Combining Power and Arithmetic Optimization via Datapath Rewriting

Samuel Coward[1,2], Theo Drane[1], Emiliano Morini[1] and George A. Constantinides[2]
[1] *Intel Corporation,* [2] *Imperial College London,*
Email: {samuel.coward, theo.drane, emiliano.morini}@intel.com, g.constantinides@imperial.ac.uk

***Abstract*—Industrial datapath designers consider dynamic power consumption to be a key metric. Arithmetic circuits contribute a major component of total chip power consumption and are therefore a common target for power optimization. While arithmetic circuit area and dynamic power consumption are often correlated, there is also a tradeoff to consider, as additional gates can be added to explicitly reduce arithmetic circuit activity and hence reduce power consumption. In this work, we consider two forms of power optimization and their interaction: circuit area reduction via arithmetic optimization, and the elimination of redundant computations using both data and clock gating. By encoding both these classes of optimization as local rewrites of expressions, our tool flow can simultaneously explore them, uncovering new opportunities for power saving through arithmetic rewrites using the e-graph data structure. Since power consumption is highly dependent upon the workload performed by the circuit, our tool flow facilitates a data dependent design paradigm, where an implementation is automatically tailored to particular contexts of data activity. We develop an automated RTL to RTL optimization framework, ROVER, that takes circuit input stimuli and generates power-efficient architectures. We evaluate the effectiveness on both open-source arithmetic benchmarks and benchmarks derived from Intel production examples. The tool is able to reduce the total power consumption by up to 33.9%.**

1. Introduction

The three mostly common circuit quality metrics used in digital hardware design are power, performance and area, abbreviated to PPA. The performance of a circuit refers to how fast the circuit can execute the specified computation, the area is a measurement of how much silicon the circuit occupies on a die, which is highly correlated with manufacturing cost. Lastly, power is a measurement of the energy per unit time used to perform a given computation. The rise of custom accelerators presents the opportunity to optimize arithmetic hardware designs for particular computations, allowing us to perform those computations using less energy.

Whilst performance and area are relatively simple to estimate, power can only be estimated accurately knowing the data values being operated upon, e.g. via representative workloads. The majority of power estimation tools in the electronic design automation (EDA) industry are based on randomly generated or real-world simulation stimuli [1], [2]. In this work, we focus on dynamic power consumption that is influenced heavily by register transfer level (RTL) design. From a simulation of a circuit design it is possible to infer bit-level switching activity, that is the frequency of transitions of a bit from zero to one or one to zero. These switching activities can be translated into power consumption estimates via a power model.

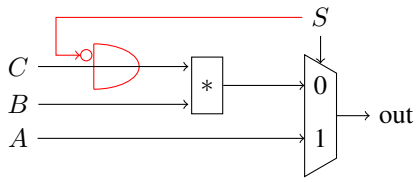


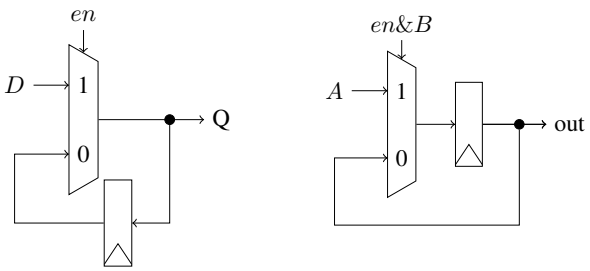
Figure 1. An operand isolation opportunity. In the original circuit (black), the input to the multiplier can be data gated when the select signal is one, as shown by the red gate. The negated select signal, $\overline{S}$ is a common input to an array of AND gates equal to the bitwidth of C.

Recently, Coward *et al.* developed an RTL rewriting framework to address both area [3] and performance [4]. The core of their approach is an e(quivalence)-graph representation of RTL that facilitates exploration of these two metrics. In this work, we leverage this RTL rewriting framework and complete the PPA axes. We encode power optimizations such as clock gating and operand isolation via a set of local rewrites that, when combined with arithmetic rewrites, facilitate the exploration of new design spaces and the discovery of novel power efficient designs. We also exploit the compact e-graph representation to develop a computationally efficient power model that enables data-dependent circuit design. We develop a tool, ROVER, which can customize an implementation based on representative workloads.

In Section 2 we provide background on RTL power analysis and optimization along with an introduction to e-graphs. In Section 3 we describe how ROVER encodes power optimizations in terms of RTL rewrites. We also describe ROVER's power model and how the compact representation provided by the e-graph yields efficiency gains for RTL simulation. Lastly, in Section 4 we demonstrate ROVER's impact on power consumption on a set of benchmarks.

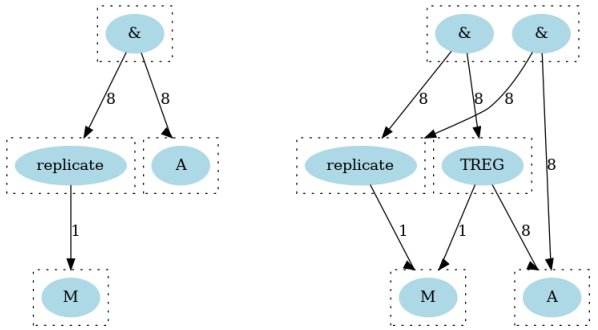
The paper contains the following novel contributions:

- a set of local equivalence preserving RTL rewrites that capture power-specific optimizations,
- an encoding of clock gating and operand isolation that goes beyond current mux tree analysis,



(a) A transparent register. (b) An enabled register.

