-classes and $E \subseteq \mathcal{N} \times \mathcal{C}$ be the set of e-graph edges. Additionally, let $\mathcal{N}_c$ be the set of nodes in a particular e-class c. For each node $n \in \mathcal{N}$, we associate some cost, $\text{cost}(n)$, based on the theoretical cost metric and a binary variable $x_n \in \{0, 1\}$, indicating whether n is implemented in the final RTL. The objective function of the ILP is described in (20). The program constraints ensure that we extract a valid circuit description. The first constraint (21) ensures that at least one node from all children e-classes of a selected node is implemented. The final constraint ensures that for all output expressions found in the set of e-classes $\mathcal{S}$, we generate a circuit producing that output.

$$\text{minimize: } \sum_{n \in \mathcal{N}} \text{cost}(n) x_n \text{ subject to:} \qquad (20)$$

$$\forall (n, c) \in E. \ x_n \leq \sum_{n' \in \mathcal{N}_c} x_{n'} \qquad (21)$$

$$\forall c \in \mathcal{S}. \ \sum_{n \in \mathcal{N}_c} x_n = 1. \qquad (22)$$

Since e-graphs may contain cycles we include additional topological sorting variables associated with each class t_c. Let N denote the number of e-classes and $\mathcal{C}(n)$ be the e-class containing node n. The constraint (23) ensures that the output expression is acyclic.

$$\forall (n, k) \in E \quad t_{\mathcal{C}(n)} - N x_n - t_k \geq 1 - N \qquad (23)$$

Selecting a node $n \in \mathcal{N}_c$ with child k, *i.e.* $x_n = 1$, the constraint simplifies to $t_c \geq t_k + 1$ to get a topologically sorted result, whereas in the case $x_n = 0$, the constraint is vacuously satisfied. To solve this ILP problem we deploy the CBC solver [48]. The ILP solution corresponds to a single VeriLang expression, that is a minimal circuit implementation according to the theoretical area metric.

C. Code Generation

Having obtained a VeriLang expression, ROVER translates this expression into System Verilog to be processed by downstream synthesis tools. The translation is implemented as an e-class analysis, as described in Section II-B. Initializing a code generation e-graph with a single VeriLang expression, the e-class analysis is constructed from the leaves upwards producing a valid System Verilog implementation. To each e-class we assign a unique signal name, its defined bitwidth and the System Verilog string that implements the particular operation in the e-class. Each e-class in the e-graph corresponds to a single line of functional System Verilog in the output. Traversing the e-graph, ROVER defines a signal at each e-class and assigns the stored expression to that signal name.

An advantage of the e-graph approach is that ROVER can maintain a mapping between user defined signal names and e-classes throughout the exploration. If such an e-class is present in the extracted implementation, ROVER overwrites the signal name of the appropriate e-class in the code generation e-graph. As a result, the generated System Verilog retains a subset of the original signal names. For example, if a user defined a signal `two_x`, assigning it to the expression $x + x$, and that was rewritten as $x \ll 1$, then the `two_x` signal would still appear in the generated output, with a different assignment.

VI. VERIFICATION

To increase trust and ensure that the input and generated circuit designs are equivalent, ROVER generates verification scripts for a commercial EC. In many cases, the EC is able to prove the functional equivalence of the input and ROVER generated RTL, without any additional guidance. However,

there are instances where the equivalence engine returns an inconclusive result [16]. Debugging inconclusive proofs can be time consuming for verification engineers. To provide a robust verification flow, ROVER uses the `egg` proof production feature [35] described in Section II-B, to decompose the verification problem into a sequence of simple sub-problems.

ROVER uses proof production to extract a sequence of intermediate VeriLang expressions, differing by a single local rewrite at each step. The sequence traces a path between the input and optimized expressions, as shown in Figure 2. Using the ROVER back-end, each intermediate VeriLang expression is converted to System Verilog. Each pair in the sequence is proven equivalent using the EC, constructing the chain of reasoning that the original and optimized implementations are equivalent. To further aide proof convergence, ROVER identifies the specific signal modified in each pair via an additional lemma. ROVER's proof sequences can contain hundreds of intermediate steps. ROVER generates both the RTL and proof scripts, providing a proof certificate to the user which can be re-run to verify the RTL.

