Imperial College London
Department of Electrical and Electronic Engineering

Equality Saturation for Circuit Synthesis and Verification

Samuel Coward

Supervised by George Constantinides and Theo Drane

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in
Electrical and Electronic Engineering of Imperial College London and
the Diploma of Imperial College, February 2025

Statement of Originality

I, Samuel Coward, declare that the work presented in this thesis is my own, and that any other work has been appropriately referenced.

Abstract

The core computational components of a modern Application Specific Integrated Circuit (ASIC) are implemented as datapath circuits, typically specified at the Register Transfer Level (RTL). In industry, the design, optimization and verification of such circuits is mostly a manual process performed by skilled engineers. This effort is a worthwhile investment since datapath circuits are usually the most timing critical circuits, occupying a significant proportion of the total circuit area. Unfortunately, due to the aggressive optimizations and inherent logical complexity, datapath circuits are associated with an equally complex verification challenge.

This thesis takes steps towards the automation of the optimization and verification of datapath circuits, developing tools capable of matching manual design performed by skilled engineers. A core idea behind the thesis, is to raise automated datapath optimization to the abstraction level typically explored by human engineers. This thesis proposes an approach that leverages a data structure, the e-graph that was originally invented by the formal methods community. Via an approach they call equality saturation, the compiler community have repurposed this data structure for program optimization. Applying equality saturation to datapath circuit optimization yields significant improvements in circuit power, performance and area, whilst also offering a robust verification flow that provides the correctness guarantees required by industrial circuit design.

This thesis describes a number of theoretical enhancements to equality saturation that permit the expression of new classes of optimization that go beyond those implemented by existing technology. The first of these formalizes the connection between equality saturation and program analysis techniques, creating a positive feedback loop that leads to analysis refinement and deeper exploration. The second, describes an approach to encoding context-awareness in equality saturation, capturing optimizations that stem from constraints expressed within a program.

Acknowledgements

First and foremost I wish to express my gratitude to Intel Corporation, not only for funding my PhD, but also for their willingness to experiment with a highly integrated PhD model. The access to Intel engineers, designs and resources has ensured the research is highly relevant in an industrial setting, addressing the difficult questions that remain in modern circuit design. This thesis would not exist without the vision and determination of Theo Drane, the consistent support of Altug Koker and those from both Intel and Imperial that brought both parties to an agreement, in particular Susan Capello and Jing Sheng Pang.

Thanks to the sponsored PhD model, I had the privilege of working with two wonderful supervisors. My deepest gratitude goes to both Theo Drane from Intel and George Constantinides from Imperial. It was a paper pointer from George during the early months of my PhD, that introduced me to the programming languages field and influenced so much of my PhD. George's ability to reduce a problem down to a core theoretical question helped guide me through much of the PhD, and allowed me to connect with a broad range of research communities. Meanwhile, Theo's drive and industrial perspective, pushed the research to not only answer difficult questions in circuit design, but to concisely communicate the solutions to countless different audiences. I would also like to thank my examiners Alastair Donaldson and Tom Melham for their challenging questions and valuable suggestions that have helped to shape this final thesis. Perhaps the most important lesson I learnt, is that the times of greatest success are the times at which it is most important to keep one's feet firmly grounded.

Within Intel, I was fortunate to be part of the Intel Numerical and System Level Design Group, which grew to six full-time staff at its peak. My sincere thanks go to Emiliano Morini for teaching me everything I know about verification and for many laughs along the way, to Rafael Barbalho for sharing his expertise in software design and for his drive to see my research deployed as widely as possible, to Bill Zorn for being an authority on all things programming languages, and to Chris Poole for his unbounded enthusiasm for research. Without this team, the scope of my research would be far more limited and the quality of my tools would be less. More broadly, I have worked with many great people across all parts of Intel. I am grateful to each one of them.

Being physically located in the Circuits and Systems Group (CAS) at Imperial has led to friendships and collaborations that I have no doubt will stand the test of time. To Aditya, Diederick, Alex M, Ben C, Alex D, Ben B and Zhewen my thanks go to making me feel so welcome at the start of my PhD. I would also like to thank Yann and Jianyi for exciting collaborations and for standing alongside me in the pursuit of better tools for hardware designers. To my travel companion extraordinaire, Marta Andronic, thank you for bringing an unrivaled energy to the office and for bringing positive culture change to CAS. A huge thanks must go to Wiesia Hsissen, who keeps the whole CAS group on track and has made my life easier on countless occasions. There are too many friends to highlight individually but my thanks go to Guoxuan, Mingzhu, Toni, Dan, Ebby, Michalis, Quentin, Pedro, Cheng, Violet, Cano, Keran, John, Omar, Sina and Zehui.

Lastly, it remains to thank the friends and family that supported me along the way. To James Van Der Walt, thank you for inviting me into your home on so many occasions and I can only apologize for any chaos connected to my visits, although I do not take sole responsibility. Thank you to my parents, without the hours spent on times tables in the car and your endless support I would not be where I am today. Of course, many thanks go to my partner Karolina, who has helped me to become a more considerate person, as we have navigated a sometimes challenging but mostly fortunate life together.

The chance to simultaneously span academia and industry has provided opportunities that few will have and I hope that I have made the most of them. To anyone who is fortunate enough to be offered such a role, I would highly recommend it.

Acronyms

AI	abstract interpretation
ASIC	Application Specific Integrated Circuit
CSA	carry-save adder
CSD	Canonical Signed Digit
DPV	Datapath Validation
EC	equivalence checking
ECO	Engineering Change Order
EDA	Electronic Design Automation
FIR	Finite Impulse Response
FMA	Fused Multiply-Add
FPGA	Field Programmable Gate Array
FV	Formal Verification
GPU	Graphics Processing Unit
HDL	hardware description languages
HLS	High-Level Synthesis
IA	interval arithmetic
ILP	integer linear programming
IP	Intellectual Property
LRM	language reference manual
LS	Logic Synthesis
LZC	leading-zero count

MCM Multiple Constant Multiplication

PPA power, performance and area

RL reinforcement learning

RTL Register Transfer Level

SAT Satisfiability

SMT Satisfiability Modulo Theories

STE Symbolic Trajectory Evaluation

TCAD IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In 2020, global microchip manufacturing reached one trillion units, according to a European Union report that also projects semiconductor demand will double between 2022 and 2030 [1]. This means that the need for efficient digital circuit design techniques is greater than ever. All digital circuit design approaches pair a skilled engineer (or team of engineers) with a suite of automatic design and verification tools. Unfortunately, consolidation in the Electronic Design Automation (EDA) industry has left engineers with few core design tools to choose from.

In 2024 there are broadly two classes of approach to digital circuit design. The first, more mature method, defines a digital circuit and its timing behavior at the RT-level, using an entrenched hardware description languages (HDL) such as Verilog [2] or VHDL [3]. The second, more recently developed approach, High-Level Synthesis (HLS) [4, 5, 6, 7], allows engineers to write their digital circuit designs at a higher level of abstraction, using extensions of software languages like C++. HLS tools consume these high-level circuit designs that do not specify the circuit's timing behavior. The HLS tool performs the complex steps to generate a functionally equivalent RTL design with the appropriate timing behavior. In both cases, the RTL generated is passed onto a Logic Synthesis (LS) tool [8, 9], the next step in the digital circuit design tool flow. Both design flows rely on graph representations of digital circuits, a dataflow graph for encoding data dependencies and a control flow graph for encoding possible execution paths. Accompanying the circuit design process is an architect, who also takes the natural language specification and produces a model, usually in a language like C++. Using a suite of verification tools the RTL implementations are verified against this model by a verification engineer, feeding any bugs back to the design engineers. Figure 1.1 illustrates the alternative methodologies and the respective tool feedback loops. In their current form, neither approach allows engineers to efficiently deliver high-performance and provably correct digital circuit designs, particularly for datapath circuits.

A challenge in the low-level design approach is that in addition to specifying the circuit's functional and timing behavior, the RTL engineer must also consider their power, performance and area (PPA) targets. Whilst LS performs many complex optimizations, it operates on a mostly fixed dataflow graph. For RTL written at a word-level of abstraction, LS will not re-order operations or apply many higher-level transformations, such as local resource sharing. This thesis will demonstrate that such techniques offer significant PPA improvements. My experience in Intel has shown how expert RTL engineers manually optimize their HDL code to realize these PPA gains. Often this involves iteratively running LS to evaluate the impact of the most recent changes, further refining the design based on reports generated by LS. Unfortunately, this is both bug prone and time consuming due to the long LS compile times, reducing the productivity of the design flow.