Figure 2. Circuit diagrams of the TREG and REG operators.



(a) Initial e-graph contains $A \,\&\, \{8\{M\}\}$ (b) Apply rewrite $A\&\{8\{M\}\} \rightarrow \texttt{TREG}(A, M)\&\{8\{M\}\}$

Figure 3. E-graph rewriting of a masking operation. Dashed boxes represent e-classes of equivalent expressions. A new equivalent expression is added to the e-graph represented by the second & operator in the root e-class.

- a computationally-efficient methodology to simulate a large set of design choices, leveraging the compact e-graph representation,
- an automated method for data-driven design.

2. Background

2.1. RTL Power Optimization and Analysis

Power optimizations can be broadly separated into two groups. First, a set of optimizations that primarily target circuit area reduction, since there is a correlation between circuit area and dynamic power consumption. This is intuitive because a smaller circuit area corresponds to fewer gates and thus fewer gates to toggle. Several previous works from both academia [5], [6], [7] and industry [8] have explored datapath RTL area reduction, including one work that used e-graph rewriting [3].

The second set of optimizations, which are the primary focus of this work, detects opportunities to switch off, or gate, sub-circuits in the design. Clock gating and operand isolation are two such optimizations. For a clock gating example, consider a pipelined floating-point adder in which exception cases, *e.g.* NaNs, are handled on a separate exception path. If we detect an exceptional input in the first stage, we can gate all registers on the standard input path for subsequent stages, since the result is redundant. Gating the registers stops the register outputs changing and hence prevents any toggling of the downstream combinational logic. The additional gating logic adds an area (and possible delay) overhead which must be evaluated alongside the data-dependent power saving. For an operand isolation example, consider Figure 1. In this circuit we can identify a redundant operation and construct an activation signal that we use to zero one of the multiplier inputs, limiting operator power consumption. We refer to this technique as data gating. Alternatively, both multiplier inputs could be "frozen" using transparent registers to eliminate redundant toggling [9], [10]. The transparent register shown in Figure 2a has an enable signal, which, when high, allows the input to transparently flow through to the output and, when low, freezes the output. Prior work has called this circuit a transparent latch, but since we operate in a synchronous domain we shall instead call it a transparent register.

In academia, clock gating has been explored at a gate-level [11] and from a clock tree synthesis perspective [12]. A subset of industrial tools, such as Synopsys Power Compiler [1] and Cadence Joules [2], are incorporated into the logic synthesis engines and automatically perform clock-gating optimizations. Siemens PowerPro [13] is a standalone RTL to RTL tool that targets sequential clock gating. A limitation of these approaches is that they rely on analyzing the mux tree structure of the RTL design, but this may miss opportunities as we shall see in Section 3.2. The automation of operand isolation has been explored at both the word-level [9] and at gate-level [10], [14]. An e-graph based rewriting approach allows us to compare and combine these different power optimization techniques.

RTL power analysis tools typically rely on simulation to estimate power consumption of a given design. Tool users can either provide simulation stimuli or set input switching activities and static probabilities [1], [2]. For a given simulation period, the switching activity describes how frequently each bit of the given signal transitions from zero to one or vice versa, and the static probability specifies what proportion of the time that bit is expected to be in the one state. Commercial logic synthesis tools [15], take user provided simulation configurations and perform power optimizations guided by the simulation.

This paper describes how to encode the power optimizations discussed above as local RTL rewrites and a framework to explore the combination of power, arithmetic and area optimizations. Our tool encodes arithmetic optimizations performed by downstream synthesis tools in the optimization flow, a key factor not considered by prior work. For analysis the underlying data structure of our tool also enables efficient evaluation of the power consumption of equivalent design candidates.

2.2. E-Graphs

An e-graph is typically initialized with a single expression, such that every e-class contains a single node. The e-graph is grown via constructive rewrite application, where

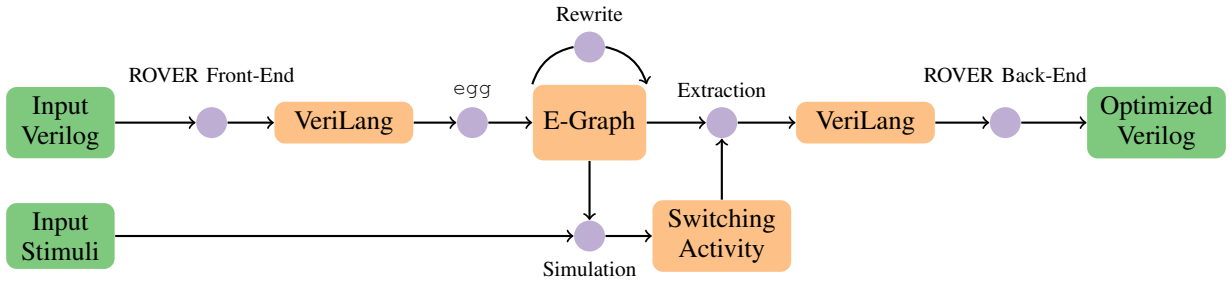


Figure 4. ROVER's power optimization tool flow. Users provide an input Verilog design and input stimuli via simulation data or switching activity statistics.

a pattern, l, is matched in the e-graph and gets rewritten to a different expression, r. The e-graph retains the original expression l in the data structure, r is simply added to the matched e-class. Figure 3 shows an example e-graph before and after applying a rewrite. Given a set of rewrites, all rewrites are applied to the e-graph at each rewriting iteration. Determining an optimal sequence of rewrite applications is deferred to the extraction phase. We grow an e-graph until we reach a computational limit or we reach a state called saturation, where further rewriting adds no new nodes to the data structure. The final e-graph represents a set of equivalent implementations. The process to select the 'best' implementation is known as extraction and is typically based on a custom cost model for each application.