VII. RESULTS

We used ROVER to optimize a number of industrially and academically sourced RTL benchmarks, automatically producing optimized RTL implementations. The original and optimized designs are synthesized using a commercial synthesis tool for a TSMC 5nm cell library. We also study the synthesis reports to analyze the effectiveness of ROVER's datapath clustering optimizations. Using the approach described in Section VI we verified the functional equivalence of the original and optimized architectures. We compare each pair of designs at two points along the area-delay trade-off curve using logic synthesis. Firstly, we compare at the minimal delay target at which both designs can meet timing (rounded to the nearest 10 picoseconds), corresponding to the vertical dashed line in Figure 8. The second comparison point, is at the minimum area that both designs can fit within (yielding different performance levels), corresponding to the horizontal dashed line in Figure 8.

The results are summarized in Table III. We will primarily focus on the area and delay impact since the cell count and power measurements are proportional to the area in this work. In Figure 8 we plot the area-delay profile comparing the original and ROVER optimized designs across the delay spectrum. We separate the results into two contributions. Firstly, we show how ROVER can optimize general RTL benchmarks. Then we demonstrate how ROVER can optimize different instances of parameterizable RTL, generating a suite of tailored implementations.

A. Exploiting Datapath Optimizations

The first set of benchmarks in Table III are Intel RTL designs. The first benchmark is a kernel from the Intel media module. The initial design was optimized by hand by a hardware design expert. ROVER is able to automatically optimize the design and achieve comparable results to the manually optimized RTL, discovering the opportunity to merge two multiplication arrays into a single array using the "Merge Mult
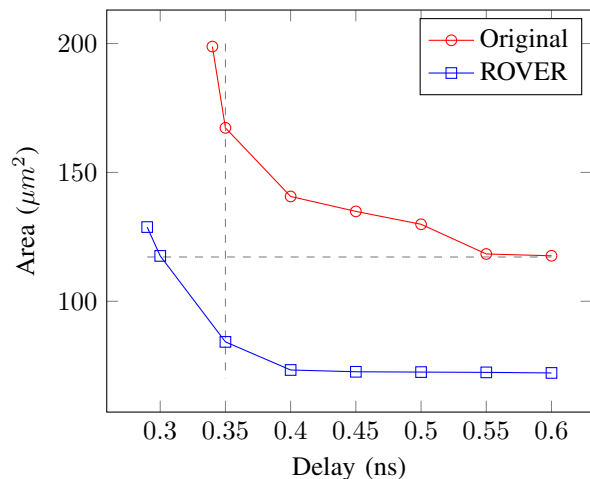

Fig. 8. Area-delay profiles for the original and ROVER optimized Media Kernel designs. The dashed grey lines indicate the minimum area and delay comparison points used in Table III.

Array" rewrite. Studying the reports generated by the synthesis tool, we can identify the source of the area reduction. The original design produces four datapath clusters, corresponding to four carry-propagate adders in the synthesized netlist. By contrast, the ROVER optimized design produces two datapath clusters, halving the number of carry-propagate adders in the generated netlist. These improvements translate to a 14.7% reduction in minimum achievable delay within a circuit area 35.4% smaller. In the logic synthesis engine, further arithmetic clustering is prevented because the tool detects datapath leakage (as described in Section II-A) due to supposed truncation in the following System Verilog.

$$a[8:0]= 9'd256 - \{1'b0,b[7:0]\};$$

This analysis, however, is flawed. There is in fact no overflow as we are dealing with constants. ROVER meanwhile, rewrites this expression to avoid this supposed datapath leakage. The Weight Calculation benchmark is a two-stage pipelined design computing pixel offsets in the graphics pipeline. ROVER optimizes each stage independently. By rewriting the MUX tree structure within each stage, using the "Sel Mul" rewrites, ROVER reduces the number of multipliers instantiated from five to three. The work of Verma *et al.* [3] has no ability to combine multipliers by manipulating the MUX tree structure, so can not reach these designs generated by ROVER.

