Automatic Datapath Optimization using E-Graphs

Samuel Coward
Numerical Hardware Group
Intel Corporation
Email: samuel.coward@intel.com

George A. Constantinides
Electrical and Electronic Engineering
Imperial College London
Email: g.constantinides@imperial.ac.uk

Theo Drane
Numerical Hardware Group
Intel Corporation
Email: theo.drane@intel.com

Abstract—**Manual optimization of Register Transfer Level (RTL) datapath is commonplace in industry but holds back development as it can be very time consuming. We utilize the fact that a complex transformation of one RTL into another equivalent RTL can be broken down into a sequence of smaller, localized transformations. By representing RTL as a graph and deploying modern graph rewriting techniques we can automate the circuit design space exploration, allowing us to discover functionally equivalent but optimized architectures. We demonstrate that modern rewriting frameworks can adequately capture a wide variety of complex optimizations performed by human designers on bit-vector manipulating code, including significant error-prone subtleties regarding the validity of transformations under complex interactions of bitwidths. The proposed automated optimization approach is able to reproduce the results of typical industrial manual optimization, resulting in a reduction in circuit area by up to 71%. Not only does our tool discover optimized RTL, but also correctly identifies that the optimal architecture to implement a given arithmetic expression can depend on the width of the operands, thus producing a library of optimized designs rather than the single design point typically generated by manual optimization. In addition, we demonstrate that prior academic work on maximally exploiting carry-save representation and on multiple constant multiplication are both generalized and extended, falling out as special cases of this paper.**

Index Terms—**hardware optimization, design automation, datapath design**

I. INTRODUCTION

In industry and academia, Register Transfer Level (RTL) development is limited to minimal design space exploration due to the complexity of design space and is slowed down by long debug timelines. RTL optimization of datapath designs is still often a manual task, as synthesis tools are unable to achieve the results of a skilled engineer [1].

A key observation is that manual RTL optimization is typically performed by applying a number of known 'useful' transformations to a design. These transformations, and their domain of validity, are accumulated through years of engineer design experience. In combination, these transformations may result in substantial changes to the underlying RTL. Apart from some simple transformations implemented automatically in modern ASIC design tools [2], the process of determining a sequence of transformations to apply to an RTL design is currently based on designer intuition [3], largely due to the non-convex nature of the design space: it is often necessary to apply an early transformation that results in a worse-quality circuit before then applying a later one leading to an overall improvement. It is this process we seek to automate.

Such automation facilitates the creation of bitwidth dependent architectures, where different parameterisations may result in different architectures: the best design approach for narrow bitwidths may not be the best design approach for wider bitwidths. A range of RTLs automatically generated from a single parameterisable input retains the ease-of-use benefits of parameterisable RTL without sacrificing quality.

Existing commercial synthesis tools are capable of merging together consecutive additions in a circuit design to make best use of carry-save representations [1]. However when arithmetic is interspersed with logic, the tools frequently miss potential optimization opportunities [4].

We aim to leverage existing commercial synthesis tools by transforming RTL to a form the existing tools can maximally optimize. We focus on combinational RTL, although the techniques described are equally applicable to pipelined designs via retiming. Given a design in the form of an RTL implementation R, we aim to find an RTL implementation R' that minimises $\text{cost}(R')$ for some cost function, such that the two RTLs are functionally equivalent, $R \simeq R'$. We define the equivalence relation $\simeq$ as $R \simeq R'$ if and only if for all possible inputs, all outputs of R and R' are equal.

We represent such RTL as a data-flow graph, where operators and operands are represented by nodes with edges, labelled by bitwidth, connecting operands and operators. This graphical representation allows us to formulate the problem as a graph optimization problem, where we are allowed to manipulate the graph with equivalence-preserving transformations. This formulation allows us to take advantage of recent advances in e(quivalence)-graph and equality saturation [5] technology, discussed in Section II, alongside previous motivating work in automated RTL optimization and design. Application of e-graphs to the RTL optimization problem is presented in Section III. We demonstrate results in Section IV and validate our cost metric in Section V.

The paper contains the following novel contributions:

- application of e-graphs and equality saturation to automate datapath RTL optimization,
- a precisely defined set of rewrites that facilitate efficient design space exploration together with their domains of applicability in designs utilizing multiple bitwidths,
- an automated method to optimize architectures as a function of bitwidth parameters,
- quantification of a 'noise floor' in datapath logic synthesis using 'fuzzing' techniques from software testing.

II. BACKGROUND

A. Datapath Optimization

A useful example of transformation-based datapath improvement comes from Verma, Brisk and Ienne [4], which automatically applies data-flow transformations to maximally exploit carry-save representation. Their primary objective is to cluster additions together in the data-flow graph, a useful target as full carry-propagate addition is generally expensive and can often be avoided. This can be done by deploying compressor trees, circuits taking three or more input words which get reduced to two output words: a carry and a save. Using a carry-propagate adder to sum the carry and the save returns the sum of all the inputs. This can be beneficial as, for example, combining two consecutive carry-propagate adders into a compressor tree and one carry-propagate adder saves the cost of one carry-propagate adder at the expense of a compressor tree. We generalize this work using our methodology, which is able to replicate the results obtained in [4] as a special case.