Coward, *et al.* introduced an intermediate language, VeriLang, that can represent combinational RTL in an e-graph [3], [4]. We adopt the same representation here, but extend it to incorporate registers, allowing us to represent pipelined designs in the e-graph. Earlier work by the same authors developed an RTL optimization tool using mixed precision RTL rewriting to reduce circuit area and delay. We build on their framework as a foundation, extending it to target power optimization. Our key insight is that their delay optimization work [4], introduced `ASSUME` nodes to exploit the mux tree structure of a design in an e-graph rewriting framework. The `ASSUME` node associates code branches with *observability don't care* conditions and allow the e-graph to capture equivalence under some constraint, which they show to be useful in identifying logic to accelerate special cases, e.g. near/far path floating point addition. As noted above, *observability don't cares* are also – for other reasons – of central relevance to power-saving optimizations in RTL datapath design, and we describe in this paper how to exploit that commonality.

Recently, a general purpose and extensible e-graph library, `egg`, was released [16]. `egg` provides a number of innovations in performance and e-graph features that has led to applications in numerical stability improvement [17] and FPGA multiplier design [7].

3. Methodology

In Figure 4 we provide an overview of how ROVER optimizes a design to reduce power consumption. Using the ROVER front-end, an input Verilog design is converted to VeriLang, which `egg` uses to initialize an e-graph [3]. ROVER then applies a set of power optimization rewrites, described in Sections 3.1 and 3.2, to the e-graph, constructing a set of implementation candidates. To accurately model per implementation power consumption, ROVER simulates the entire e-graph based on user configured input stimuli as described in Section 3.3. The user configured input stimuli provide, for every module input, a sequence of bitvectors that are fed one per clock cycle. Throughout this work, we assume a single clock domain, for simplicity. E-graph simulation provides switching activities for all the internal signals of all the candidates, which are fed into the power model, described in Section 3.4. The power model is used by ROVER to determine the optimal implementation, producing a VeriLang expression. The ROVER back-end converts the extracted VeriLang expression into Verilog.

In the original paper where VeriLang was introduced [3], the authors envisaged its semantics as operating over Boolean values. In this work, we modify the semantics of VeriLang to consider input variables as *streams* of Boolean data, such that a new data point enters the module every clock cycle. Since we consider streams of data, every intermediate signal created in the e-graph has an associated stream. The semantics of combinational operators are such that each clock cycle the new data points are used to generate a new output within the same cycle.

We incorporate a new VeriLang operator, `REG`, which describes a register with an enable signal. The corresponding circuit is shown in Figure 2b. Given input a and enable signal en with associated data streams a_i and en_i, $\texttt{REG}(a, en)$ has the following semantics:

$$\texttt{REG}(a_i, en_i) = \begin{cases} 0 & \text{, if } i == 0 \\ a_{i-1} & \text{, if } en_{i-1} \\ \texttt{REG}(a_{i-1}, en_{i-1}) & \text{, else.} \end{cases} \quad (1)$$

These semantics assume that a register is initialized to zero.

3.1. Data Gating

In this section we describe a set of rewrites that encodes the operand isolation optimizations described in Section 2.1. A key challenge in expressing operand isolation via local rewrites, is that having identified a redundant computation from a functional perspective we do not care what value is

TABLE 1. A SET OF RTL REWRITES ENCODING OPERAND ISOLATION AND CLOCK GATING OPTIMIZATIONS. WE DEFINE FOUR SETS OF OPERATORS SUCH THAT $\texttt{OP}$ IS ANY ARITHMETIC OR LOGICAL VERILANG OPERATOR, $\texttt{OP1} \in \{*, \ll, \gg, +, -\}$, $\texttt{OP2} \in \texttt{OP1} \setminus \{+, -\}$ AND $\texttt{OP3}$ IS ANY BOOLEAN OPERATOR. WE USE w_a TO DENOTE THE BITWIDTH OF A BITVECTOR a, w_o TO DENOTE THE OUTPUT BITWIDTH OF AN OPERATION.