The next two benchmarks are taken from [3], where ROVER generalizes and exceeds the capabilities of this prior work. The first example is a familiar finite impulse response (FIR) filter with 8-taps (a 3-tap version is shown in Figure VII-B). Via the "Arithmetic Logic Exchange" rewrites, ROVER explores all the alternative arithmetic clustering opportunities extracting an optimal clustering according to the theoretical cost metric. In contrast, the logic synthesis engine appears to greedily cluster all operators. This maintains carry-save representation throughout, but, we speculate, results in shifting carry-save representations, incurring additional circuit area overhead. The ADPCM decoder is a design which approximates a 16×4 multiplier. For this benchmark, both ROVER and the logic synthe-

TABLE III
LOGIC SYNTHESIS RESULTS COMPARING THE ORIGINAL AND ROVER OPTIMIZED DESIGNS UNDER TWO DIFFERENT SYNTHESIS CONSTRAINTS. FIRSTLY, AT THE MINIMUM DELAY WHICH BOTH DESIGNS COULD MEET AND SECONDLY, CONSTRAINED TO THE MINIMUM AREA THAT BOTH DESIGNS COULD MEET. DELAY, POWER AND AREA ARE MEASURED IN NS, μW AND μm^2, RESPECTIVELY. WE BOLD THE BEST RESULT FOR EACH METRIC.

Source	Benchmarks	Min Delay	Original			ROVER			Min Area	Original	ROVER
			Cells	Power	Area	Cells	Power	Area		Delay	Delay
Intel	Media Kernel	0.35	1759	959.4	167.3	**918**	**427.9**	**84.2** (−50%)	117.6	0.60	**0.30** (−50%)
	Weight Calculation	0.25	1353	927.1	75.3	**1030**	**719.4**	**57.8** (−23%)	39.8	0.84	**0.40** (−52%)
Open-Source	FIR Filter Kernel	0.67	8067	2839.0	552.6	**7846**	**1837.9**	**428.6** (−22%)	209.0	4.40	**4.09** (−07%)
	ADPCM Decoder [49]	0.12	620	197.4	41.8	**556**	**190.6**	**38.0** (−09%)	20.8	**0.84**	**0.84** (+00%)
	Shifted FMA	0.22	996	502.0	83.7	**855**	**445.1**	**68.6** (−18%)	54.6	0.85	**0.31** (−64%)
	Shift Mult	0.30	2864	1356.4	240.1	**1317**	**522.0**	**88.8** (−63%)	150.7	1.88	**0.26** (−86%)
	MCM(3,7,21)	0.12	**894**	**161.0**	**36.6**	1015	249.2	51.4 (+40%)	23.3	0.81	**0.58** (−28%)
	MCM(5,93)	0.12	**687**	**204.8**	**38.2**	778	292.0	53.6 (+40%)	22.4	0.73	**0.58** (−21%)
	MCM(7,19,31)	0.09	**1079**	**230.0**	**53.3**	1082	236.4	54.1 (+02%)	21.8	**0.72**	**0.72** (−00%)

TABLE IV
ROVER PERFORMANCE AND E-GRAPH SIZE BEFORE/AFTER REWRITING.

Benchmark	Init Nodes	Final Nodes	Extract	Runtime (sec)
Media Kernel	45	1312	ILP	10.67
Weight Calc.	107	3036	ILP	165.00
FIR Filter	30	8487	ILP	155.90
ADPCM	17	7290	Greedy	16.64
Shifted FMA	13	26	ILP	0.09
Shift Mult	13	72	ILP	0.13
MCM(3,7,21)	13	17493	ILP	135.00
MCM(5,93)	12	2986	ILP	113.86
MCM(7,19,31)	13	7601	ILP	50.59

sis engine achieve a complete clustering. ROVER manipulates the MUX tree structure, whilst the logic synthesis tool appears to add additional operators to facilitate the clustering.

The next two benchmarks demonstrate optimizations beyond the capabilities of [3]. Shifted FMA exploits multiplication-manipulating rewrites since logic synthesis tools will effectively cluster multiplications followed by additions to reduce the number of carry-propagate adders. As in the FIR filter example, the logic synthesis greedily clusters, such that it must perform a shift of a carry-save representation. By moving the shift ROVER enables a simpler arithmetic clustering. Shift Mult is a kernel extracted from a floating point multiplier that normalizes the product of two denormals. By re-ordering the shift and multiplication operators a smaller multiplier can be instantiated, reducing the circuit area. In contrast, the logic synthesis tool does not manipulate the higher-level dataflow graph to explore the interaction of arithmetic and logical operators, and does not discovers this opportunity. These ROVER optimizations are not reachable by [3], since their tool did not explore the interaction between multiplication and logic.