Another well-studied transformation beyond the reach of standard commercial synthesis tools is the multiple constant multiplication (MCM) problem [6], [7]. The MCM problem asks, given a set of integer coefficients $\{a_1, ..., a_n\}$ and variable x, what is the optimal architecture to compute the set $\{a_1 \times x, ..., a_n \times x\}$? Competing solutions use a fixed number representation of the constants [7], often canonical signed digit (CSD) representation [8], and/or deploy an adder graph algorithm [6]. A transformation based approach also captures both of these methods.

In addition to these special cases, there is a wide variety of transformations that can be captured through standard arithmetic rewrites, *e.g.* associativity, distributivity, *etc.* Often these rewrites interact with each other, in the sense that applying one type of transformation opens or closes the door to applying a different class of transformation.

B. E-graphs and Equality Saturation

Equivalence graphs, commonly called e-graphs, provide a dense representation of equivalence classes (e-classes) of expressions [9]. Often found in theorem provers, this data structure enables a graph optimization technique called equality saturation [5], [10], [11]. The e-graph represents expressions, where the nodes, known as e-nodes, represent functions (including variables and constants, as 0-arity functions) and are partitioned into a set of e-classes. The intuition is that e-classes can be used to compactly represent *equivalent* expressions, whose evaluation always leads to the same result. Edges represent function inputs and are from e-nodes to e-classes; see Figure 1, where dashed lines represent e-class boundaries, solid ellipses are e-nodes and arrows are edges. We define $\mathcal{C}$ to be the set of e-classes, $\mathcal{N}$ the set of e-nodes and $E \subseteq \mathcal{N} \times \mathcal{C}$ the set of edges. We also introduce $\mathcal{N}_c$ to denote the set of e-nodes in a given e-class c.

Rewrites define equivalences over expressions, for example $x + x \rightarrow 2 \times x$ says that $x + x$ is equivalent to $2 \times x$. Such

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

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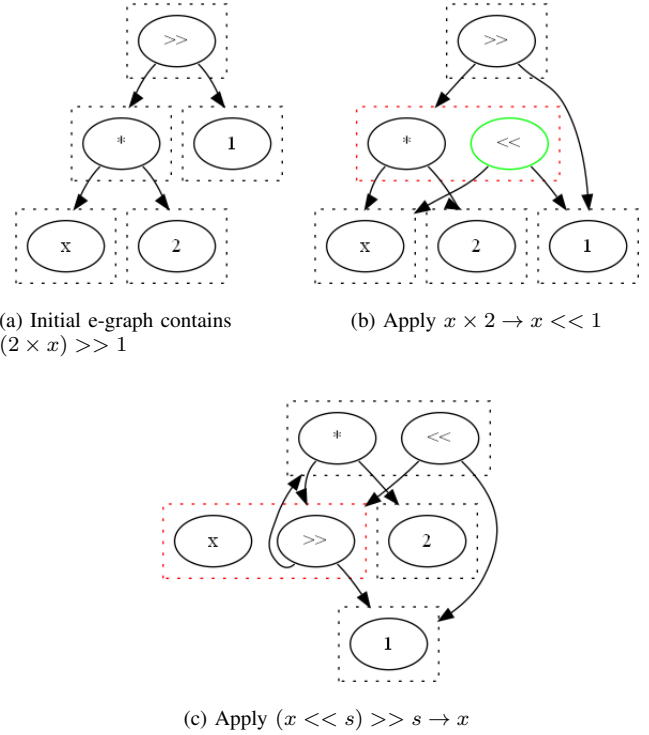
(c) Apply $(x << s) >> s \rightarrow x$

Fig. 1. 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.

rewrites are applied constructively to the e-graph, meaning that the left hand side of the rewrite remains in the data structure. Constructive rewrite application avoids the concern of which order to apply rewrites in. As rewrites are applied, the e-graph grows monotonically, representing more and more equivalent expressions, and hence naturally capturing the interaction between different rewrite rules.

Equality saturation provides us with a stopping condition. At the point where further rewrites add no additional information, we say that the e-graph has saturated. From an e-graph representing potentially infinitely many equivalent expressions we may choose the "best" expression [5].

egg is a recent Rust e-graph library, which is intended to be a general purpose and reusable implementation [5]. It adds powerful performance optimizations over existing, usually bespoke, e-graph implementations along with some useful additional features. It has been used to automatically improve the numerical stability of floating point expressions [12], map programs onto hardware accelerators [13] and optimize linear algebra [14]. To build a functioning e-graph optimization tool, egg must be supplied with a language definition – that is a set of operator names together with their arity, and a rewrite set – that is a set of equivalences over the given language definition.

III. METHODOLOGY

This section demonstrates how e-graphs can be applied to the RTL optimization problem. We use a natural graphical
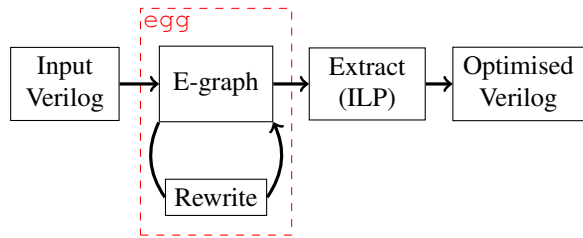
Fig. 2. Flow diagram describing the operation of the tool.

TABLE I
OPERATORS DEFINED IN OUR `egg` IMPLEMENTATION OF RTL
OPTIMIZATION. WE INCLUDE THE ARCHITECTURE CHOSEN FOR
THEORETICAL COST ASSIGNMENT.