Group	Name	Left-Hand Side	Right-Hand Side
Data Gate	Gate Left	$s\,?\,b\,:\,c$	$s\,?\,(b\,\&\,\{w_b\{s\}\})\,:\,c$
	Gate Right	$s\,?\,b\,:\,c$	$s\,?\,b\,:\,(c\,\&\,\{w_c\{\overline{s}\}\})$
	Propagate Mask	$(a\ \texttt{op1}\ b)\,\&\,\{w_o\{s\}\}$	$(a\,\&\,\{w_a\{s\}\})\ \texttt{op1}\ (b\,\&\,\{w_b\{s\}\})$
	Propagate Mask Left	$(a\ \texttt{op2}\ b)\,\&\,\{w_o\{s\}\}$	$(a\,\&\,\{w_a\{s\}\})\ \texttt{op2}\ b$
	Propagate Mux Mask	$(s_1\,?\,a\,:\,b)\,\&\,\{w_o\{s_2\}\}$	$s_1\,?\,(a\,\&\,\{w_a\{s_2\}\})\,:\,(b\,\&\,\{w_b\{s_2\}\})$
	Propagate Mux Mask Right	$(s_1\,?\,a\,:\,b)\,\&\,\{w_o\{s_2\}\}$	$s_1 \& s_2\,?\,a\,:\,(b\,\&\,\{w_b\{s_2\}\})$
	Propagate Mux Mask Left	$(s_1\,?\,a\,:\,b)\,\&\,\{w_o\{s_2\}\}$	$s_1 \,\|\, \overline{s_2}\,?\,(a\,\&\,\{w_a\{s_2\}\})\,:\,b$
	Combine Masks	$(a\,\&\,\{w_a\{s_1\}\})\,\&\,\{w_a\{s_2\}\}$	$a\,\&\,\{w_a\{s_1 \& s_2\}\}$
Transparent Registers	Transp Reg Left	$s\,?\,b\,:\,c$	$s\,?\,\texttt{TREG}(b, s)\,:\,c$
	Transp Reg Right	$s\,?\,b\,:\,c$	$s\,?\,b\,:\,\texttt{TREG}(c, \overline{s})$
	Transp Reg Mask	$a\,\&\,\{w_a\{s\}\}$	$\texttt{TREG}(a, s)\,\&\,\{w_a\{s\}\}$
	Transp Reg Saturate	$a\,\|\,\{w_a\{s\}\}$	$\texttt{TREG}(a, \overline{s})\,\|\,\{w_a\{s\}\}$
	Transp Reg Reg	$\texttt{REG}(a, en)$	$\texttt{REG}\,(\texttt{TREG}(a, en), en)$
	Propagate	$\texttt{TREG}(a\ \texttt{op}\ b, s)$	$\texttt{TREG}(a, s)\ \texttt{op}\ \texttt{TREG}(b, s)$
	Propagate Mux	$\texttt{TREG}(s_1\,?\,a\,:\,b, s_2)$	$\texttt{TREG}(s_1, s_2)\,?\,\texttt{TREG}(a, s_2)\,:\,\texttt{TREG}(b, s_2)$
	Combine Transp Reg	$\texttt{TREG}(\texttt{TREG}(a, s_1), s_2)$	$\texttt{TREG}(a, s_1\,\&\,s_2)$
Clock Gate & Retime	Retime Boolean	$\texttt{REG}(a, en)\ \ \texttt{op3}\ \ \texttt{REG}(b, en)$	$\texttt{REG}(a\ \ \texttt{op3}\ \ b, en)$
	Clock Gate Reg	$\texttt{TREG}(\texttt{REG}(a, en), \texttt{REG}(b, en))$	$\texttt{REG}(a, en\,\&\,b)$

produced under certain conditions. Therefore, from a functional perspective, when we do not care, we can generate any value we choose. However, the values that we select have a significant impact upon power consumption.

Before progressing, we define notation used throughout. We let w_x denote the bitwidth of a bitvector variable x and let w_o denote the output bitwidth of an operation. We use $\{w\{S\}\}$ to denote w-fold replication of a bitvector S, which will usually be a single bit. Lastly, we use $\overline{S}$ to denote the bitwise logical complement of a bitvector S.

In one approach to perform operand isolation we can apply data gating to each branch of a mux operator. Data gating creates a mask by duplicating a select signal and applies a bitwise AND operation. This rewrite explicitly zeroes redundant outputs. The first group in Table 1 contains rewrites to create initial data gating operations. We include two "Gate" rewrites as we may wish to data gate only the true branch, only the false branch or both, by applying "Gate Left" and "Gate Right" in sequence. The rewrite from (2) to (3) illustrates the creation of a mask and data gating of a mux branch, in order to avoid dynamic power in the multiplier. Table 1 next describes how the data gating operations are propagated over arithmetic operations, since these operators typically account for the largest power consumption in datapath circuits. The "Propagate" rewrites incrementally gate larger sub-circuits. For a subset of operators, *e.g.* multiplication, it is equivalent to data gate a single operand, as illustrated in (3) and (4). The rewrites in Table 1 encode the optimization shown in Figure 1.

$$S\ ?\ A\ :\ (C * B) \qquad \rightarrow \textit{ (Gate Right)} \tag{2}$$

$$S\ ?\ A\ :\ (C * B)\,\&\,\{w_o\{\overline{S}\}\} \rightarrow \textit{(Propagate Left)} \tag{3}$$

$$S\ ?\ A\ :\ (C\,\&\,\{w_c\{\overline{S}\}\}) * B \tag{4}$$

The impact of data gating redundant operations depends on the wider module context. Exploring data gating via e-graph rewriting allows ROVER to retain a set of gated and ungated designs, deferring architecture selection and evaluation to the extraction phase. For example, in

$$(s\,?\,f(a)\ :\ b) + g(f(a)), \tag{5}$$

the computation of $f(a)$ appears redundant when s is zero, however we always use $f(a)$ in the computation of $g(f(a))$, thus there is no value in a gated version.

Applying these rewrites to a nested mux structure, ROVER naturally generates nested gating operations which are combined via classical Boolean rewriting. Such an approach constructs *observability don't care* conditions that are not present in the original design. These newly created conditions can be simplified using Boolean rewriting.

In addition to the rewrites described in Table 1, ROVER deploys the arithmetic and area optimization rewrites described in [3], that crucially encode downstream logic synthesis optimizations. ROVER also includes standard Boolean rewrites for optimizing logical expressions. Exploring these transformations in parallel, ROVER discovers architectures providing an efficient area power tradeoff.