In the high-level design flow, the engineer no longer needs to specify the timing behavior but currently, nothing in the design flow performs the PPA optimizations implemented by the experienced RTL engineer. The HLS engineer is not expected to understand low-level digital circuit design principles, nor do the input languages allow them to succinctly express the same optimizations. Meanwhile, whilst HLS tools have matured significantly, they currently defer datapath optimizations to LS which, as already discussed, lacks the capabilities of expert engineers. For custom datapath designs, my experience and the continuing prevalence of RTL suggests that circuits designed using these high-level techniques are unable to match the PPA of those produced using the low-level approach, limiting industrial application to prototyping exercises. The introduction of a tool-chain, where LS consumes the output of HLS, introduces a potential correlation problem, where the circuit quality predicted by HLS may not match that produced by LS. Such mismatches can be frustrating, if not impossible, to resolve and

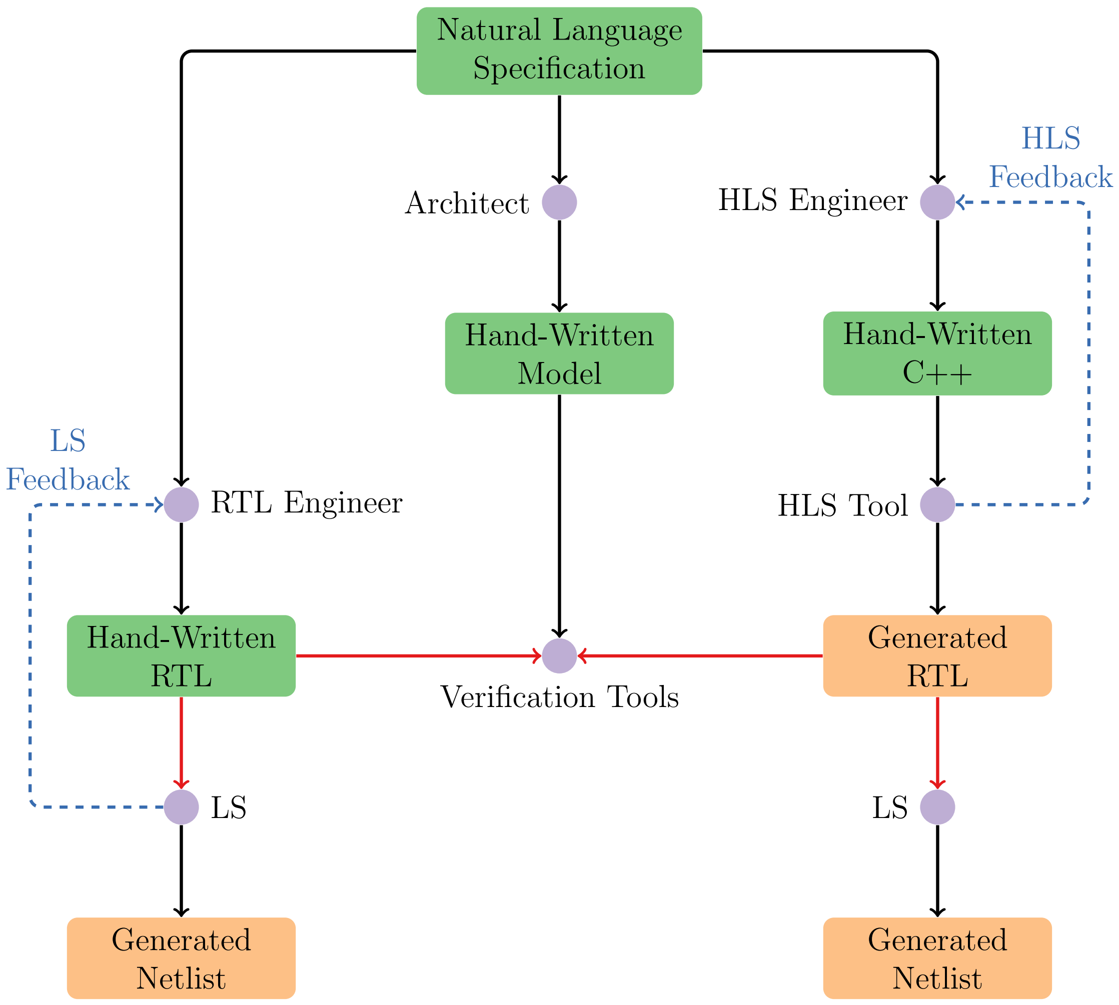


Figure 1.1: A comparison of digital circuit design flow methodologies. Red arrows denote where the design techniques described in this thesis are applied.

more importantly they could drive the HLS tool to make poor design decisions. Compounding the problem, many HLS tools lack an associated formal verification flow that can prove the correctness of the generated RTL, although Cadence's Stratus [6] is now more tightly integrated with the Jasper verification platform [10].

The above discussion highlights a gap between the high-level software optimizations performed by HLS and the low-level optimizations performed by tools like LS. Crucially this under-explored gap holds the potential for substantial gains in PPA of digital circuits, as Figure 1.2 shows. Currently, these PPA gains are realized via time consuming and bug-prone manual RTL optimization and design space exploration.

Borrowing terminology from a former EDA tool [11], we refer to this challenge as behavioral synthesis. More precisely, behavioral synthesis will refer to the process of transforming behavioral RTL, the sort of code written without considering PPA targets, into highly optimized RTL, for example after an expert RTL engineer has optimized the behavioral design. For the same reasons that HLS claims to improve circuit quality in design domains it is well suited to [6], automating behavioral synthesis also expands design space exploration resulting in better quality circuit designs. It also takes a large step towards realizing the productivity gains promised by HLS tools, reducing the engineering effort required to produce an optimized implementation. By its nature, behavioral synthesis takes larger steps through the design space, introducing a potentially challenging associated verification problem.

The automation of behavioral synthesis could be tackled in one of two ways: equipping HLS tools with the ability to capture and perform low-level hardware design optimizations, or by building upon existing RTL design flows. This PhD elected to deploy the latter approach. By operating on RTL, the techniques developed can be applied to both the human-written and automatically generated RTL, providing an opportunity for immediate industrial impact.

A key observation is that manual behavioral synthesis 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 [8], the process of determining a sequence of transformations to apply to an RTL design is currently based on designer intuition [12], 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. Figure 1.3 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 [13].

This thesis outlines an approach to automate and verify the results of behavioral synthesis for datapath circuits. Such an approach removes the need for manual RTL optimization, and

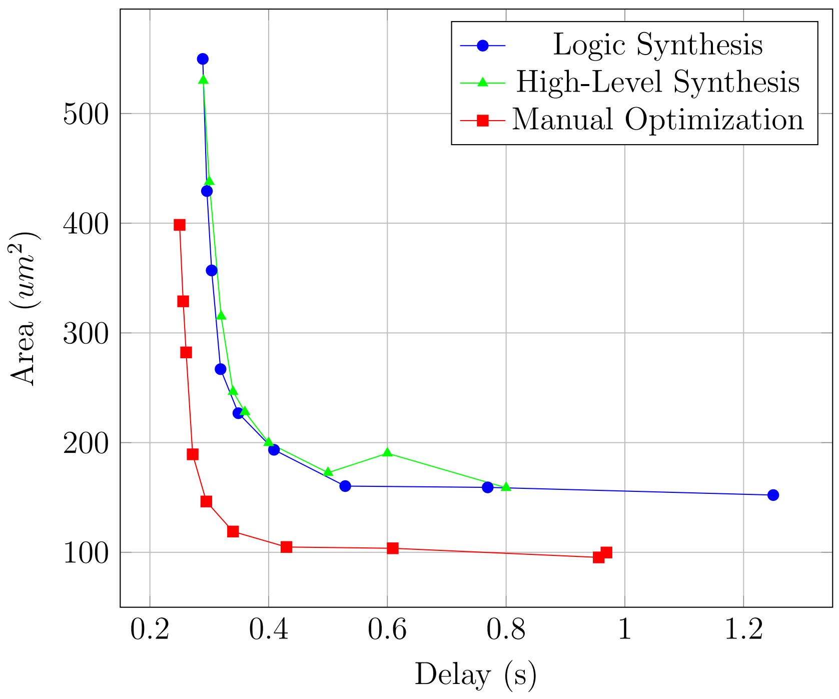


Figure 1.2: An area delay profile of the competing synthesis tools and human implementations of an Intel Media Kernel.