Operator	Symbol	Arity	Architecture
Left/Right Shift	$<< / >>$	2	Mux Tree
Addition/Subtraction	+/-	2	Prefix Adder (PA) [17]
Negation	-	1	PA
Multiplication	$\times$	2	Booth Radix-4 [18]
Multiplexer	$\cdot ? \cdot : \cdot$	3	Mux gates
Not/Inversion	$\sim$	1	One-input gates
Concatenate	{,}	n	Wiring
Comparison	$> / <$	2	PA
Sum	SUM	n	CSA and PA
Muxed Mult Array	MUXAR	3	Array Reduction and PA
Fused Multiply-Add	FMA	3	Booth Radix-4

(a) Consecutive additions   (b) Merged additions encoded as a SUM

Fig. 3. The edge labels contain the operand's index and the operand's bitwidth in square brackets.

representation of RTL, using data-flow graphs allowing us to fit the optimization problem into the e-graph framework. In this framework the e-graph contains classes of equivalent bitvector manipulating expressions and the rewrites transform such expressions to alternative equivalent expressions. The tool parses input Verilog using Yosys [15] and converts it into nested S-expressions in Common Lisp [16].

    `term::=(operator [term] [term]...[term])`
The syntax is defined by the language described in Section III-A. These expressions are converted into e-graphs by `egg`. From the e-graph we extract a nested S-expression from which RTL is automatically generated, writing one operation per line. Figure 2 provides a flow diagram of the tool.

A. Language

We consider the problem from the abstraction level of a Verilog parser and operate on finite length bitvectors. We target bitvector arithmetic and bitwise Boolean operations as they form the basis of low-level datapath optimization. Including the bitwidth of these vectors is crucial to correctly model the circuit's behaviour. Bitwidths are also essential to correctly evaluate the cost of a given operation, clearly an 8-bit addition is less expensive than a 32-bit addition. The defined operations represented by nodes in the e-graph are described in Table I, and are defined for all inputs.

The first set in the table is a subset of the operators defined in Verilog [19]. These operators are fundamental in most arithmetic circuit designs and manipulating designs using them can have significant effects on power, performance and area. Bitwidth information is included in the e-graph a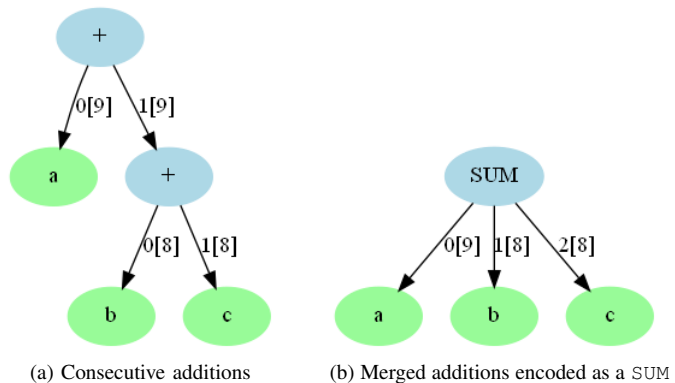s edge labels between the nodes. In addition to these basic operators, we introduce a second set of operators in Table I (below the dashed line), which encode the merging and sharing capabilities of modern synthesis tools. In [4], the benefits of merging adjacent additions to maximise the use of carry-save are exploited. We are able to capture the same effect in an automated manner through introduction of a SUM node in our language definition, which combines an arbitrary number of additions into a single node. This node encodes the compressor tree and carry-propagate adder described in Section I. Figure 3 shows how consecutive additions can be rewritten as a SUM node. Other such merging operators are the fused multiply-add (FMA) and the less familiar muxed multiplication array (MUXAR), which blends two disjoint multiplication arrays into one. These are discussed further in Section III-B.

B. Rewrites

We have identified and formalized rules that capture many of the manual transformations that Intel's Numerical Hardware Group regularly deploy manually. The set of rewrites described in Table II define equivalences over expressions. Each of the rewrites incorporates bitwidth information, where bitwidths are associated with operands. We introduce a left subscript notation, $_p x$, to denote a bitvector x with length p bits. Left subscript takes the lowest operator precedence.

`egg` supports conditional rewrites. We make use of this facility to allow us to decide whether a rule may be applied based on its bitwidth information. The sufficient (but not always necessary) conditions under which each rule applies are described in Table II. As an example, consider the "Distribute Mult Over Add" rewrite, where the sufficient condition is a function of the bitwidth parameters, $\min(q, u, v) \geq r$. When this condition is satisfied the rewrite can be safely applied as the equivalence holds. For some rewrites, a parameter appears on the right hand side but not on the left, for example q in the associativity rewrites. This corresponds to a condition constraining but not fully determining the values that these undefined parameters can take, in which case we select the minimum feasible bitwidth. The conditions have been validated by creating a parameterizable SMT query for each rewrite using the theory of fixed size bitvectors [20],