3.2. Clock Gating

In the previous section, we described how data gating rewrites can encode operand isolation. In this section, we shall describe a set of local equivalence preserving rewrites that create transparent registers providing an alternative way to achieve operand isolation. In (1) we defined the semantics of REG. We define an additional VeriLang operator, TREG, representing a transparent register, shown in Figure 2a, with semantics that are similar to those of REG.

$$\texttt{TREG}(a_i, b_i) = \begin{cases} a_i & \text{, if } b_i \\ \texttt{TREG}(a_{i-1}, b_{i-1}) & \text{, elif } i > 0 \\ 0 & \text{, else} \end{cases}$$

The second group in Table 1 contains a set of rewrites, similar to the first group, that encode the creation and propagation of TREG operators. We improve upon approaches based on mux tree analysis by including the "Transp Reg Mask/Saturate" rewrites that detect redundant computation, as used in Figure 3. We also create transparent registers from register enable signals, since disabled registers correspond to redundant computation. The "Combine Transp Reg" rewrite allows ROVER to construct complex *observability don't care* signals that may not be present in the initial design. These signals may be simplified via Boolean rewriting.

In the final group in Table 1, we describe how ROVER encodes clock gating via local rewrites. When the TREG operator meets the output of a register, it represents an opportunity to refine the enable condition of the register, eliminating the overhead of the transparent register. We can prove the equivalence of

$$L_i = \texttt{TREG}(\texttt{REG}(a_i, en_i), \texttt{REG}(b_i, en_i)) \text{ and}$$
$$R_i = \texttt{REG}(a_i, en_i \& b_i)$$

for all clock cycles i via induction. First, let

$$p_i = \texttt{REG}(a_i, en_i) \text{ and } q_i = \texttt{REG}(b_i, en_i).$$

Suppose $\forall i \leq k\ L_i = R_i$, then if $en_k = 1$:

$$\begin{aligned} q_{k+1} &= b_k \qquad p_{k+1} = a_k \\ L_{k+1} &= q_{k+1}\,?\,p_{k+1}\ :\ L_k \\ &= b_k \quad ?\,a_k \quad :\ L_k \end{aligned}$$

Then, since $en_k = 1$ and $R_k = L_k$,

$$\begin{aligned} R_{k+1} &= en_k \& b_k\,?\,a_k\ :\ R_k \\ &= \quad b_k\,?\,a_k\ :\ R_k = L_{k+1} \end{aligned}$$

Now if $en_k = 0$, then $R_{k+1} = R_k = L_k$ and

$$\begin{aligned} q_{k+1} &= q_k \qquad p_{k+1} = p_k \\ L_{k+1} &= q_k\,?\,p_k\ :\ L_k \\ q_k = 1 \Rightarrow L_k &= q_k\,?\,p_k\ :\ L_{k-1} = p_k \end{aligned}$$

Therefore $L_{k+1} = L_k = R_{k+1}$ and hence $R_{k+1} = L_{k+1}$ for all values of en_k. Under the zero register initialization assumption it is trivial to prove $L_0 = R_0$.

A key requirement of the "Clock Gate Reg" rewrite, is that the *observability don't care* condition be available in the previous clock cycle. This constraint ensures that the register is disabled for the clock cycle corresponding to the redundant computation. In certain cases, it may be necessary to move operations into earlier clock cycles to ensure the gating signal is available in the correct cycle. To transfer operations between clock cycles we implement limited retiming of Boolean operators.

As described in Figure 4, ROVER applies all rewrites described to grow an e-graph of equivalent implementations until a user defined limit or saturation is reached. The final e-graph contains designs with different combinations of gating and arithmetic optimizations. Determining which combination of optimizations produce the most power efficient design is left to extraction, which we describe next.

3.3. Simulation

In order to analyze power consumption, ROVER must first simulate all designs within the e-graph based on a set of stimuli. ROVER takes an additional input configuration file that provides simulation stimuli or sets the switching activity for each module input. If the user only defines a switching activity and simulation length, ROVER automatically generates simulation stimuli for all module inputs using an algorithm that randomly toggles each bit in a bitvector according to the configured toggle rate.

Since all nodes in a given e-class are functionally equivalent, ROVER simulates one node per e-class to obtain simulation data for the entire class. This observation provides a significant computational efficiency gain, as the complexity of simulating all designs in the e-graph scales with the number of e-classes. Meanwhile, the number of distinct designs contained in the e-graph can grow exponentially with the number of classes [7], as shown for the example of Figure 1 in Figure 5. In this example, the number of e-classes grows by a factor of four whilst the number of designs grows by a factor of 1000. The number of e-classes in Figure 5 does not grow monotonically, since in later rewriting iterations e-classes get merged due to proof of equivalence generated by ROVER reducing the number of classes. Note that, whilst a single node evaluation can be shared across the e-class, each node in the e-class may require more or less power to produce that same value. For example, $x + x$ and $x \ll 1$ are functionally equivalent but may consume significantly different power. It is this difference our extraction is designed to estimate and exploit.

From the e-class simulation data, ROVER calculates an average switching activity across the entire output word of that e-class. For example, for a 3-bit word with switching activities of 0.25 for bit 0, 0.5 for bit 1 and 0.75 for bit 2 would average to 0.5 across the entire word.