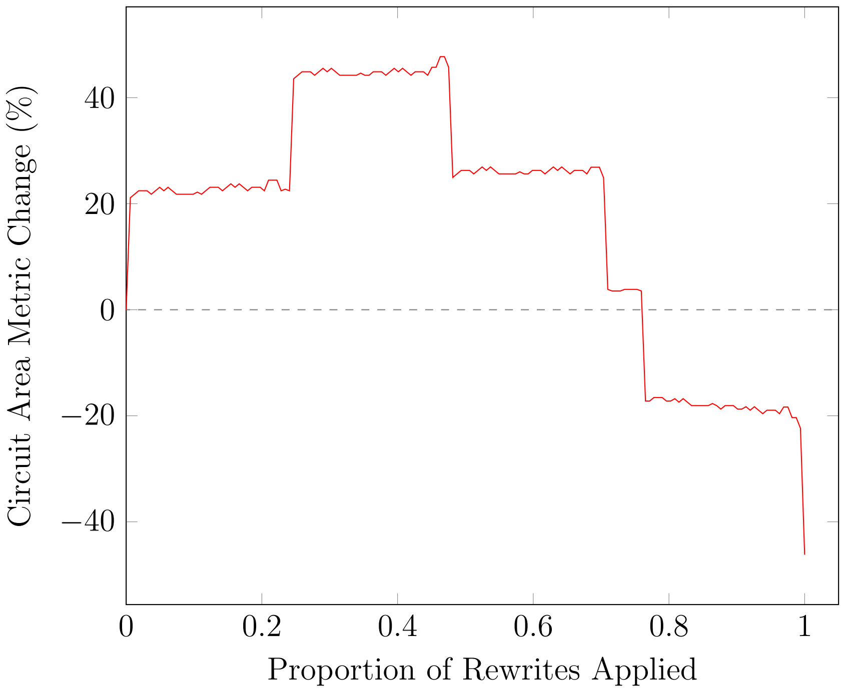


Figure 1.3: Progression of design cost throughout RTL rewriting for the Weight Calculation benchmark (described in Section 3.5). The plot shows the percentage change in the circuit area metric compared to the original design at every point in the rewrite chain. The area metric may converge non-monotonically.

enables greater design space exploration. The robust verification flow ensures that engineers can deploy the resulting circuit designs with full confidence. At the core of the approach is a technique from the programming languages community, equality saturation [14, 15]. Equality saturation addresses the challenge of determining an optimal transformation order leveraging the e(quivalence)-graph data structure, which is capable of exploring, in parallel, all possible transformation orderings, deferring the selection of the optimal ordering until a later stage. This thesis describes the application of equality saturation to behavioral synthesis and verification, and describes several general purpose extensions to the underlying equality saturation technology.

1.1 Problem Statements

This thesis focuses on two primary problems, an optimization problem and a verification problem. An important definition will be that of functional equivalence. Two circuits are functionally equivalent, $R \simeq R'$, if and only if for all possible inputs, all outputs of R and R' are equal.

Problem 1: Circuit Optimization

Given a design in the form of an RTL implementation R, find an RTL implementation R' that minimizes cost(R') for some cost function, such that $R \simeq R'$.

Problem 2: Circuit Verification

Given two designs in the form of RTL implementations, R_1 and R_2, mathematically prove that they are functionally equivalent, $R_1 \simeq R_2$, or produce an input (or sequence of inputs) for which the two circuits produce distinct results (a counter-example).

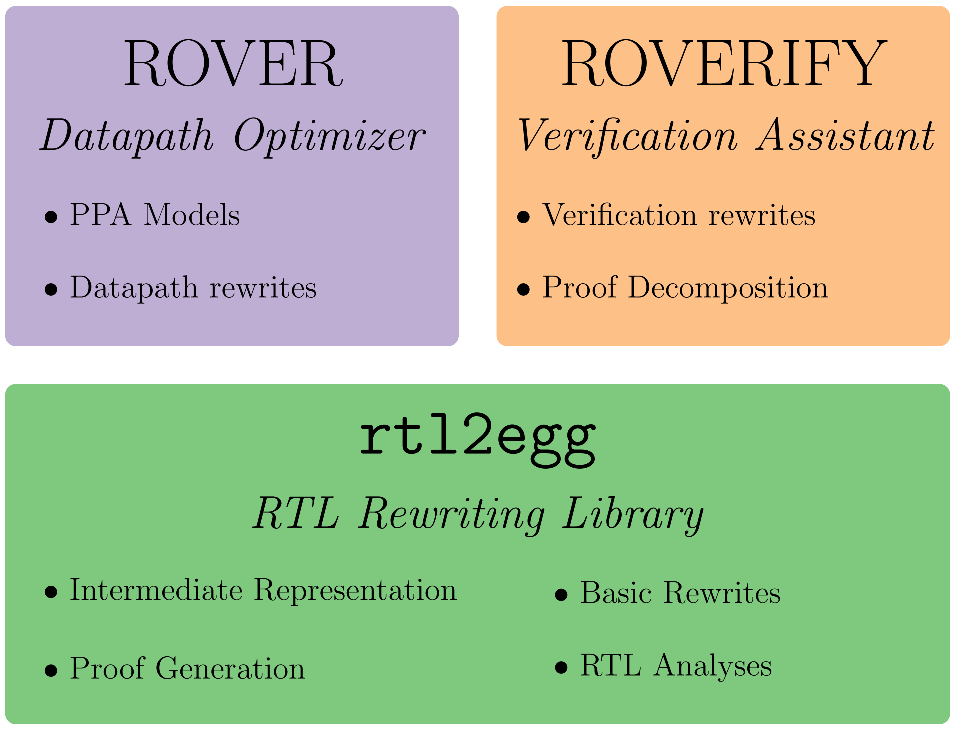


Figure 1.4: A diagrammatic representation of the library and tool suite developed.

1.2 Contributions

My PhD can be separated into a set of theoretical contributions and a suite of libraries and tools that apply the theory to digital circuit design and verification challenges. The theoretical developments were driven by an objective to express and evaluate hardware specific optimizations using the e-graph data structure. Firstly, an intermediate representation, VeriLang, is defined. VeriLang facilitates the representation of RTL using an e-graph and permits the description of multi-precision transformations, as described in Chapter 3. To further e-graph based analysis of digital circuits, Chapter 4 takes steps towards formalizing the connection between equality saturation and abstract interpretation, a key theory underpinning much of program analysis. To truly automate the techniques used by expert designers, a way to automatically reason about constraints present within the designs is essential. Chapter 5 introduces an approach to express sub-domain equivalences within an e-graph, enabling constraint-aware optimization.

Each of the theoretical contributions has been practically realized via a suite of digital circuit design and verification tools. At the foundation of this suite, is the `rtl2egg` library, built on-top of the `egg` e-graph library [15]. The `rtl2egg` library provides interfaces to map (System) Verilog to and from the `egg` library, via the VeriLang intermediate representation. It is also capable of generating associated proof certificates that can be checked using commercial verification tools to guarantee the correctness of its output. Beyond interfaces, `rtl2egg` implements a set of basic equivalence preserving RTL rewrites and RTL analyses.

A pair of applications are implemented on-top of `rtl2egg`. First, ROVER, an RTL optimization engine targeting word-level datapath circuit designs, initially developed to minimize circuit area (Chapter 3). ROVER later gained circuit analysis capabilities that leverage the formal framework described in Chapter 4. Taking advantage of circuit properties discovered via the analyses, ROVER later optimized circuit performance (Chapter 5) and power (Chapter 6). The second tool, ROVERIFY (Chapter 7), a formal verification assistant, is capable of automating proof decomposition for equivalence checking of an RTL implementation against an RTL specification. The complete software suite is illustrated in Figure 1.4. The remainder of the thesis will focus on the techniques and applications themselves rather than the software implementation details.

The most substantial contributions of the thesis are summarized as follows:

- the application of equality saturation to Register Transfer Level (RTL) datapath PPA optimization,
- multi-bitwidth and multi-signage rewrites that enable 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 formal verification tools,
- a robust method to verify the correctness of the generated RTL based on automated proof decomposition,
- a formal framework to enable program analyses on an e-graph,
- an accurate value range analysis for bitvector arithmetic in RTL designs,

- a general purpose encoding of multiple equivalence relations and constraint-aware optimization via e-graph rewriting,
- a computationally-efficient methodology to simulate a large set of design choices, leveraging the compact e-graph representation,
- a word-level e-graph framework that composes a set of sub-problems from local rewrites to assist formal verification tools.

1.3 Implementation and Evaluation

The industrial setting in which the research was conducted means that it is not possible to open-source the software implementation. This action avoids the risk of valuable Intel Intellectual Property (IP) becoming publicly available, a risk given the use of production Intel designs as benchmarks. Despite the lack of accompanying software, this thesis discloses the techniques necessary to reproduce the results presented throughout. In fact, there have already been several reproductions of elements of this thesis for use as a baseline comparison [16] or as an optimization engine in a circuit synthesis flow [17].

The research is evaluated using a combination of open-source designs taken from prior works and Intel-provided closed-source designs. A lack of standard and industrially relevant datapath benchmarks meant that it was necessary to compile relevant benchmark suites throughout this thesis. A summary of the benchmarks used is provided in Appendix A. Each chapter motivates the benchmark selection used to evaluate that project.

1.4 Thesis Outline

The thesis is organized into the following chapters.

Chapter 2 introduces the necessary background on digital circuit design and verification.

From the field of programming languages, equality saturation and the theory underpinning program analysis are both described.

Chapter 3 provides a first introduction to ROVER, describing the foundations of an e-graph based RTL rewriting framework, which is applied to circuit area minimization.

Chapter 4 formalizes the connection between e-graph rewriting and program analysis. To demonstrate broader applicability, the theory is used to derive tight bounds on arithmetic expressions.

Chapter 5 introduces the theory needed to encode constraint-aware optimization within an e-graph. ROVER is extended to incorporate constraint-awareness allowing the tool to discover highly optimized floating-point arithmetic components.

Chapter 6 completes ROVER's understanding of the key circuit design metrics, developing a power model and associated power reduction rewrites, which are combined with the existing arithmetic optimizations.

Chapter 7 tackles the verification problem, describing the application of the underlying RTL rewriting framework resulting in an automated verification assistant, ROVERIFY, and an automatic bug fixing tool, ROVERIFIX.