Class	Name	Left-hand Side → Right-hand Side	Sufficient Condition
Bitvector Arithmetic Identities	Commutativity	$_r(_p a *_q b) \to _r(_q b *_p a)$	True
	Mult Associativity	$_t(_u(_p a \times _r b) \times _s c) \to _t(_p a \times_q(_r b \times_s c))$	$(q \geq t \vee r+s \leq q)$ $\wedge(u \geq t \vee p+r \leq u)$
	Add Associativity	$_t(_u(_p a +_r b) +_s c) \to _t(_p a +_q(_r b +_s c))$	$(q \geq t \vee \max(r,s) < q)$ $\wedge(u \geq t \vee \max(p,r) < u)$
	Distribute Mult over Add	$_r(_p a \times_q(_s b +_t c)) \to _r(_u(_p a \times_s b) +_v(_p a \times_t c))$	$\min(q,u,v) \geq r$
	Sum Same	$_q(_p a +_p a) \to _q(_2 2 \times_p a)$	True
	Mult Sum Same	$_r(_s(_p a \times_q b) +_q b) \to _r(_t(_p a +_1 1) \times_q b)$	$t > p \wedge s \geq p+q$
	Add Zero	$_p(_p a +_q b) \to _p(a)$	$b \equiv 0 \mod 2^p$
	Sub to Neg	$_r(_p a -_q b) \to _r(_p a +_q(-_q b))$	True
	Mult by One	$_p(_p a \times_q b) \to _p(a)$	$b \equiv 1 \mod 2^p$
	Mult by Two	$_r(_p a \times_2 2) \to _r(_p a <<_1 1)$	True
Bitvector Logic Identities	Merge Left Shift	$_r(_u(_p a <<_q b) <<_s c) \to _r(_p a <<_t(_q b +_s c))$	$t > \max(q,s) \wedge u \geq r$
	Merge Right Shift	$_r(_u(_p a >>_q b) >>_s c) \to _r(_p a >>_t(_q b +_s c))$	$t > \max(q,s) \wedge u \geq p$
	Redundant Sel	$_p(_1 b ? _p a : _p a) \to _p a$	True
	Neg Not	$_r(-_p a) \to _r(_p(\sim(_p a)) +_1 1)$	$r \leq p$
	Not over Con	$_r(\sim(_{q+s}\{_q a, _s b\})) \to _r\{_q(\sim(_q a)), _s(\sim(_s b))\}$	$q + s \geq r$
Constant Expansion	Mult Constant	$_r(_q c \times_p x) \to$ $_r(_r(_q(_2 2 \times _{q-1} c[q-1:1]) \times_p x) +_p(_1 c[0] \times_p x))$	c constant
	One to Two Mult	$_p(_1 1 \times_p x) \to _p(_q(_2 2 \times_p x) -_p x)$	$q > p$
Arithmetic Logic Exchange	Left Shift Add	$_r(_s(_p a +_q b) <<_t c) \to _r(_u(_p a <<_t c) +_u(_q b <<_t c))$	$(s \geq r \vee \max(p,q) < s) \wedge u \geq r$
	Add Right Shift	$_r(_p a +_q(_t b >>_u c)) \to _r(_v(_s(_p a <<_u c) +_t b) >>_u c)$	$q \geq t \wedge s \geq p + 2^u - 1$ $\wedge v > \max(s,t)$
	Left Shift Mult	$_r(_t(_p a \times_q b) <<_u c) \to _r(_v(_p a <<_u c) \times_q b)$	$t \geq r \wedge v \geq r$
	Sel Add	$_r(_1 e ? _r(_p a +_q b) : _r(_p c +_q d)) \to$ $_r(_p(_1 e ? _p a : _p c) +_q(_1 e ? _q b : _q d))$	True
	Sel Add Zero	$_p(_1 e ? _p(_p a +_q b) : _p c) \to _p(_p(_1 e ? _p a : _p c) +_q(_1 e ? _q b : 0))$	True
	Move Sel Zero	$_r(_p(_1 b ? _p 0 : _p a) \times_q c) \to _r(_p a \times_q(_1 b ? _q 0 : _q c))$	True
	Concat to Add	$_r\{_p a, _q b\} \to _r(_s(_p a <<_u q) +_q b)$	$s \geq p + 2^u - 1 \wedge u \geq \lceil \log_2(q+1)\rceil$
Merging Ops	Merge Additions	$_{q_1}(_{p_1} a1 +_{q_2}(_{p_2} a2 +_{q_3}(_{p_3} a3 + ... +_{p_n} an)...)) \to$ $_{q_1}(\text{SUM}(_{p_1} a1, _{p_2} a2, ..., _{p_n} an))$	$q_i > \max(p_i, q_{i+1}), i=1,...,n-2$ $\wedge q_{n-1} > \max(p_{n-1}, p_n)$
	Merge Mult Array	$_t(_s(_q a \times_r b) +_s(_q c \times_r(\sim(_r b))))$ $\to _t(\text{MUXAR}(_r b, _q a, _q c))$	$s \geq q+r \wedge t > s$
	FMA Merge	$_t(_s(_p a \times_q b) +_r c) \to _t(\text{FMA}(_p a, _q b, _r c))$	$s \geq p+q \wedge t > \max(s,r)$

which have been checked for all combinations of bitwidths in $\{1,...,10\}$, however a formal equivalence check between the generated and original RTL ensures that we do not need to trust the correctness of our rewrites or of the egg library.

Arithmetic Logic Exchange rewrites are inspired by a set taken from [4], which focus on the interplay of logic and addition. To extend the prior work we have added rewrites that are able to move other arithmetic operators and we have generalized the rules so they are able to be applied in a multiple bitwidth setting.