3.4. Operator Power Model

The purpose of the power model is to order the candidate implementations so that ROVER can select the most power

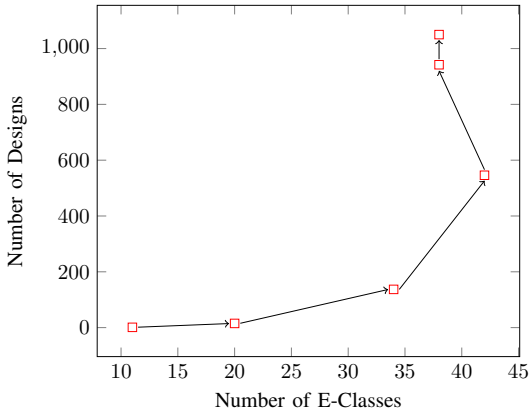


Figure 5. The number of designs vs. the number of e-classes after each iteration of rewriting the design in Figure 1. Simulation complexity scales with the number of e-classes but evaluates all designs in the e-graph.

efficient design. The e-graph simulation provides us with an average word-level switching activity for each e-class.

It is a known challenge to accurately estimate operator power consumption based on a word-level RTL implementation as it is highly dependent on downstream transformations and library selection [18]. ROVER mitigates this by encoding certain high-level datapath optimizations, such as arithmetic clustering [8], in the e-graph [3]. In this work, we combine the existing theoretical circuit area model from [3] with the simulated switching activity statistics to estimate the number of two-input gates toggling per clock cycle. For each node n in a given e-class c with child e-classes $c_0, \cdots, c_{k-1}$, we compute a power estimate, $P(n)$.

$$P(n) = A(n) \times \frac{1}{k+1} \left(T_c + \sum_{i=0}^{k-1} T_{c_i} \right) \tag{6}$$

where, $A(n)$ is an estimate of the number of two-input gates required to synthesize that operator, and T_c and T_{c_i} are the operator's output and input toggle frequencies, respectively. For arithmetic operators the area model fixes an architecture, based on known logic synthesis implementations, following the methodology described in [3]. The area model accounts for operand bitwidths and constant operands. The power model assigns an equal weight to input and output switching activities to approximate the proportion of the gates which transition each cycle. We do not model wire power consumption. In Section 4 we evaluate how accurately ROVER's model is able to estimate power consumption.

As in previous works on RTL optimization using e-graphs, to correctly account for common sub-expressions, we formulate extraction as an integer linear programming (ILP) problem [3]. Leting N denote the set of e-graph nodes, we use Boolean variables x_n to encode whether we select a particular node n and minimize,

$$\sum_{n \in N} x_n \times P(n). \tag{7}$$

Additional constraints ensure that we extract a valid implementation computing all the module outputs [3].

4. Results

To evaluate ROVER's impact on dynamic power consumption, we gathered two sets of benchmarks as shown in Table 2. The first set of three benchmarks are provided by Intel. The "Combinational Mux Add Tree", is taken from Intel low power training materials and comprises of three adders and three muxes. The example demonstrates how the dataflow graph can be rearranged to move particularly high toggling signals towards the outputs, reducing toggling in the rearranged circuit. The second benchmark, "Address Generation", is a snippet from production code, which is used as an example of how to perform power optimization in the training materials. It is comprised of two adders, a multiplier and a pair of muxes. The third benchmark, "Weight Calculation", is a production two-stage pipelined design computing pixel offsets in the graphics pipeline.

The second set of benchmarks are taken from prior publications [4], [9], [10], [13]. The "Pipelined Mux Add Tree" [9] is similar to the "Combinational Mux Add Tree" but introduces a distinct pipelined structure. It is comprised of two adders, three muxes and a pair of registers. The "Dual Path ALU" design [10] can optionally perform either a shift or addition. Next, the "Sequential Reg" benchmark is used in the PowerPro white paper to demonstrate the tool's sequential clock gating capabilities. It is a combination of registers and a mux. The "Dual Path FP Sub" is a pipelined floating point subtractor with a near/far path split [4].

For each design, we pass ROVER the original System Verilog design, which does not contain any existing power optimizations, along with a json file that specifies the input switching activities. We run ROVER twice generating an area optimized and a power optimized design in System Verilog and an estimate of the power reduction according to ROVER's power model. Using a commercial logic synthesis tool targeting a TSMC 5nm cell library, we synthesize the original and ROVER generated designs at a range of delay targets to mitigate the impact of logic synthesis noise [3]. We provide the commercial tool with the same switching activity configuration as given to ROVER. In Table 2, we show the average circuit area and average total power consumption (including leakage power) reported by the synthesis tool across the range of delay targets. The commercial synthesis tool incorporates a power analysis and optimization tool, which provides relevant power estimates based on the switching activities configured. To ensure the correctness, we verify the cycle-accurate equivalence of the original and ROVER generated designs using a commercial formal equivalence checking tool.