The "Constant Expansion" rewrites are valuable for the MCM benchmarks, where for $MCM(a_1,a_2,...,a_n)$ we ask ROVER to generate optimized RTL producing $\{a_1 \times x, a_2 \times x, ..., a_n \times x\}$. MCM(3,7,21) is an example taken from [19]. ROVER is able to match the operator count from [19], extracting a design that uses three addition/subtraction operators by sharing intermediate results. Such an architecture serializes the construction of $3 \times x$ and $21 \times x$, which at low delay targets introduces an area penalty, because the

original architecture can compute each result in parallel with no dependency. However, from the ROVER generated RTL a smaller circuit can be synthesized, as shown in Table III. For the MCM(5,93) benchmark ROVER is similarly able to use just 3 adders, matching the minimal adder count, and showing similar synthesis results to MCM(3,7,21). For the MCM(7,19,31) benchmark[1] ROVER recovers the standard CSD solution using 4 adders and matching the synthesis tool (hence the identical synthesis results). The minimal solution uses 3 adders, but is unreachable using ROVER's existing rewrites as it relies upon representing $19 = (31 + 7) \gg 1$.

In this work, we used the logic synthesis tool with all datapath optimizations enabled to provide a baseline. However, this baseline includes state-of-the-art datapath optimization techniques. If we disable these optimizations we get an alternative baseline that highlights the significance of the datapath optimizations built-in to the logic synthesis and those performed by ROVER. On average, with datapath optimization disabled the logic synthesis tool produced circuits 17.6% larger than with datapath optimization enabled, and 55.8% larger than the ROVER generated circuits. Furthermore, in 5 out of the 9 benchmarks, disabling datapath optimization led to timing violations in the synthesized netlists.

B. Bitwidth Dependent Architectures

In this section we consider parameterizable RTL designs. As the complexity of integrated circuits grows, reusable and parameterizable hardware has become increasingly popular amongst engineers and architects as it facilitates faster development. Each instance of this RTL will be synthesized using the same architecture. By contrast, ROVER automatically optimizes each instance generating a bespoke component that is optimized for a given instance.

To investigate whether ROVER can usefully adapt the architecture depending on parameter values, we considered a 3-tap FIR filter with parameterizable input bitwidths. We passed ROVER each design, increasing the input bitwidth parameter from 4 to 64 and allowed ROVER to explore the design space for each parameterization. As shown in Figure 9, ROVER extracted one of three distinct architectures. In the FIR kernel

[1]Thank you to the anonymous reviewer for providing this benchmark.

(a) Architecture 0 {4,8}

(b) Architecture 1 {12,...,24}
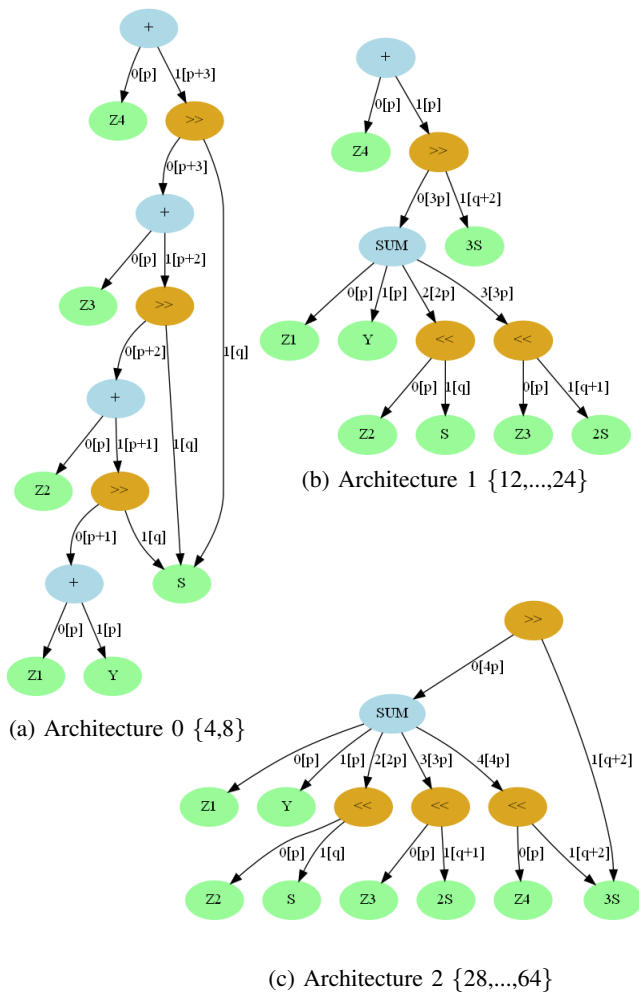
(c) Architecture 2 {28,...,64}