The "Merging Ops" rewrites allow us to correctly evaluate the cost of specific sequences of operations, that logic synthesis tools are able to effectively optimize. We have seen that the SUM node merges consecutive additions, highlighting that the area usage for consecutive additions is not additive. This captures what logic synthesis tools will do to an expression such as $a + b + c$, converting it to a compressor tree and deploying a single carry-propagate adder. Further merging optimization capabilities of Synopsys Design Compiler are documented [1]. The "Merge Mult Array" rewrite does not make use of carry-save format but identifies that two disjoint

multiplication arrays can be merged. Letting $a[i]$ represent bit i of a and $u = \lceil\log_2(r)\rceil$, MUXAR is shorthand for the right hand side of the rewrite, where the SUM represents array reduction:

$$_t(_s(_q a \times _r b) +_s(_q c \times _r(\sim(_r b)))) \to$$
$$_t(\text{SUM}(_s(_q(_1 b[0]? _q a : _q c) <<_u 0),$$
$$_s(_q(_1 b[1]? _q a : _q c) <<_u 1),...,$$
$$_s(_q(_1 b[r-1]? _q a : _q c) <<_u (r-1)))).$$

Logic synthesis is capable of exploiting this optimization if it identifies an expression of the form $(a \times b) + (c \times \sim b)$, but we must indicate the merging opportunity to our tool.

We added the "Constant Expansion" rules to re-express multiplication of a variable by a constant. These rules allow us to recreate results from the literature described in Section II-A on the MCM problem. In addition to the explicit rewrite rules, constant folding is implemented as an e-class analysis in egg [5].

C. Extraction

Having applied rewrites to the e-graph until saturation or timeout limits are reached, we now must extract the optimal

design from potentially infinitely many choices. The extraction process must select a set of e-classes to implement and for each e-class, which e-node within that class to implement, subject to the constraint that the e-class children of each selected e-node must also be selected. Choosing an optimal design requires some metric that allows us to discriminate between competing implementations. Industrial circuit design is typically judged on area, latency and power consumption. In this contribution we will only use an area metric, therefore our definition of optimal will be the smallest circuit implementation.

We have developed a theoretical area estimate cost function in terms of the number of two-input gates required for the operator. It assigns a cost per operator that is a function of the input and output bitwidths. Table I lists the architectures on which we base the cost of the more complex operators. We introduce different costs for when at least one of the operands is a constant. The cost of a complete design is then the sum of the operator costs, and the objective is to minimise this cost.

The major benefit of using a theoretical cost metric as opposed to using metrics derived from logic synthesis or HLS tools is the computation speed which, when combined with equality saturation, enables effective design space exploration. Cost metric validation is addressed in Section V.

By explicitly introducing rules for operator merging and sharing, we are able to define a cost for each node based only on its type and argument bitwidths, capturing downstream synthesis optimizations for SUM, MUXAR and FMA.

When extracting an RTL implementation from an e-graph, one is immediately faced with the question of common subexpressions. Common subexpressions are naturally extracted as part of the e-graph construction process [5] and ideally we would want to utilize this information in the resulting hardware, for example extracting $(x + 1) \times (x + 1)$ as let $y = x + 1$ in $y \times y$. However, this makes the extraction problem an inherently global problem over the e-classes, in the sense that the optimal e-node implementation for a given e-class may depend on the selected e-node implementation of the other e-classes in the graph. Previous solutions have solved this by posing optimal extraction as an integer linear programming (ILP) problem [13], [14], and we follow the same approach in this work.

Using the notation defined in Section II-B, for each e-node $n \in \mathcal{N}$ we associate a cost, $\text{cost}(n)$, given by the theoretical cost function, and a binary variable $x_n \in \{0, 1\}$, which indicates whether n is implemented in the final extracted RTL. Our objective is to minimize the total implementation cost, as described by (1). (2) then guarantees that for every node n, we implement a node from each of its child e-classes. Lastly we introduce $\mathcal{S}$, the set of e-classes representing the desired expressions to implement in RTL. (3) then ensures that all these outputs are produced by the final RTL.

$$\text{minimize:} \sum_{n \in \mathcal{N}} \text{cost}(n)x_n \text{ subject to:} \qquad (1)$$

$$\forall (n, c) \in E. \ x_n \leq \sum_{n' \in \mathcal{N}_c} x_{n'} \qquad (2)$$

$$\forall c \in \mathcal{S}. \ \sum_{n \in \mathcal{N}_c} x_n = 1. \qquad (3)$$

E-graphs may contain cycles, *e.g.* $x + 0 \to x$ induces a cycle. By introducing a topological sorting variable, t_c for each e-class c, and associated constraints (4) where N is the number of e-classes and $\mathcal{C}(n)$ is the e-class containing node n, we ensure that the output expression is acyclic.

$$\forall (n, k) \in E \quad t_{\mathcal{C}(n)} - Nx_n - t_k \geq 1 - N \qquad (4)$$

If we select a node $n \in \mathcal{N}_c$ with child k, i.e. $x_n = 1$, this 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. We use the open source GLPK solver to calculate solutions to this ILP [21].

We deploy the ILP extraction method in cases where sharing common subexpressions offers some improvement, otherwise for improved performance we can resort to the standard `egg` extraction method [5].

IV. RESULTS

The RTL test cases are automatically optimized using our `egg`-based implementation and optimized RTL is extracted. Original and optimized RTLs are synthesized using Synopsys Design Compiler for a TSMC 7nm cell library. We proved the formal equivalence of the original and optimized RTLs using Synopsys HECTOR technology, a formal equivalence checking tool that runs in minutes on these testcases.