1.5 Publications

The research in this thesis has been the subject of the following publications:

ARITH 2022 & TCAD 2024 The first publication of the PhD was presented at the 29th ARITH and described the foundations of ROVER, presenting an initial application of equality saturation to datapath synthesis. The conference paper was nominated for a best paper award but did not win. It was later followed up with a IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems (TCAD) journal extension, which forms the basis of Chapter 3.

Automatic Datapath Optimization using E-Graphs, **Samuel Coward**, Theo Drane, and George A. Constantinides, IEEE 29th Symposium on Computer Arithmetic (ARITH) 2022, *best paper candidate*

ROVER: RTL Optimization via Verified E-Graph Rewriting, **Samuel Coward**, Theo Drane, and George A. Constantinides, IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems 2024

SOAP 2023 During a brief interlude from circuit design tools, the connection between equality saturation and abstract interpretation was explored, resulting in a short paper on the subject at a program analysis workshop. The workshop paper forms the basis of Chapter 4.

Combining E-Graphs with Abstract Interpretation, **Samuel Coward**, George A. Constantinides and Theo Drane, Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP) 2023

DAC 2023 & TCAD 2024 Returning to circuit design tools, a new objective was set, to match manual floating-point unit design using automated tools. Such an objective led to the development of constraint-aware optimization and analysis techniques that were first published at the 60th DAC, then later followed up with a TCAD journal extension. These publications form the basis of Chapter 5.

Automating Constraint-Aware Datapath Optimization using E-Graphs, **Samuel Coward**, Theo Drane, and George A. Constantinides, 60th Design Automation Conference 2023

Constraint-Aware E-Graph Rewriting for Hardware Performance Optimization, **Samuel Coward**, Theo Drane, and George A. Constantinides, IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems 2024

FMCAD 2023 The automation of deeper datapath optimizations, highlighted limitations of the formal verification tools which were relied upon to prove the correctness of ROVER's output. To address these limitations, an automated verification assistant, ROVERIFY, was developed on-top of the `rtl2egg` library and was presented at FMCAD in 2023. This paper forms the basis of Chapter 7.

Datapath Verification via Word-Level E-Graph Rewriting, **Samuel Coward**, Emiliano Morini, Bryan Tan, Theo Drane, and George A. Constantinides, in Formal Methods in Computer-Aided Design 2023

ARITH 2024 The final project of the PhD considered power optimization as a first class citizen. By combining arithmetic and power optimizations, entirely new optimal architectures were discovered, that reduce dynamic power consumption at minimal area penalty. The work was presented at the 31^{st} ARITH and was nominated for a best paper award but did not win. This paper forms the basis of Chapter 6.

Combining Power and Arithmetic Optimization via Datapath Rewriting, **Samuel Coward**, Theo Drane, Emiliano Morini and George A. Constantinides, in IEEE 31^{st} Symposium on Computer Arithmetic (ARITH) 2024, *best paper candidate*

Chapter 2

Background

This thesis leverages and extends theory from the programming languages community, applying it to problems in digital circuit design, with a specific focus on datapath circuit design and verification. Sections 2.1 and 2.2 provide the necessary background on both manual and automated approaches to these two challenges. From the field of programming languages, the thesis builds upon the e-graph data structure and explores how to apply program analysis techniques to digital circuit design problems. Section 2.3 introduces program analysis and, in particular, abstract interpretation, a key theory underpinning program analysis. Section 2.4 introduces the e-graph and equality saturation, providing a review of recent developments and applications in the area.

2.1 Datapath Circuit Design

Circuit design is often categorized into control flow and datapath design. Datapath circuits apply pre-defined operations on incoming data to perform some desired computation. Such circuits are often dominated by arithmetic operators, and the rate at which such arithmetic operations can be performed determines key silicon metrics, for example, floating point operations per second (FLOPS). Control flow circuits provide the orchestration, determining what operations the datapath circuits execute and where the computed results are needed. This thesis will be dedicated entirely to datapath design and verification, because such circuits implement the critical computational elements of a computer chip. However, the general purpose design techniques presented may also be applicable to control flow circuits.

Digital circuit designs may be physically implemented via programmable logic, for example on a Field Programmable Gate Array (FPGA), or via an Application Specific Integrated Circuit (ASIC), which is built to execute some fixed instruction set, for example a Graphics Processing Unit (GPU). The design of digital circuits must take into account the chosen implementation platform, since the resource constraints and performance characteristics are highly dependent upon this choice. As such, ASIC and FPGA design tools have diverged substantially. This thesis studies the design of digital circuits implemented as an ASIC.

The datapath design background is separated into four subsections. Firstly, Section 2.1.1 reviews industrial and academic contributions to automated datapath design at the RT-level. Next, Section 2.1.2 reviews manual datapath design research, as the thesis describes the automation of many of these techniques. Specializing further, Section 2.1.3 describes low-power RTL design techniques. Lastly, Section 2.1.4 surveys higher-level techniques for digital circuit design.

2.1.1 RTL Synthesis

As described in the introduction, RTL is one of the most common design abstractions for digital circuit design, providing a cycle-accurate description of the circuit's behavior. RTL is written using an HDL, most commonly Verilog, System Verilog or VHDL. Such languages allow digital circuit designers to express low-level functions in terms of Boolean operators, usually called gate-level design. However, the languages also permit word-level descriptions using arithmetic operators such as addition and multiplication. For maintainability, portability and ease of verification, designers are encouraged to write at the higher word-level abstraction. For greater performance and circuit efficiency, skilled engineers may turn to gate-level implementations that provide the maximum level of expressibility. In reality, industrial HDL is typically a mixture of these abstractions. The RTL, described in HDL, is then passed to a logic synthesis tool, which synthesizes a functionally equivalent netlist [18]. The netlist describes the circuit in terms of low-level cells connected by physical wires, where each cell is comprised of potentially many transistors.

Industrial, closed-source, logic synthesis tools such as Synopsys' Design Compiler [8] and Cadence's Genus [19], currently dominate the semiconductor sector. In the open-source community, the Yosys framework is the leading RTL synthesis [9] solution, with a growing ecosystem based around the OpenRoad project [20].

Efficient datapath design can have a significant impact on PPA, providing opportunities to share or re-use resources, reduce the logic depth or avoid redundant computations. As in all compilation tasks, higher-abstractions offer downstream tools greater flexibility and potentially permit more complex optimization. In particular, for word-level RTL designs, the logic synthesis tool must determine how to map higher-level operations such as addition into a low-level logic cells [21]. Such arithmetic components have been the subject of extensive study [22], providing a huge set of component level implementations to choose from.

Of crucial importance in ASIC designs is the avoidance of carry-propagation, since this is generally an expensive operation because the circuit must be able to propagate a carry from the least-significant to the most-significant bit. Carry-propagate adders are typically required at the output of any arithmetic operation, but can be avoided by deploying carry-save format [21]. Carry-save format stores the output of an arithmetic operation in a redundant format using two signals, a carry and a save. By grouping arithmetic operations into datapath blocks, the number of expensive carry-propagate adders can be minimized, replacing them with a compressor tree. This approach will be described further in Section 2.1.2.

The initial passes in modern industrial logic synthesis tools aim to extract the largest possible datapath blocks from the design, minimizing the number of carry-propagate adders [23, 21]. A datapath block can be viewed as a sequence of arithmetic operations in a dataflow design where all intermediate results can be stored in a redundant carry-save format. The success of this stage of the compilation flow is highly dependent upon the input RTL design, since the industrial tools do not substantially modify or re-order the word-level dataflow graph. As a result, the question of how best to formulate the RTL such that synthesis tools can maximally deploy their optimizations has received some research attention [24]. Industrial logic synthesis tools do perform some more standard optimizations such as constant folding and sub-expression sharing [21]. Since these tools are closed-source, understanding their capabilities is challenging. Automated flows that generate outputs to be consumed by these tools must make design choices based on a model of the logic synthesis tool's capabilities. This introduces a correlation problem between the internal model of logic synthesis and the logic synthesis tool itself.

A notable academic contributor in this field is Dr. Ajay Kumar Verma from EPFL. By using a set of rewriting rules, Verma and collaborators studied how arithmetic circuits can be re-ordered to maximally cluster additions together [24]. This approach goes beyond the datapath clustering described above, minimising the number of carry-propagations by explicitly transforming the dataflow graph. Many of the transformations described in this paper are incorporated into my work. Later work by Verma then proposed an approach that ran multiple passes over the circuit at differing levels of abstraction [12]. The separation of word and bit-level transformations allows the optimizer to explore beyond a limited local design space. Unfortunately, a pass based design flow introduces a phase-ordering problem forcing the use of heuristics to guide the exploration. The phase-ordering problem is discussed in more detail in Section 2.4. The work described in this thesis, provides a more general framework capable of expressing a broader range of RTL optimizations using an efficient search mechanism that does not depend on heuristics. My work also incorporates deeper optimization techniques involving context and value range analyses, achieving results beyond the capabilities of [24].

A semi-automated RTL design and verification environment has been developed by Carl Seger and collaborators [25, 26]. The Voss II framework provides a design visualization environment and proposes an interactive design space exploration approach, where the designer guides the application of a set of transformations, which are automatically checked.