Fig. 9. Simplified FIR filter data-flow graphs representing optimal architectures for different choices of the input bitwidth parameter p and shift bitwidth parameter q. Edge labels indicate the operator index and bitwidth in square brackets. The sets in curly braces are bitwidths for which that architecture is optimal. In these graphs $2S$ and $3S$ are constant multiples of S.
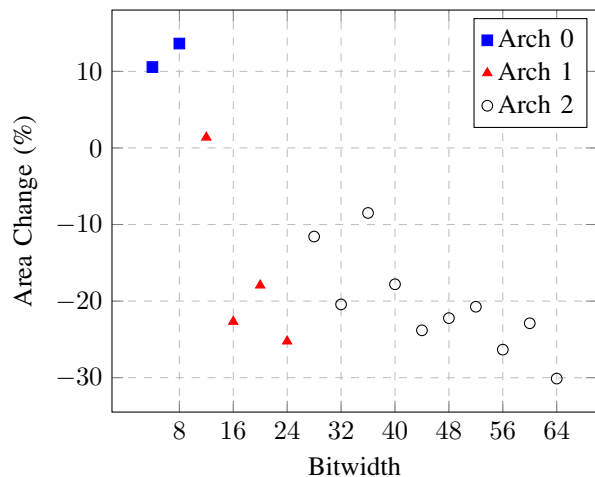

Fig. 10. Synthesis results for the 3-tap FIR kernel at a range of different bitwidths. We synthesized both the ROVER generated RTL and original RTL (Architecture 0) with a minimum delay objective. We plot the relative change in area and delay against the baseline.

customization without sacrificing IP quality.

C. Performance

Table IV presents benchmark properties and optimization statistics. For the ILP extraction method, we set a timeout limit of 120 seconds and in all the longer running benchmarks, ILP solving dominated the runtime. Note that the number of ILP constraints is proportional to the number of nodes in the final e-graph. Whilst ILP scalability is a concern, the modular nature of RTL design ensures that we rarely meet large scale problems. We resorted to the faster greedy `egg` extraction method [28] for the ADPCM decoder since there was no scope to exploit common sub-expressions in this benchmark. Extraction method selection is a user defined option for ROVER. We note that the final e-graph size is not well correlated with the number of operators in the initial e-graph. The size of the final e-graph depends more upon the structure of the initial design.

Highlighting the importance of the verification flow, for the Media Kernel and Shift Mult benchmarks, the EC returned inconclusive results, even when running for several hours, when only given the original and ROVER generated RTLs. Using the ROVER generated problem decomposition, the correctness of the generated RTL could be proven in seconds. For all other benchmarks presented here, the EC could prove the equivalence of the original and ROVER generated RTLs without the problem decomposition described in Section VI.

VIII. Cost Metric Evaluation

The primary objective of the theoretical cost metric is to steer the extraction process in order to generate an optimized architecture. Previously, we evaluated the noise floor in logic synthesis to understand inherent variability of such a complex tool [10]. We used an approach known as performance fuzzing [50], [51], that differs from the more traditional application of fuzzing to automated bug detection [50]. We
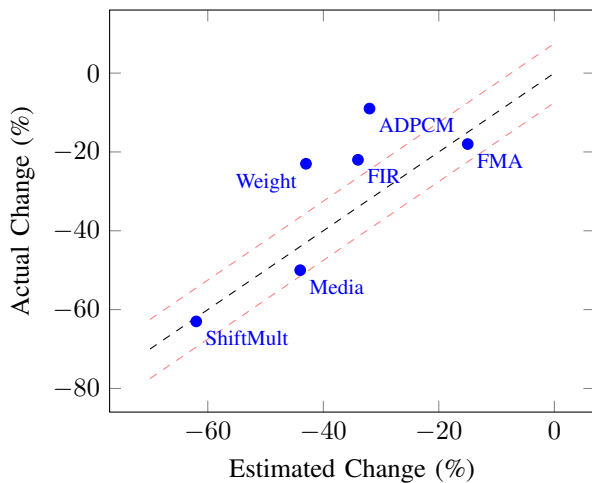
testcase the benefits of clustering consecutive additions into a `SUM` node compete with the additional shift operations required to facilitate the merging. Note that Architecture 0 uses four carry-propagate adders, Architecture 1 uses two carry-propagate adders, whilst Architecture 2 uses only a single carry-propagate adder at the expense of additional shifting logic. ROVER automatically detects the point at which this tradeoff becomes favourable.