The original and optimized RTLs were synthesised at the minimum delay of the slowest of the two, and then similarly at the minimum area of the larger of the two designs. These represent comparisons towards the endpoints of a standard area-delay curve for a design. Table III summarises the results. In Figure 4, we present the area-delay profiles for the competing architectures of a Smoothing Kernel, which highlights the points of comparison used in Table III.

We consider two sets of examples. First we demonstrate how the tool exploits complex datapath blocks to optimize designs. Then we consider bitwidth-dependent architectures, where the optimal design may vary as a function of bitwidth.

A. Datapath Optimizations using Complex Blocks

The first benchmark is a kernel from a media module, which is an industrially relevant example supplied by Intel. It was manually optimized by a single engineer over the course of one week. Our tool automatically matches the results obtained via manual optimization, making use of the "Merge Mult Array" rewrite. Figure 4 shows the area-delay curves for the original and optimized architectures. The optimized design performs strictly better, reducing the minimum achievable delay by 13% with a 28% area reduction.

Benchmark	Min Achievable Delay			Min Achievable Area		
	Delay (ns)	Orig Area (μm^2)	Opt Area (μm^2)	Area (μm^2)	Orig Delay (ns)	Opt Delay (ns)
Smoothing Kernel	0.289	550	158 (**−71%**)	150	1.25	0.29 (**−77%**)
FIR Filter Kernel	0.611	1710	679 (**−60%**)	570	1.38	2.48 (**+80%**)
ADPCM Decoder [22]	0.102	103	102 (**− 1%**)	32	0.72	0.45 (**−38%**)
Shifted FMA	0.181	310	210 (**−32%**)	81	0.97	0.57 (**−41%**)
MCM Solution	0.132	90	104 (**+15%**)	40	0.72	0.56 (**−22%**)

Benchmark	Ops	E-graph Nodes	ILP	Runtime (sec)
Smoothing Kernel	17	27,000	No	140
FIR Filter Kernel †	23	550	Yes	100
ADPCM Decoder	9	6,700	No	19
Shifted FMA	3	22	No	0.04
MCM Solution	3	4,900	Yes	31

Fig. 5. Data-flow graph representing an optimal adder tree to compute the set $3 \times x, 7 \times x$ and $21 \times x$. The blue nodes represent additions which generate the result in the node label.
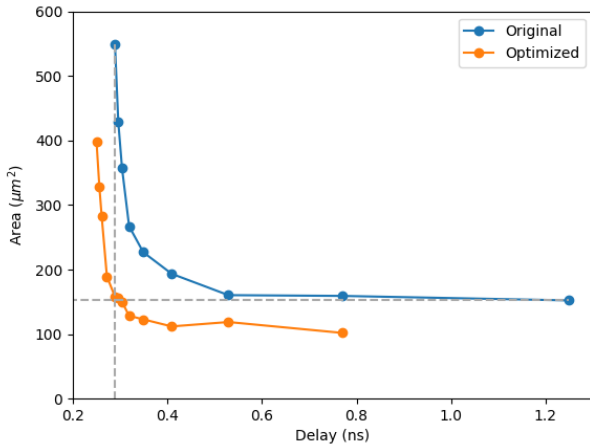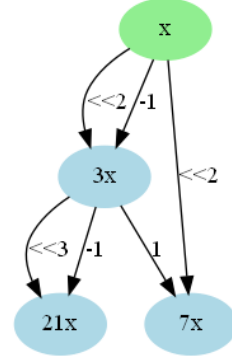

Fig. 4. Area-delay profile of the Smoothing Kernel. The horizontal/vertical grey lines represents the minimum area/delay comparison points.

Next we have a set of benchmarks taken directly from [4], which are intended to show how this prior work is generalized by our optimization tool. The first example is a finite-impulse response (FIR) filter with 8 taps. The second example, a computational kernel of the ADPCM decoder [22], is an approximation to a 16×4 multiplier. These two examples are optimized by deploying the "Arithmetic Logic Exchange" class of rewrites described in Section III-B, to cluster additions together. The FIR filter example is particularly interesting as it also introduces a multiple constant multiplication (MCM) problem [7]. Since the ILP formulation of extraction correctly accounts for the cost of common subexpressions, the minimal area solution it extracts maximally shares common subexpressions, to generate $\{2 \times x, 3 \times x, ..., 7 \times x\}$. For example, $2 \times x$

and $3 \times x$ are constructed as follows

$$x_2 = x << 1, \;\; x_3 = x_2 + x. \qquad (5)$$

Looking more generally at the MCM problem, using the "Constant Expansion" rewrites, we are able to match the operator count of a solution from [6]. An example from this paper generates adder graphs to compute $\{3 \times x, 7 \times x, 21 \times x\}$. The optimal design from our tool maximally shares common subexpressions to generate the data-flow graph shown in Figure 5, which uses 3 addition/subtraction operations to generate the results. Since logic synthesis likely implements constant multiplication using a CSD representation [8], sharing common subexpressions generates a 15% larger circuit than the basic architecture for small delay targets as the design does not match the performance of CSD.

Finally we consider a shifted FMA, which discovers the opportunity to merge a multiplication and addition that were originally separated by a left shift $(a \times b) << S + c$. Using the "Left Shift Mult" and "FMA Merge" rewrites this can be implemented as an FMA block. The approach proposed by Verma, Brisk and Ienne misses this opportunity since it will not move the shift over the multiplication [4].