In addition to this work tackling general circuit design optimization, there are many automated optimization methods targeting specific design instances. One well studied problem is the Multiple Constant Multiplication (MCM) hardware design challenge [27, 28]. The MCM problem asks, given a set of integer coefficients $\{a_1, ..., a_n\}$, what is the optimal architecture to compute the set $\{a_1 \times x, ..., a_n \times x\}$, where x is a variable. Competing solutions use a fixed number representation of the constants [28], often Canonical Signed Digit (CSD) representation [29], and or deploy an adder graph algorithm [27]. More recent work has formulated the problem as an integer linear programming (ILP) problem [30, 31, 32] or as Boolean satisfiability problem [33].

Specialized hardware generators form another class of automatic design tools, targeting specific problems. Elementary function implementations are typically constructed using hardware generators, with iterative [34] and piece-wise polynomial approximations being commonly deployed [35, 36]. The clustering of arithmetic operations, as described in Section 2.1.1, depends on hardware generators producing compressor trees for arrays of arbitrary shape [37, 38]. Making these custom hardware components flexible and usable by non-experts is a valuable target and one which this thesis take steps towards.

More recently, reinforcement learning (RL) has been successfully applied to the design of parallel prefix adders by researchers at NVIDIA [39]. By training an RL agent they were able to automatically generate highly optimized designs of prefix adders for a range of bitwidths. This technique proved successful and many instances of these generated circuits can be found in upcoming NVIDIA GPUs. Whilst successful such an approach deployed significant compute resources to optimize a widely used but highly specialized circuit, therefore may not be generalizable and scalable. A less mature, but potentially promising direction, is using Large Language Models to generate Verilog [40].

Clearly bespoke approaches will be able to solve such restricted domain optimization problems much more efficiently than a general approach. However, this raises the question of whether a more general approach can still reach the optimal solutions obtained by the bespoke method. This is a relevant question as instances of these specific problems can arise as components of more complex designs, which may be hidden or emerge during design space exploration. Chapter 3 will describe how to automate higher-level optimizations in an RTL rewriting framework that mitigates the logic synthesis correlation challenge and avoids the phase-ordering problem. Such a framework can be extended to incorporate custom solutions to specific problems.

2.1.2 Manual Datapath Design

Despite the large range of automated tools, industrial datapath design and optimization are often still manual tasks. Whilst this thesis will focus entirely on automated design, a short background on manual design is included, since the objective is to generalize and automate the type of optimizations performed by hand [38]. Over many years the computer arithmetic community has developed a large collection of highly optimized component-level designs. For example, arithmetic operations such as addition and multiplication are well-studied [22], yet composing these components and customizing them to the a particular use case remains a challenge. This section will provide a brief survey of key components that will be used throughout this thesis, as understanding the underlying architectures will help to guide automated design decisions.

A first important building block is the full-adder, which takes three input bits and produces two output bits representing their sum. Such low-level circuits can be composed to construct higher-level components, for example key datapath circuits to compute bitvector addition. The simplest and minimal circuit area implementation is a ripple carry adder, where full-adders are chained together to produce the sum of two bitvectors [22]. Many faster adders have been proposed [38], but of particular importance is the parallel prefix structure [41] that facilitates the decomposition into a logarithmic tree structure. Through standard bitvector manipulations, subtraction can also be implemented using these same structures. The design of addition circuits is well covered in numerous textbooks [22, 29, 38].

Unsurprisingly, via the schoolbook approach, multiplication can be decomposed into a sequence of additions. However to improve performance it is most commonly separated into three stages. First, an array creation stage, where each row of the array represents the product of one operand with a single (or potentially multiple) bits of the other operand. The height of this array is then reduced to just two rows using a compressor tree [42, 43], comprised of compressor cells such as the full-adder [38]. Lastly, a bitvector adder sums the remaining two rows. Once again, there has been significant research into fast and area efficient multiplication circuits, in particular looking at alternative encoding schemes to reduce the height of the multiplication array, all of which are covered in textbooks on the subject [22, 29, 38].

These fundamental arithmetic operators can be modified and extended to perform other operations. Of particular importance is the compound adder [38], which, for minimal overhead can produce both $a+b$ and $a+b+1$. At first, this may not be obviously useful. However this compound adder can be used to efficiently compute an absolute difference (deployed in Chapter 5) and an optional pre-rounding for floating-point hardware. More complex components such as a leading zero counters, detectors and anticipators are also widely used [44], becoming part of standard IP libraries [45].

Floating-point hardware design increases the complexity once again, introducing an alternative mathematical meaning to different slices of the input bitvector. In particular, floating-point addition and subtraction have received significant attention [46, 47, 48]. Beyond single floating-point operations, there has been interest in larger compound components where rounding can be deferred to avoid error accumulation [49, 50, 51, 52, 53]. Such components offer a significant design space to explore, and are typically built by floating-point design experts, who understand the complex accuracy requirements of their application. This thesis will take steps towards automating this design space exploration, producing results comparable with manual implementations.

2.1.3 Low-Power RTL Design

Historically, low-power ASIC design has been the concern of specialized design teams targeting mobile devices. Today, the rise of power efficient mobile chips and large scale data centers has ensured that power efficiency is a key objective for almost all ASICs. Digital circuits consume energy through the charging and discharging of capacitances inherent in CMOS logic gate design. The rate of energy consumption due to logic gate switching is known as dynamic power. They also use energy at idle due to leakage currents, known as leakage power. Dynamic power is the primary concern for RTL designers, since leakage power is more heavily influenced by lower-level design choices, such as cell selection.

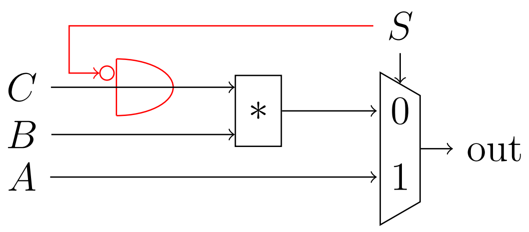


Figure 2.1: An operand isolation opportunity. The input to the multiplier can be data gated when the select signal is one, as shown by the red gate. The negated select signal, $\sim S$ is a common input to an array of AND gates equal to the bitwidth of C.

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. The second set of optimizations 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 an exceptional input is detected in the first stage, all registers on the standard input path can be gated 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 2.1. In this circuit the multiplier is a redundant operation, so one of its inputs can be zeroed using an activation signal, 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 [54, 55]. The transparent register is a synchronous component that has an enable signal, which, when high, allows the input to transparently flow through to the output. When low, the transparent register outputs the same signal as the previous clock cycle. Prior work has often implemented a transparent latch, which is a similar, asynchronous component, that instantly freezes a signal value when disabled [54, 55].

In academia, clock gating has been explored at a gate-level [56] and from a clock tree synthesis perspective [57]. A subset of industrial tools, such as Synopsys Power Compiler [58] and

Cadence Joules [59], are incorporated into their proprietary logic synthesis engines and automatically perform clock-gating optimizations. The Synopsys synthesis tools also take operand switching frequencies into account when selecting arithmetic components, for example when deciding which multiplier operand to Booth encode [21]. Siemens PowerPro [60] 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, which may miss opportunities as we shall see in Section 6.1.2. The automation of operand isolation has been explored at both the word-level [54] and at gate-level [55, 61], typically adding additional operators to a netlist in-order to reduce switching activity.

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 [58, 59]. 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 [8], take user provided simulation configurations and perform power optimizations guided by the simulation.

Much of the academic community appears to have lost interest in RTL power optimization, with few papers published in this area over the last decade. Chapter 6 attempts to reverse this trend, describing how to encode the power optimizations discussed above as local RTL rewrites and a framework to explore and evaluate the combination of power, arithmetic and area optimizations.

2.1.4 High-Level Synthesis

Whilst not the primary focus of this thesis, HLS provides an alternative design flow that aims to resolve many of the challenges discussed in this thesis. HLS aims to substantially raise the abstraction level of hardware design by allowing engineers to specify their circuits in languages like C. HLS tools automatically map a high-level software program into a custom hardware design in a low-level HDL, *e.g.* Verilog. A production HLS development flow comprises three steps. First, a high-level specification of an algorithm is *manually* rewritten following the recommended coding guidelines producing code that is amenable to optimization by the HLS tool. Second, the rewritten HLS program usually contains design constraints expressed via inline directives or pragmas to exploit hardware parallelism and resource sharing. The process of exploring these constraints is already semi-automated [62, 63, 64]. Finally, the optimized design constraints are sent with the HLS program to the HLS tool, which synthesizes a hardware design. The HLS tool automatically performs certain hardware optimizations, such as hardware scheduling and binding, which maps the start times of operations into clock cycles with efficient hardware resource sharing [65, 66, 67, 68]. The HLS tool also performs register retiming to achieve a high clock frequency [67, 68]. HLS tools are less mature than logic synthesis tools, with industrial leaders being Cadence's Stratus for ASIC design [6] and Vitis HLS for FPGA design [7]. In the open-source domain, LegUp [5] and Bambu [69] have been the subject of sustained research efforts.