For each bitwidth, we synthesized Architecture 0 and the distinct ROVER generated RTL (which implements either Architecture 0, 1 or 2) at the minimum delay target that both can meet. Figure 10 plots synthesis results at each bitwidth comparing against the baseline, Architecture 0 (Figure 9). The architectural selections made by ROVER reduce the circuit area by up to 30% and by 15% on average. For 4-bit and 8-bit designs, ROVER increases the circuit area despite deploying the same architecture as the baseline. This is due to synthesis noise, an effect quantified precisely in [10]. Using ROVER to automatically generate an optimized design for each parameterization allows engineers to avoid manual

Fig. 11. ROVER's predicted percentage change vs. the actual percentage change based on logic synthesis at the minimum delay target. Points above/below the diagonal indicate that ROVER over/under-predicts the area reduction. We omit the MCM results. Red lines represent the synthesis noise window.

randomly applied non-functional mutations to designs, for example renaming a variable in RTL, and observed up to a 15% difference in logic synthesis area. ROVER's cost model cannot be expected to capture this. The variability is equally likely to benefit ROVER as it is to be detrimental for the results shown in Table III. However, the overall benefit demonstrated by ROVER is statistically significant and explainable.

To evaluate the accuracy of the cost model, we plot the ROVER estimated circuit area reduction against the actual change seen in the logic synthesis results at the minimum delay target in Figure 11. The graph shows that ROVER both under- and over-estimates the benefit of its optimizations but does provide a reasonable indicator. The ADPCM and Weight benchmarks exhibit significant over-estimates. In the ADPCM example, ROVER manipulates the MUX tree structure of the design to enable arithmetic clustering, which the synthesis tool exploits successfully. Analyzing the datapath extraction report generated during synthesis of the original ADPCM design, we see that the synthesis tool is already capable of manipulating this design to cluster the arithmetic operations limiting the observable benefit of ROVER's optimizations. For the Weight Calculation benchmark, ROVER reduces the number of multipliers instantiated by two. In the original design, the synthesis tool includes these multipliers in a datapath cluster, therefore the circuit area benefit is less than the full multiplier area cost. The omitted MCM benchmarks highlight the limitations of an area only model, as the benefit depends upon the delay target.

IX. CONCLUSION

This paper presents methods to exploit the properties of the e-graph data structure, finding an ideal application in the RTL optimization problem. E-graphs greatly simplify this problem by avoiding any need to specify an order in which to apply transformations whilst maintaining bit identical functionality. The e-graph's foundations rest on functional equivalence principles, which are crucial in hardware design where the correctness requirements are higher than most

other domains. By defining a set of parameterized bitvector-manipulating transformations, learnt from Intel engineers, we have matched human-engineered designs in terms of circuit quality. The productivity and circuit quality benefits that stem from automated rewriting techniques allow engineers to write behavioural, less bug prone designs and leave the optimization to a tool that can provide verified implementations.

Future work will seek to address delay optimization; this will allow ROVER to select different arithmetic operator architectures depending on the timing budget available. We will also address the limitations of the rewrite condition synthesis flow, which currently relies upon an unproven extrapolation assumption. We believe the integration of a theorem prover such as ACL2 [52] will allow us to prove this assumption. For extraction, we will resolve the ILP bottleneck in ROVER's current implementation, by leveraging the outcome of a community effort to improve common sub-expression aware extraction[2]. Lastly, as noted in the MCM discussion, there are scenarios in which bespoke tools yield optimal solutions more efficiently. Through dynamic rewrites, we will provide an interface to such tools.