B. Bitwidth Dependent Optimal Architecture

The last set of results considers parameterizable RTL, as in general, RTL engineers do not generate alternative architectures for different bitwidths. As a result, designers may be sacrificing performance. To demonstrate this we again consider the FIR filter, but the 4 tap variant, where the original

architecture is shown in Figure 6, which we will refer to as Architecture 0. We optimized Architecture 0 using our tool for increasing bitwidths and observed that the selected architecture varied between Architectures 0, 1 and 2 (Figure 6). The difference between Architectures 1 and 2 is that the addition involving $Z4$ in Architecture 1 has been pushed back over the right shift, incurring the cost of an extra left shift, but saving a full carry-propagate adder.

The architectural choices are determined by the theoretical two-input gate cost metric, discussed in Section III-C. We synthesized each of the three architectures for bitwidths $4, 8, ...64$, updating the delay target each time to the minimum delay that all three architectures could meet at that bitwidth. The theoretically optimal architecture for each bitwidth can be seen in Figure 6. For 56% of the bitwidths, the architecture selected by the theoretical cost metric gave the lowest area result from logic synthesis. The ability to always choose the minimum area design according to logic synthesis using the theoretical metric is limited, since logic synthesis always incorporates delay considerations into its results and there is noise in the results. We demonstrate this 'noise floor' in Section V.

C. Performance

For these test cases only the "Shifted FMA" e-graph saturated as we limited the exploration to 10 iterations of e-graph rewriting [5]. For test cases using ILP extraction we limited the solver to a 100 second time budget, which means that the FIR Filter solution is classified as feasible but is possibly suboptimal. There is a tradeoff between growing the e-graph and solving the ILP efficiently. Only the MCM solution included the "Constant Expansion" rewrite class since these rewrites lead to exponential growth of the e-graph.

V. Cost Metric Validation

This section evaluates the theoretical cost metric in terms of its ability to steer optimal architecture choices. To do so we compare the theoretical cost to logic synthesis area costs. The results from Table III demonstrate that for all test cases, the theoretically chosen architecture improved at least the performance or the area of the design.

When determining the accuracy of a cost estimate, it is necessary to consider inherent variability of the logic synthesis process. Small non-functional tweaks, *e.g.* changing a variable name in RTL code, can have impact on the synthesis results. This forms a 'noise floor' against which any theoretical cost model can be validated. To evaluate this noise floor, we used a technique known as fuzzing [23], which involves automatically generating random mutations to a program. We fuzz the RTL allowing two types of semantics-preserving mutations: variable renaming and swapping the order of always/assign blocks [19] in the code, modifications which one would not expect to have a meaningful impact on synthesis results.

We provide results for the Smoothing Kernel and 4 tap FIR Filter, synthesizing 30 fuzzed designs in each case at relevant delay targets. Variability of the results is shown in Figure 7.
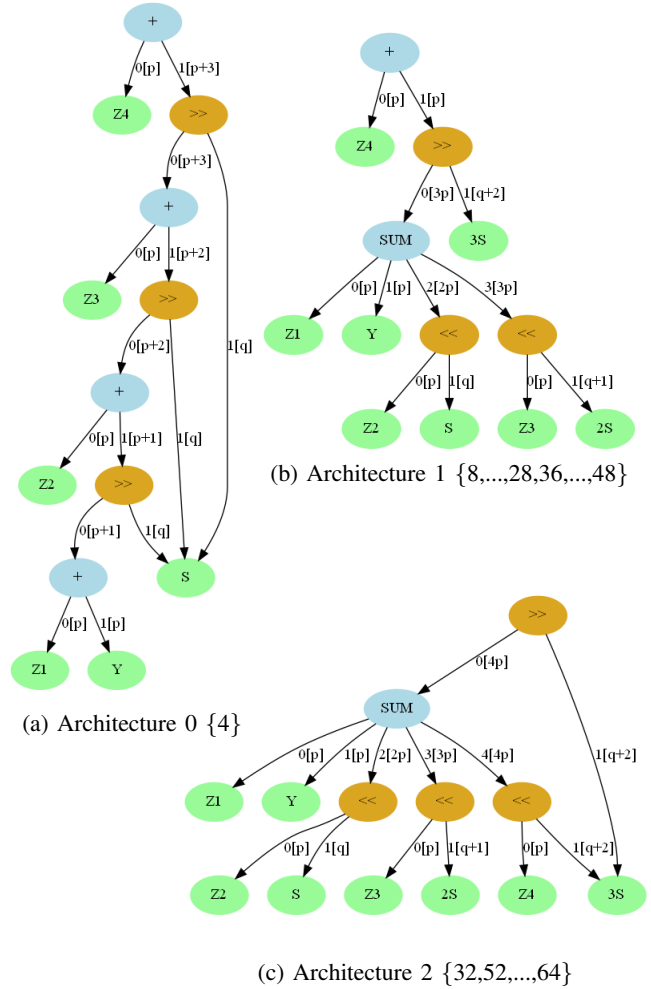