The resulting RTL description can be fed into logic synthesis to produce a final netlist. Once again designers are faced with a problem. How to formulate the C-code such that the tool can best optimize it? Approaches such as graph neural networks [70] and genetic algorithms [71] have been applied to this problem. Both these works provide an alternative way to answer the tool correlation questions tackled in this thesis, albeit at a lower abstraction level.

Several works have even looked at how to deploy e-graphs in the HLS flow, with one work studying the decomposition of large multipliers [72]. Although not discussed in this thesis, with collaborators, I recently explored the application of techniques discussed in this thesis to HLS code optimization [73], integrating the e-graph into a mature software compilation stack.

2.2 Datapath Formal Verification

The design of digital circuits generates an equally important associated challenge, to determine the correctness of a given implementation. Formal verification is a mathematical approach to proving the correctness of a circuit for all possible input sequences, under some assumed operating conditions. For industrial scale circuits, such correctness guarantees cannot be obtained via simulation since the input space is too large to exhaustively test. Formal verification of datapath circuits is particularly challenging as they are subject to intense optimization effort, both automated and manual, in the design phase.

Classic formal property verification methods successfully used to verify state machines and communication protocols are not able to verify datapath dominated circuits. During the 1990's a number of Theorem Proving [74, 75, 76, 77, 78] and Symbolic Trajectory Evaluation (STE) [79] approaches were developed. These techniques involved comparison against mathematical specifications, often taken from industry standards, *e.g.* The IEEE Standard for Floating-Point Arithmetic [80]. Whilst capable of verifying complex designs, Theorem Proving and STE suffered from common downsides, a high barrier to entry and maintenance of complex code bases. One relevant work overcomes the barrier to entry by combining rewriting and theorem proving to automatically verify the correctness of gate-level multiplier designs in RTL [81, 82]. In this work, the authors deploy ACL2 verified [83] rewrites to transform optimized implementations into normalized implementations.

In the last decade, the circuit design industry has converged on equivalence checking (EC), defining two circuit representations to be equivalent if for all valid inputs they generate identical outputs. EC has been used in several contexts in the semiconductor industry [84, 85]. The most popular types of EC are Boolean, Sequential and Transactional. This thesis primarily focuses on Transactional EC of combinational circuit descriptions, where the result of a given computation in the *implementation* is compared against the result of the same computation in the trusted *specification*. The output of the comparison can be *Pass*, when a property is proven, *Fail*, when the property is not true (a counterexample is generated), or *Inconclusive*, when the tool does not manage to either prove or disprove a property. This setup is illustrated in Figure 2.2. Sequential EC is instead usually used to compare a version of the RTL against an altered one, for example when performance optimizations are introduced (such as clock gating) or small algorithmic changes are implemented. In this case, the cycle-accurate equivalence of the two designs is verified. Boolean EC targets gate-level verification, often comparing a synthesized netlist to an RTL implementation.

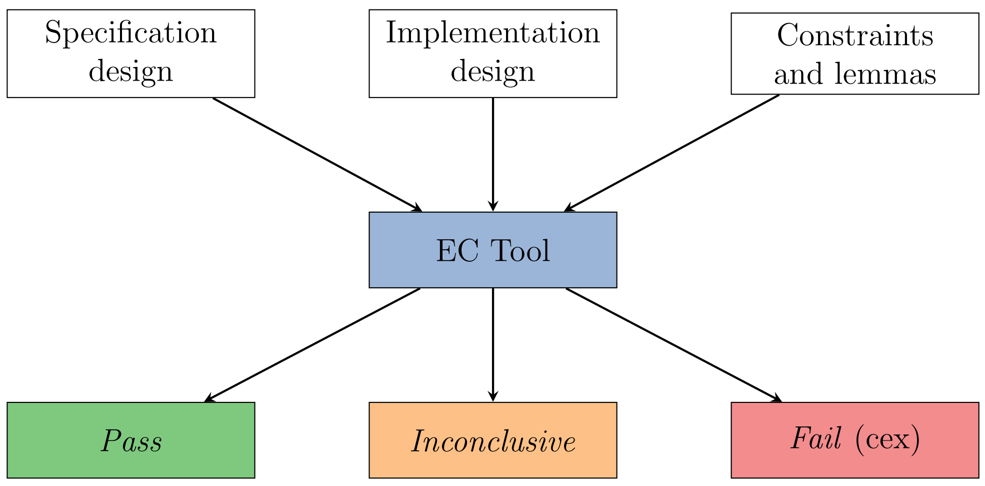


Figure 2.2: The inputs of an EC tool are two designs, a specification and an implementation, a set of constraints to drive the possible values to tests and a set of lemmas to prove. Each lemma can pass, fail or be inconclusive. A counterexample (cex) is provided for each failing lemma.

A trusted reference is fundamental for EC. One standard verification flow used in the semiconductor industry is the following: starting from a component specification, a developer writes a high-level reference C++ design without any interaction with the designer who writes the RTL implementation, providing diversity and independence between the two, which are then formally tested for equivalence. Many more tests can be run on the C++ code, due to the great difference in simulation speed between C++ and RTL. Furthermore, given the lack of low-level optimizations in this type of code, it is easier to identify the hard corner cases and test them thoroughly. If the EC tool passes but there is a bug in the implementation, it would require the same bug to be encoded independently in both models, which are written in different programming languages and at a different level of abstraction. This is usually described as C2RTL EC [10]. Another option is RTL to RTL EC, where the reference is a trusted version of the same design in RTL [86], usually an earlier version or based on a third-party library like Synopsys' DesignWare [8].

Automated EC tools have been developed in academia [87] and industry [86, 88], all of which rely on orchestrating a suite of proof engines to prove the formal equivalence of the implementation and specification. In academia, SymbiYosys is built on the Yosys synthesis tool, providing an integrated verification environment. For datapath verification, SymbiYosys produces encodings in bitvector theories then relies on Satisfiability (SAT) and Satisfiability Modulo Theories

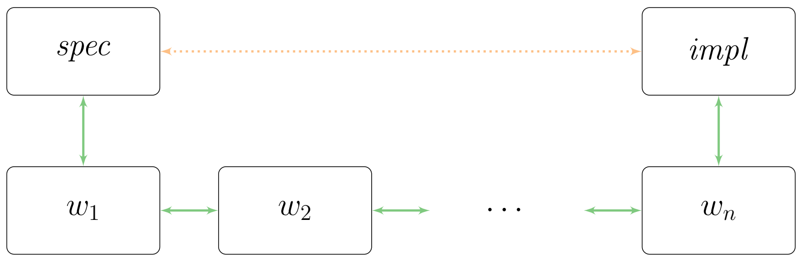


Figure 2.3: The waterfall approach used by FV engineers. The dashed line between *spec* and *impl* represents an inconclusive verification. Convergence is achieved introducing n intermediate designs w_i and proving the equivalences of all the pairs ($spec$, w_1), (w_1, w_2), . . . , (w_n, $impl$).

(SMT) solvers, such as CVC5 [89] and Z3 [90]. Industrial tools, such as Synopsys' Datapath Validation (DPV), Cadence's Jasper [10] and Siemens' SLEC [88], are more sophisticated and incorporate a range of proof engines, SAT, SMT, binary decision diagrams (BDD) and rewriting, to prove the equivalence of the two designs. This orchestration layer above the proof engines facilitates problem partitioning and decomposition, yielding greater capabilities than a single engine alone.

Despite significant advances in Formal Verification (FV) tools, inconclusive results are commonplace in practice and require advanced techniques to achieve full convergence, occupying most of the FV engineer's time. A common approach is to generate a "waterfall", where the verification between implementation and specification is decomposed into a sequence of EC proofs by introducing intermediate designs, as shown in Figure 2.3. If all the intermediate equivalence steps are proven, the equivalence between specification and implementation holds. Throughout this thesis, automated proof decomposition is repeatedly utilized to provide robust FV certificates. Chapter 7 explores an enhancement to the EC tools themselves, since even small examples can prove challenging for existing tools.

2.3 Program Analysis

To generate competitive hardware designs it is necessary to learn certain properties of a program or design. Traditional compilers separate program analysis and optimization into distinct passes, where the analysis may enable or prove the validity of deeper optimizations, for example dead code elimination [91, 92]. A widely used theory underpinning program analysis is abstract interpretation (AI), which over-approximates program properties using an abstract domain [93]. An element in an abstract domain typically represents a set of possible program properties. This element over-approximates a program's properties if the concrete properties of the program are a subset of the possible properties represented by the abstract element. AI is used in static program analysis [94], where the program is analyzed without being executed. Based on lattice theory, it is a theory for computing sound approximations of program properties.