4.1. Dynamic Power Reduction

Table 2 compares total power and area results for each of the benchmarks before and after ROVER optimization. ROVER reduces total power consumption by up to

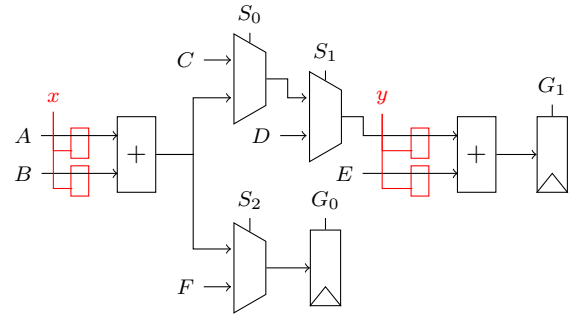
TABLE 2. LOGIC SYNTHESIS RESULTS COMPARING THE AVERAGE TOTAL POWER CONSUMPTION AND AVERAGE AREA ACROSS SEVERAL DELAY TARGETS. WE COMPARE THE BASELINE, AGAINST TWO IMPLEMENTATIONS GENERATED BY ROVER, ONE TARGETING AREA OPTIMIZATION AND ONE TARGETING POWER OPTIMIZATION. WE BOLD THE BEST RESULT FOR EACH METRIC. WE REPORT THE RELATIVE CHANGE VS THE BASELINE AND HIGHLIGHT OUR NEW CONTRIBUTION IN BLUE. WE INCLUDE THE NUMBER OF NODES IN THE INITIAL E-GRAPH FOR EACH BENCHMARK.

Benchmark	Nodes	Baseline		Area Optimized		Power Optimized	
		Area (μm^2)	Power (μW)	Area (μm^2)	Power (μW)	Area (μm^2)	Power (μW)
Comb. Mux Add Tree	20	32.9	98.2	32.8 (- 0.4%)	98.8 (+ 0.5%)	**31.0** (- 7.4%)	**83.2** (-15.5%)
Address Generation	22	58.5	421.9	**57.1** (- 0.2%)	419.2 (-0.6%)	57.2 (+ 2.2%)	**301.2** (-28.7%)
Weight Calculation	81	51.6	1141.4	**46.4** (-10.2%)	1072.3 (-6.1%)	53.3 (+ 3.2%)	**871.5** (-23.7%)
Pipe. Mux Add Tree [9]	23	**38.6**	852.3	**38.6** (0.0%)	852.3 (0.0%)	44.1 (+14.9%)	**615.3** (-27.2%)
Dual Op ALU [10]	17	**6.5**	186.9	**6.5** (0.0%)	186.9 (0.0%)	7.5 (+15.1%)	**146.8** (-21.3%)
Sequential Reg [13]	13	**12.4**	579.6	**12.4** (0.0%)	579.6 (0.0%)	12.8 (+ 2.9%)	**383.0** (-33.9%)
Dual Path FP Sub [4]	62	**27.8**	1097.1	**27.8** (0.0%)	1097.1 (0.0%)	29.0 (+ 3.5%)	**929.4** (-14.9%)

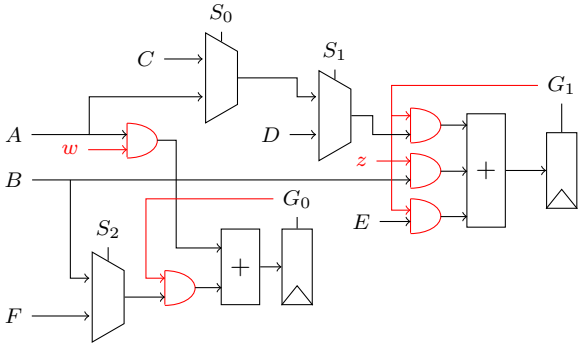
33.9% and 23.6% on average at the expense of an average 5.0% increase in circuit area. The reported power reduction is for a representative set of switching activity configurations. The area optimized designs do not demonstrate the same power reduction but show some limited area improvements. For several designs the area optimization could not find any improvement, returning the baseline implementation. Whilst ROVER only models dynamic power, we evaluate based on total power, including leakage power.

To best understand the benefit of exploring arithmetic, area and power in tandem, we study an open-source benchmark. Figure 6 shows the circuits corresponding to the baseline "Pipelined Mux Add Tree", the design proposed in [9] and the ROVER generated version. The optimizations proposed in [9] add transparent registers to both adder inputs, as this work only added operators. Meanwhile, ROVER performs an entirely different optimization, re-ordering the dataflow graph to push the adders towards the output of the circuit. Note that the ROVER generated design contains a three input adder, which, thanks to ROVER's comprehension of logic synthesis optimizations, is recognised as only a single carry-save adder, rather than two full carry-propagate adders. ROVER then inserts area efficient data gating on the adder inputs to save power. Synthesizing the design proposed in [9], the ROVER generated architecture is strictly better, consuming 11% less power within 17% less area.

For the "Combinational Mux Add Tree" ROVER once again re-orders the mux tree converting three separate adders to one single adder taking four inputs. This differs from the solution proposed in the Intel training materials. ROVER's design reduces both power and area by 10% when compared to the design proposed in the training materials. In the "Address Generation" benchmark, ROVER deploys data gating as recommended by the training material, but also an optimization to combine two adders into one three input adder. This area optimization offsets the overhead of the gating operators, leading to only a 2.2% increase in area. For the "Dual Op ALU" and "Sequential Reg" benchmarks, ROVER is able to rediscover the optimizations proposed in [10] and [13], demonstrating ROVER's ability to generalize prior work. Lastly, ROVER recognises the distinct computational paths in the "Dual Path FP Sub" and inserts the appropriate clock gating for each path.



(a) Baseline design (black). In [9] the authors add the transparent registers (red), where $x = (\overline{S_2}\&G_0)\|(S_0\&\overline{S_1}\&G_1)$ and $y = G_1$.



(b) ROVER generated design. ROVER rearranged the mux tree and added data gating (red), where $w = \overline{S_2}\&G_0$ and $z = G_1\&\overline{S_1}\&S_0$.

Figure 6. Circuit diagrams of the "Pipelined Mux Add Tree" benchmark with power optimizations from prior work and from ROVER.