REFERENCES

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *Transactions on Embedded Computing Systems*, vol. 13, no. 2, 2013.
[2] Cadence, "Stratus HLS," 2023. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html
[3] A. K. Verma, P. Brisk, and P. Ienne, "Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1761–1774, 2008.
[4] S. Xydis, K. Pekmestzi, D. Soudris, and G. Economakos, "Compiler-in-the-loop exploration during datapath synthesis for higher quality delay-area trade-offs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 18, no. 1, 2012.
[5] D. S. Harish Ram, M. C. Bhuvaneswari, and S. S. Prabhu, "A novel framework for applying multiobjective GA and PSO based approaches for simultaneous area, delay, and power optimization in high level synthesis of datapaths," *VLSI Design*, vol. 2012, 2012.
[6] V. Krishnan and S. Katkoori, "A genetic algorithm for the design space exploration of datapaths during high-level synthesis," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, 2006.
[7] R. Roy, J. Raiman, N. Kant, I. Elkin, R. Kirby, M. Siu, S. Oberman, S. Godil, and B. Catanzaro, "PrefixRL: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning," in *Proceedings - Design Automation Conference*, vol. 2021-December, 2021.
[8] K. D. Cooper and L. Torczon, *Engineering a compiler: Second edition*. Elsevier, 2011.
[9] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: a new paradigm for arithmetic circuit design," in *Proceedings -Design, Automation and Test in Europe, DATE*, 2008.
[10] S. Coward, G. A. Constantinides, and T. Drane, "Automatic Datapath Optimization using E-Graphs," in *IEEE 29th Symposium on Computer Arithmetic (ARITH)*. IEEE, 9 2022, pp. 43–50.
[11] R. Zimmermann, "Datapath synthesis for standard-cell design," in *19th IEEE Symposium on Computer Arithmetic*, 2009.
[12] Synopsys, "Coding Guidelines for Datapath Synthesis," Synopsys, Mountain View, Tech. Rep., 12 2019.

[2]https://github.com/egraphs-good/extraction-gym

[13] C. Seger, "Voss II," Chalmers, 2023. [Online]. Available: https://github.com/TeamVoss/VossII

[14] J. Pope and C.-J. H. Seger, "Bifröst: Creating Hardware With Building Blocks," in *2023 Forum on Specification & Design Languages (FDL)*, 2023, pp. 1–8.

[15] E. Ustun, I. San, J. Yin, C. Yu, and Z. Zhang, "IMpress: Large Integer Multiplication Expression Rewriting for FPGA HLS," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–10.

[16] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to RTL equivalence checking," in *Proceedings -Design, Automation and Test in Europe, DATE*, 2009.

[17] S. Coward, E. Morini, B. Tan, T. Drane, and G. Constantinides, "Datapath Verification via Word-Level E-Graph Rewriting," in *Formal Methods in Computer-Aided Design*, 8 2023. [Online]. Available: http://arxiv.org/abs/2308.00431

[18] C. Yu and M. Ciesielski, "Automatic word-level abstraction of datapath," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 1718–1721.

[19] O. Gustafsson, "A difference based adder graph heuristic for multiple constant multiplication problems," in *IEEE International Symposium on Circuits and Systems*, 2007, pp. 1097–1100.

[20] R. I. Hartley, "Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers," *IEEE Transactions on Circuits and Systems*, vol. 11, 1996.

[21] M. Kumm, "Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays," Ph.D. dissertation, Universität Kassel, Kassel, Germany, Wiesbaden, 2016.

[22] M. D. Ercegovac and T. Lang, *Digital arithmetic*. Elsevier, 2004.

[23] F. De Dinechin, S. I. Filip, M. Kumm, and A. Volkova, "Towards Arithmetic-Centered Filter Design," in *Proceedings - Symposium on Computer Arithmetic*, vol. 2021-June, 2021.

[24] M. Kumm, "Optimal constant multiplication using integer linear programming," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, 2018.

[25] R. Garcia and A. Volkova, "Toward the Multiple Constant Multiplication at Minimal Hardware Cost," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 5, 2023.

[26] N. Fiege, M. Kumm, and P. Zipf, "Bit-Level Optimized Constant Multiplication Using Boolean Satisfiability," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 1, pp. 249–261, 2024.