A general objective of static analysis tools is to efficiently compute abstractions that provide an accurate over-approximation of the concrete program properties. Cheap to compute interpretations typically over-approximate, sacrificing accuracy for computational efficiency. A simple, but relatively weak, interpretation is interval arithmetic, approximating numerical variables by their possible input ranges, and operators by their natural interval extension. Knowledge of these variable ranges may enable further optimizations or prove the absence of dangerous behavior, for example division by zero. To reduce the coarseness of the over-approximation, existing tools have incorporated term rewriting to discover "analysis-friendly" program representations [95], since alternative program representations yield distinct abstract interpretations. For example, consider two integer variables $a \in [0, 1]$ and $b \in [-2, 1]$, an interval interpretation of equivalent expressions $a \times b - b$ and $(a - 1) \times b$ yields $[-3, 3]$ and $[-1, 2]$, respectively. Chapter 4 will explore the theory underpinning the combination of e-graphs and program analysis. This exploration is one contribution to work deepening the connection between abstract interpretation and compilation [96].

Capturing constraints in an abstract interpretation is typically more computationally expensive and involves analyzing relational domains. Prior work has combined constraint-awareness with interval arithmetic [97], which shall also be combined and applied to RTL optimization in Chapter 5. This thesis will take inspiration from [98], where the authors introduce special `ASSUME` operators, that couple sub-programs with the constraints (introduced by control structures) which can be assumed upon evaluation. This coupling facilitates abstraction refinement, yielding a more precise abstraction.

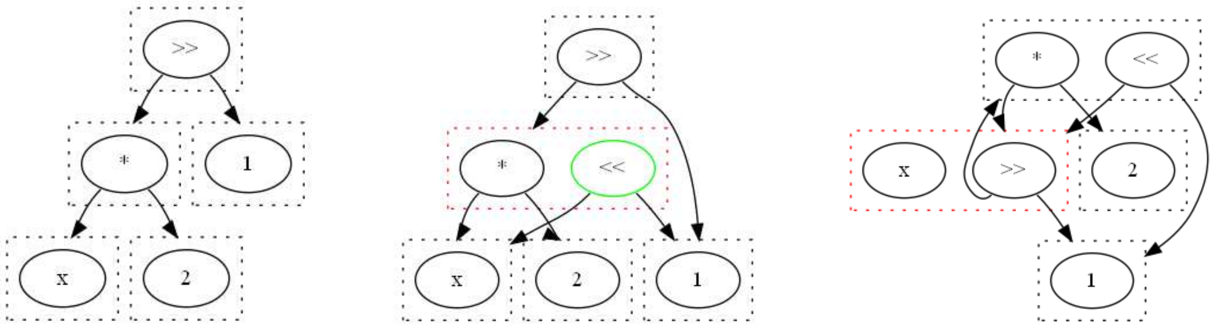
In hardware design, existing HLS [6, 7] and verification tools [86] are known to compute a value range analysis. Interval arithmetic of RTL has been combined with SAT to aid circuit verification [99]. In general, there are far fewer RTL analysis tools than can be found in the software domain, with great emphasis placed on verification and simulation tools. This thesis will make extensive use of RTL program analysis, to enable deeper optimizations. The interaction between RTL analyses and optimization has been explored using model checking [100], although this represents a relatively specific solution that sought to remove unutilized handshake logic.

2.4 E-Graphs and Equality Saturation

The thesis builds on a core data structure, the e-graph, and leverages an associated program optimization technique, equality saturation. First proposed by Nelson [101], equivalence graphs, commonly called e-graphs, provide a compact representation of equivalence classes (e-classes) of expressions. Often found in theorem provers [90, 89], this data structure enables a graph optimization technique called equality saturation [15, 102, 14]. The e-graph represents expressions, where the nodes, known as e-nodes, represent function symbols (including variables and constants, as 0-arity functions) and are partitioned into a set of e-classes, with respect to an congruence relation, $\cong$. The intuition is that e-classes can be used to compactly represent *equivalent* expressions. Edges represent function inputs and are from e-nodes to e-classes; see Figure 2.4, where dashed lines represent e-class boundaries, solid ellipses are e-nodes and arrows denote operator arguments. This representation intuitively captures the local choice of how to implement a given subset of the program.

2.4.1 Equality Saturation and Rewriting

Equality saturation is based on the theory of uninterpreted functions, where operators only gain meaning via rewrites and analyses defined on them. Rewrites, $\ell \to r$ define equivalent expressions such that when ℓ is matched in an e-graph, a process known as e-matching [103], the expression r gets added to the same e-class. For example, for any expression x, $x + x \to 2 \times x$



(a) Initial e-graph contains $(2 \times x) >> 1$ (b) Apply $x \times 2^i \rightarrow x << i$ (c) Apply $(x << s) >> s \rightarrow x$

Figure 2.4: An example of how rewrites can be applied to an e-graph representing standard integer arithmetic. The dashed boxes represent equivalence classes. Green nodes represent new nodes added as a result of a rewrite, whilst the red dashed boxes just highlight which equivalence class has grown.

encodes the equivalence $x + x \cong 2 \times x$ in the e-graph. Such rewrites are applied constructively to the e-graph, meaning that the left-hand side of the rewrite remains in the data structure, avoiding 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. When a rewrite is applied, a pair of e-classes may be merged. To maintain the congruence relation, this merging is propagated via congruence closure. For example, if an e-graph contains terms $f(x)$ and $f(y)$, and a rewrite $x \rightarrow y$ is applied, the x and y equivalence classes are merged. This rewrite implies, by congruence, that $f(x) \cong f(y)$ and hence the equivalence classes containing $f(x)$ and $f(y)$ should be merged.

Rewrite rules, and in particular conditional rewrite rules will play a central role throughout this thesis. A conditional rewrite is of the form $\phi \Rightarrow \ell \rightarrow r$, where ϕ denotes some predicate. For example $x \neq 0 \Rightarrow x/x \rightarrow 1$. Such rewrites require the e-graph construction algorithm to determine whether the rewrite is applicable for each matching instance [15].

In traditional compilers, determining an optimal order in which to apply transformations is known as the phase-ordering problem [92]. The key idea behind equality saturation is to defer this ordering choice and efficiently explore all possible rewrite schedules. Whilst potentially computationally expensive, 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 sat-

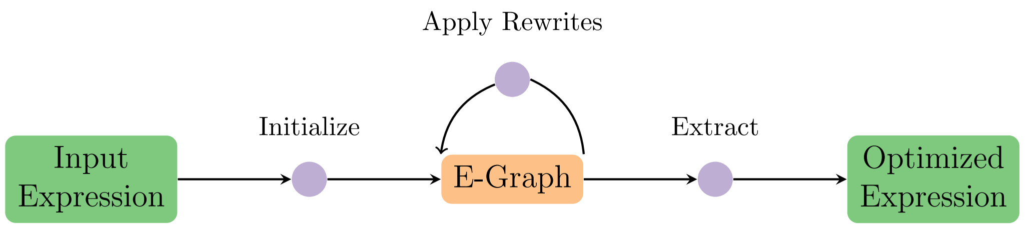


Figure 2.5: A typical equality saturation flow, where rewrites are iteratively applied in a loop.

urated. From an e-graph representing potentially infinitely many equivalent expressions, the best expression is selected via a process known as extraction [15]. Typically this choice is driven by an application specific cost model, imposing an order on the expressions. A simple greedy extraction method selects the e-node of lowest cost from each e-class. Figure 2.5 provides an overview of a standard equality saturation flow.

One limitation of the e-graph is that all rewrites define equivalence with respect to the same equivalence relation, making it a challenge to express constraint-aware optimizations. Consider the following expression containing an arbitrary function f.

$$(x > 0\ ?\ f(\texttt{abs}(x))\ :\ 0) \cong (x > 0\ ?\ f(x)\ :\ 0) \tag{2.1}$$

Whilst these two expressions are clearly equivalent, it is difficult to express via local rewrites since, in general, the rewrite $\texttt{abs}(x) \rightarrow x$ is not valid. However, the constraint-aware rewrite $x > 0 \Rightarrow \texttt{abs}(x) \rightarrow x$ is always valid. To resolve this conflict, one prior work proposes “Colored E-Graphs” [104]. In a colored e-graph, different equivalence relations (each represented by a different color) are layered on the e-graph, creating a hierarchy, at the expense of complexity in modifying the underlying congruence closure algorithm. Chapter 5 proposes a simpler approach to encode context in the e-graph without modifying the e-graph data structure. More recently, there has been work to unify these approaches to encoding context within the e-graph [105], viewing the collection of sub-domain equivalence relations as a lattice structure.

2.4.2 Egg

`egg` is a recent e-graph library, which is intended to provide a general purpose and reusable implementation [15]. It adds powerful performance optimizations over existing, usually bespoke, e-graph implementations along with some useful additional features. 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 an operator cost, and a rewrite set – that is a set of equivalences over the given language definition.

`egg` handles the search for rewrite opportunities, a process known as e-matching [103], merging the new terms into the matched e-classes. The library also implements an efficient congruence closure algorithm [101], to maintain the congruence relation throughout the e-graph. A performance enhancement introduced by `egg` is to apply rewrites in iterations, detecting a set of valid rewrites and inserting the new terms, but only propagating these new equivalences (via congruence closure) through the e-graph at the end of each iteration [15]. This invariant restoration process is called rebuilding by the `egg` authors. The performance benefit comes from the de-duplication of updates to the underlying data structures [15].