(a) Architecture 0 {4}

(b) Architecture 1 {8,...,28,36,...,48}

(c) Architecture 2 {32,52,...,64}

Fig. 6. Simplified FIR filter data-flow graphs representing optimal architectures for different choices of the input bitwidth parameter p and shift bitwidth parameter q. The sets in curly braces are bitwidths for which that architecture is optimal. In these graphs $Zi = Xi \times Ai$ and $2S$ and $3S$ are computed according to (5).
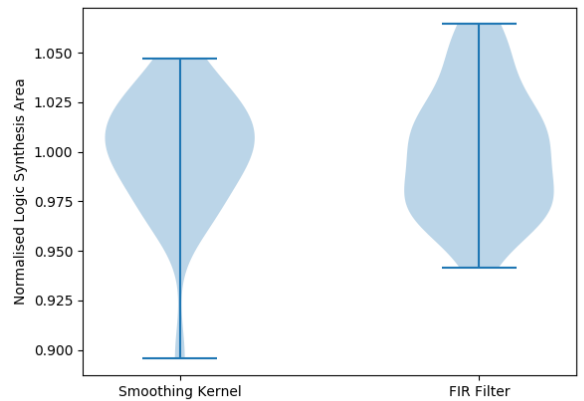

Fig. 7. A violin plot depicting the logic synthesis area results for 30 fuzzed designs of the Smoothing Kernel and the FIR Filter at a 0.5ns delay target. For each violin, the area results are normalised by the mean.
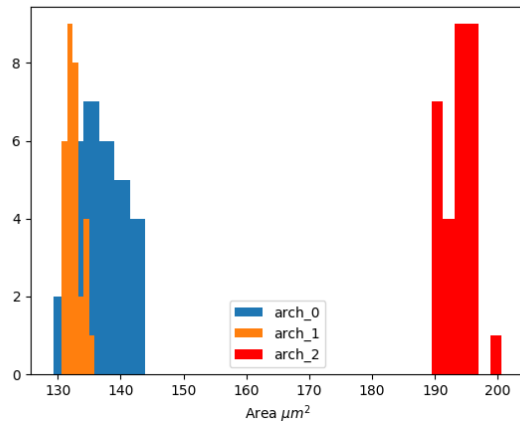
Fig. 8. Histogram plot of logic synthesis area results for 30 fuzzed designs for each of the three FIR Filter architectures for 12 bits (Fig 6).

Figure 8 highlights how this noise can affect the architectural choices made in Section IV-B. For 12 bit inputs the synthesis results for fuzzed Architectures 0 and 1 overlap, with Architecture 1 generating lower area results on average whilst Architecture 0 obtains the minimum area result. This noise is not captured by the theoretical cost metric, as these fuzzed designs are theoretically identical. Applying this to other bitwidth inputs there are cases where there is clearly an optimal choice.

VI. Conclusion

This paper has demonstrated the application of e-graphs and equality saturation to the RTL datapath optimization problem. Applying a precisely defined set of bitwidth dependent equivalence preserving transformations, in the form of rewrites, we efficiently explore the design space and extract optimized RTL using egg. We also quantify the noise floor in logic synthesis results to understand the limits of a theoretical cost metric.

The results show that this automated rewriting technique can match the results of a skilled hardware engineer within a short timescale. The tool was able to achieve up to 71% area improvement and up to 77% delay improvement. We also demonstrate that automatic RTL optimization can generate different architectures for different bitwidth designs, since the tradeoff points are bitwidth dependent.

Future work will address the limitations of an area-only cost metric by incorporating delay and power allowing us to generate Pareto optimal solution curves [24]. We will expand the domain to tackle floating point operations and incorporate greater support for automated design verification. We will also address the scalability limits of applying equality saturation by incorporating intelligent design space search procedures.

Acknowledgment

References

[1] Synopsys, "Coding Guidelines for Datapath Synthesis," Synopsys, Mountain View, Tech. Rep., 12 2019.
[2] ——, "Design Compiler User Guide S-2021.06-SP2," Synopsys, Mountain View, Tech. Rep., 6 2021.
[3] A. K. Verma, P. Brisk, and P. Ienne, "Challenges in automatic optimization of arithmetic circuits," in *Proceedings - IEEE Symposium on Computer Arithmetic*, 2009, pp. 213–218.
[4] ——, "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.
[5] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, 2021.
[6] 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.
[7] R. I. Hartley, "Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers," *IEEE Transactions on Circuits and Systems*, vol. 11, 1996.
[8] M. D. Ercegovac and T. Lang, *Digital arithmetic*. Elsevier, 2004.
[9] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.
[10] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.
[11] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *ACM SIGPLAN Notices*, vol. 44, no. 1, 2009.
[12] 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.
[13] G. H. Smith, A. Liu, S. Lyubomirsky, S. Davidson, J. McMahan, M. Taylor, L. Ceze, and Z. Tatlock, "Pure tensor program rewriting via access patterns (representation pearl)," in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, 2021.
[14] 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.
[15] C. Wolf and J. Glaser, "Yosys-A Free Verilog Synthesis Suite," in *Proceedings of Austrochip*, 2013.
[16] G. Steele, *Common LISP: the language*. Elsevier, 1990.
[17] A. Beaumont-Smith and C.-C. Lim, "Parallel prefix adder design," in *Proceedings - IEEE Symposium on Computer Arithmetic*, 2001, pp. 218–225.
[18] I. Koren, *Computer arithmetic algorithms*. AK Peters/CRC Press, 2018.
[19] D. Thomas and P. Moorby, *The Verilog® hardware description language*. Springer Science & Business Media, 2008.
[20] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016.
[21] A. Makhorin, "GLPK (GNU linear programming kit)," 2008. [Online]. Available: http://www.gnu.org/s/glpk/glpk.html
[22] 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.
[23] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[24] 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.