[27] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.

[28] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," in *Proceedings of the ACM on Principles of Programming Languages*, vol. 5, no. POPL, 2021.

[29] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *ACM SIGPLAN Notices*, vol. 44, no. 1. Association for Computing Machinery, 2009.

[30] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed super-optimizer," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2002.

[31] L. De Moura and N. Bjørner, "Efficient E-matching for SMT solvers," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4603 LNAI, 2007.

[32] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *Journal of the ACM*, vol. 52, no. 3, 2005.

[33] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *ACM SIGPLAN Notices*, vol. 47, no. 10, 2012.

[34] S. Coward, G. A. Constantinides, and T. Drane, "Abstract Interpretation on E-Graphs," 3 2022. [Online]. Available: https://arxiv.org/abs/2203.09191

[35] O. Flatt, S. Coward, M. Willsey, Z. Tatlock, and P. Panchekha, "Small Proofs from Congruence Closure," in *Formal Methods in Computer-Aided Design*, 9 2022.

[36] L. De Moura and N. Bjørner, "Z3: An efficient SMT Solver," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4963 LNCS. Springer, 2008.

[37] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 1–11, 2015.

[38] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock, "Synthesizing structured CAD models with equality saturation and inverse transformations," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 31–44.

[39] E. Ustun, C. Yu, and Z. Zhang, "Equality Saturation for Datapath Synthesis: A Pathway to Pareto Optimality," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[40] A. Beaumont-Smith and C.-C. Lim, "Parallel prefix adder design," in *Proceedings - IEEE Symposium on Computer Arithmetic*, 2001, pp. 218–225.

[41] I. Koren, *Computer arithmetic algorithms*. AK Peters/CRC Press, 2018.

[42] D. Thomas and P. Moorby, *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[43] G. Steele, *Common LISP: the language*. Elsevier, 1990.

[44] M. Popoloski, "Slang," 2023. [Online]. Available: https://github.com/MikePopoloski/slang

[45] A. Solar-Lezama, "Program Synthesis by Sketching," Ph.D. dissertation, 2009.

[46] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[47] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu, "SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra," *Proceedings of the VLDB Endowment*, vol. 13, no. 11, 2020.

[48] J. Forrest, T. Ralphs, H. G. Santos, S. Vigerske, J. Forrest, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, Jan-Willem, rlougee, jpgoncal1, S. Brito, h-i-gassmann, Cristina, M. Saltzman, tosttost, B. Pitrus, F. MATSUSHIMA, and to-st, "coin-or/Cbc: Release releases/2.10.10," 4 2023. [Online]. Available: https://zenodo.org/record/7843975

[49] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the Annual International Symposium on Microarchitecture*, 1997.

[50] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," 2020.

[51] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, "Mariposa: Measuring SMT Instability in Automated Program Verification," in *Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design – FMCAD*. TU Wien Academic Press, 2023, pp. 178–188.

[52] W. A. Hunt, M. Kaufmann, J. S. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, 2017.

Samuel Coward received a BSc in Mathematics in 2018 and an MPhil in Scientific Computing in 2019, from the University of Cambridge. He is currently studying for a PhD in Electrical and Electronic Engineering at Imperial College London, whilst also working as a Graphics Hardware Engineer at Intel Corporation. Samuel's research focuses on automating RTL design and program analysis techniques to increase chip design productivity and quality.

George A. Constantinides (S'96, M'01, SM'08) received the Ph.D. degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Professor of Digital Computation and Associate Dean of Engineering. He has served as chair of the FPGA, FPL and FPT conferences. He currently serves on several program committees and has published over 200 research papers in peer refereed journals and international conferences. Prof. Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.

Theo Drane started working for the Datapath consultancy, Arithmatica, in 2002 after a Mathematics degree from the University of Cambridge, UK. He moved to Imagination Technologies in 2005, where he subsequently founded their Datapath team while studying for a PhD at Imperial College London's EEE Department. In December 2018, after a stint within Cadence Design System's Logic Synthesis division, Genus, he joined Intel's Graphics Group. His applied research Numerical Hardware & System Level Design Group focuses on all aspects of architecting, implementing, optimizing and verifying math intensive hardware.