For all but two benchmarks ROVER ran in less than 10 seconds, taking only a few seconds for the majority. For the "Address Generation" and "Weight Calculation" benchmarks ROVER ran for 130 seconds and 160 seconds, respectively. These long running cases were dominated by the ILP solver. Comparing the reduction in power consumption predicted by ROVER's model against the actual impact reported by logic synthesis, we see that for a group of five benchmarks the model provides a relevant estimate of the power reduction, within 14 percentage points of the actual. However, for the "Combinational Mux Add Tree" and "Address Generation" the model overestimates the improvement in power consumption by around 45 percentage points. We

TABLE 3. EACH ROW REPRESENTS A DIFFERENT SWITCHING ACTIVITY CONFIGURATION (CFG.) FOR THE MUX SELECT AND REGISTER ENABLE SIGNALS IN THE "PIPELINED MUX ADD TREE" (FIGURE 6A). FOR EACH CFG., IF ROVER INSERTED A DATA GATE (TRANSPARENT REGISTER) USING ONE OF THESE SIGNALS, WE COLOR THE CORRESPONDING CELL GREEN (PURPLE).

	Muxes			Registers		Total Power (mW)	
Cfg.	S_0	S_1	S_2	G_0	G_1	Baseline	ROVER
1	0.1	0.1	0.1	0.1	0.1	1.09	0.76 (-30%)
2	0.1	0.1	0.1	0.8	0.8	1.09	0.95 (-14%)
3	0.8	0.8	0.8	0.8	0.8	1.30	1.15 (-11%)
4	0.8	0.8	0.8	0.1	0.1	1.29	1.03 (-20%)

attribute this to two causes. First, for both benchmarks, the area model predicted an area reduction that was not realized. Second, the power model uses only a simple linear relationship between operator power and toggle frequencies.

4.2. Data Dependent Design

To demonstrate how ROVER is capable of tailoring the implementation to the computation, we study how ROVER's output changes as we modify the switching activities of mux select and register enable signals. We focus on the "Pipelined Mux Add Tree" as shown in Figure 6. In Table 3 we pass ROVER four different switching activity configurations and report the optimizations selected by ROVER for each. Given Cfg. 1, ROVER elects to insert data gating using the S_0, S_1, S_2 and G_1 signals. Given Cfg. 3, where we increase the switching activity for all signals, ROVER instead elects to insert a single transparent register. In Cfg. 4, we see ROVER use the G_0 signal to data gate, which we do not see in other configurations. The final columns show how the power benefit of ROVER's optimizations varies with switching activities.

5. Conclusions and Future Work

This paper describes how to encode power optimizations, such as operand isolation and clock gating, as local equivalence preserving rewrites. By phrasing power reduction as a rewrite problem we can combine it with existing arithmetic rewrites to explore both power and area in tandem. We developed an e-graph based rewriting framework, ROVER, that can simultaneously explore and balance the area-power tradeoff. The e-graph enables efficient simulation of many functionally equivalent implementations as we only need to simulate one node from each class. Optimizing a set of benchmarks using ROVER we see a 23.6% reduction in total power consumption on average for just an average circuit area increase of 5.0%. We show the importance of ROVER's understanding of downstream logic synthesis optimizations, leading to designs not seen in prior work.

Whilst ROVER is able to generalize the majority of optimizations discussed in prior work, the optimization presented in Figure 6a, is not expressible via the current set of rewrites. We can describe this as a resource sharing problem, a general optimization that future work on e-graph rewriting will address. We will also automatically derive useful case-splits from simulation stimuli to insert into the RTL.

References

[1] Synopsys, "Power Compiler," 2023. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/power-compiler.html

[2] Cadence, "Joules RTL Power Solution," 2023. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/power-analysis/joules-rtl-power-solution.html

[3] 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.

[4] S. Coward, G. Constantinides, and T. Drane, "Automating Constraint-Aware Datapath Optimization using E-Graphs," in *Design Automation Conference*, 2023.

[5] 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.

[6] 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.

[7] 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.

[8] R. Zimmermann, "Datapath synthesis for standard-cell design," in *Proceedings - Symposium on Computer Arithmetic*, 2009.

[9] M. Münch, B. Wurth, R. Mehra, J. Sproch, and N. Wehn, "Automating RT-level operand isolation to minimize power consumption in datapaths," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000*, 2000.

[10] V. Tiwari, S. Malik, and P. Ashar, "Guarded evaluation: pushing power management to logic synthesis/design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, 1998.

[11] A. P. Hurst, "Automatic synthesis of clock gating logic with controlled netlist perturbation," in *Proceedings - Design Automation Conference*, 2008.

[12] M. Donno, A. Ivaldi, L. Benini, and E. Macii, "Clock-tree power optimization based on RTL clock-gating," in *Proceedings - Design Automation Conference*, 2003.

[13] Siemens Digital Industries Software, "Automatic sequential clock gating with PowerPro," 2021.

[14] T. T. Hoang and P. Larsson-Edefors, "Data-width-driven power gating of integer arithmetic circuits," in *Proceedings - 2012 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2012*, 2012.

[15] Synopsys, "Design Compiler User Guide S-2021.06-SP2," Synopsys, Mountain View, Tech. Rep., 6 2021.

[16] 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.

[17] 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.

[18] S. Reda and A. N. Nowroz, "Power modeling and characterization of computing devices: A survey," *Foundations and Trends in Electronic Design Automation*, vol. 6, no. 2, 2012.