The e-graph is grown until some computational limit is reached, often a user-specified number of iterations, or saturation is reached. In many applications, the objective is to produce the "best" equivalent expression. To select the best expression from the final e-graph, `egg` implements a default greedy extraction method. It traverses the e-graph working bottom-up towards the root, using the supplied local cost function to choose the best equivalent node from each equivalence class. The cost function is local, as the node cost is only a function of the node itself and its children. The best expression is then given by the node with the lowest cost that is found in the same equivalence class as the original root of the initial e-graph.

The `egg` library adds two features on-top of the basic equality saturation engines, that will be used throughout this thesis. Firstly, e-class analysis allows users to implement program analyses on the e-graph. Analysis data can be attached to each e-class, lifting an abstract interpretation to the e-graph level. Given an abstract interpretation defined on the expressions represented in the e-graph, abstract elements can be combined using the semi-lattice meet or join, depending on the analysis objectives. The consequences of this combination will be investigated in Chapter 4. Previous work introduced a congruence closure abstract domain, that raised constraint reasoning to the level of e-classes, improving the precision of static analyses tools [106].

The second valuable feature is the ability to produce proofs [107]. Given two expressions represented within the same e-class, `egg` is able to produce a sequence of rewrite applications that map one expression to the other. Such a feature is valuable for providing a certificate that can be checked independently to verify the equivalence of two expressions derived from an e-graph. This thesis relies on this feature to provide a robust formal certificate that can be checked using industrial tools.

2.4.3 Applications

E-graphs were invented in the 1980's for use in the theorem proving community [101]. To this day, e-graphs are a key data structure utilized by SMT solvers [90, 89], retaining a collection of equivalence facts to be combined and added to by specific theory engines. The equality saturation approach and application to optimization problems was more recently proposed [14]. Much of the recent research into applications and extensions to equality saturation have been fueled by the release of the `egg` library.

The earliest users of `egg` were the Herbie developers [108, 15]. Herbie is a tool that uses equality saturation to automatically improve the numerical stability of floating-point programs. Replacing Herbie's original custom equality saturation implementation with `egg` yielded over a 3000x runtime improvement, inspiring other applications to build on the `egg` library.

A comprehensive survey of recent e-graph research developments and applications is beyond the scope of this background, so only a few notable applications will be highlighted. In linear algebra, equality saturation has been used to optimally map programs onto vector processors [109, 110] and optimize linear algebra kernels for machine learning [111]. In the field of compilers, equality saturation has been integrated into the industrial open-source Cranelift compiler [112], where the developers introduced a novel representation of control-flow in an e-graph. In the program synthesis community, equality saturation has been leveraged to synthesize efficient rewrite rule sets [113] further developing their methodology to target specific workloads [114].

In the hardware domain, there is a growing interest in the application of e-graphs to hardware design challenges as a result of the work presented in this thesis and other work published at the same time. At the lower level, equality saturation helped to minimize gate count in logic synthesis [115] and improved the results in hardware synthesis challenges [17]. Whilst at a higher abstraction level, `egg` was used within FPGA HLS to determine optimal multiplier decomposition [72] and in ASIC HLS to combine loop-level and datapath optimizations [73]. The ability to generate diverse hardware mutations also led to applications in fault tolerant hardware design [116]. In addition to these research contributions, there have been a number of position papers presented that advocate for the wider adoption of e-graph techniques to address hardware challenges [117, 118]. The contributions described in this thesis represent some of the earliest work in this research direction.

Chapter 3

Circuit Area Minimization via Verified E-Graph Rewriting at the RT-Level

This chapter describes an automatic equality saturation based approach to address the circuit optimization problem described in Section 1.1, specifically targeting circuit area minimization. Starting from a behavioral RTL design, the technique automatically generates an optimized RTL implementation and accompanying proof certificate that can be formally checked to guarantee that the implementation is functionally correct. The generated implementation is passed to an industrial logic synthesis tool, producing a netlist from which relevant circuit quality metrics can be extracted. In the context of the given tool chain, the optimization objective is to generate RTL that the logic synthesis tool can synthesize into the minimal area netlist representation.

Phrasing RTL optimization as an equality saturation problem introduces a number of challenges. First, how can RTL designs be represented in an e-graph? Second, what is a sufficiently general and expressive set of rewrites that capture manual RTL optimization? Third, having grown an e-graph of equivalent design candidates, how should the implementation corresponding to the minimal area netlist be selected, ensuring that the selection is well correlated with the downstream logic synthesis tool? Finally, having generated an optimized implementation, how can the correctness of the design be formally proven using a robust methodology?

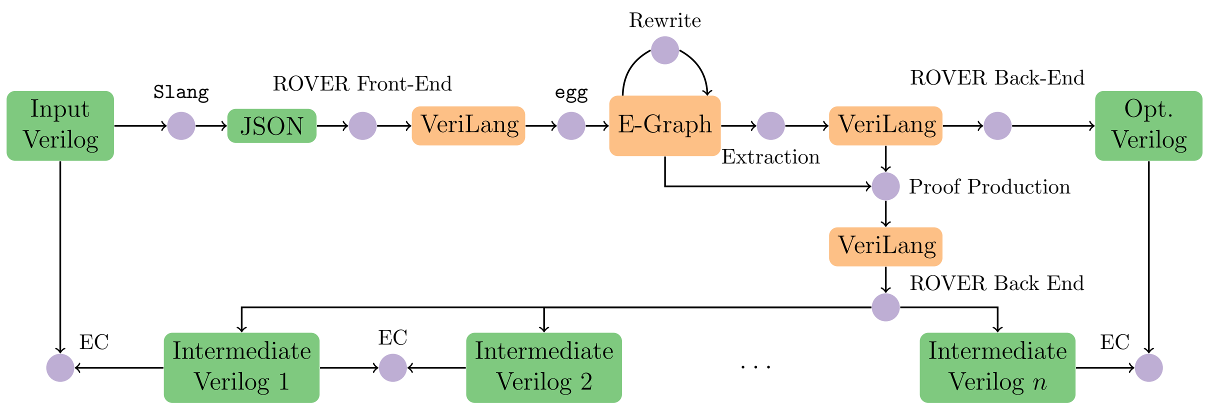


Figure 3.1: Flow diagram describing the operation of ROVER. The intermediate RTL designs are formally verified to be functionally equivalent using EC forming a chain of reasoning. The orange boxes denote the novel contributions.

This chapter is organized as follows. Section 3.1 describes an intermediate language that facilitates the representation of RTL as an e-graph. Then, Section 3.2 describes a set of mixed precision rewrites that encode manual RTL optimizations, and capture the dependence between circuit area and bitwidth. A data-driven approach to synthesizing rewrite validity conditions is also introduced. Section 3.3 describes how the minimal circuit area design is extracted from the generated e-graph, capturing downstream logic synthesis optimizations to ensure better correlation. A robust, proof decomposition based, verification methodology is described in Section 3.4. The approach is evaluated through the development of an RTL optimization tool, ROVER, capable of matching matching manual RTL design and correctly encoding the downstream logic synthesis capabilities. Figure 3.1 provides an overview of the ROVER tool flow. Sections 3.5 and 3.6 conclude with the results of applying ROVER to a range of arithmetic circuit design benchmarks.

The work described in this chapter was first published at ARITH in 2022 [119], and was later followed by a 2024 journal extension published in TCAD [120]. The primary contributions described in this chapter are:

- the 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,

Table 3.1: 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 capture downstream optimizations. The "Component Architecture" describes the implementation assumed by ROVER when estimating the cost of each operator. For example, all adders are assumed to be implemented using a prefix adder. During logic synthesis an alternative implementation may be chosen.

Operator	Symbol	Arity	Component Architecture
Add/Sub	+/-	2	Prefix Adder (PA) [41]
Negation	-	1	PA
Multiplication	$\times$	2	Booth Radix-4 [22]
Reduce	&, |, ^	1	Log Tree
Inverse Reduce	$\sim$&, $\sim$|, $\sim$^	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

- an automated method to generate necessary and sufficient conditions for RTL rewrites using an EC,
- a robust method to verify the correctness of the generated RTL based on automated proof decomposition.

3.1 Intermediate Representation

RTL exploration via e-graph rewriting is facilitated by an intermediate language, VeriLang, which when combined with a parser and generator permits translation to and from Verilog/System Verilog [2]. Since e-graphs work with expressions, VeriLang is a nested S-expression language in Common Lisp [121]. A formal description is given in Grammar 3.1.

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

$$\texttt{(+ 9 8 unsign x 8 unsign y)}. \tag{3.1}$$

The VeriLang semantics interprets `terms` as integers.

$$[\![\cdot]\!] : \texttt{term} \to \mathbb{Z} \tag{3.2}$$

The semantics are then defined in terms of integer arithmetic:

$$[\![(op\ w\ w_1\ s_1\ t_1\ \dots\ w_n\ s_n\ t_n)]\!] = \tag{3.3}$$

$$([\![op]\!]\ [\![t_1]\!]_{w_1,s_1}\ \dots\ [\![t_n]\!]_{w_n,s_n})_{w,\texttt{unsign}} \tag{3.4}$$

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

$$k_{w,s} = \begin{cases} k \mod 2^w, & \text{if } s == \texttt{unsign} \\ 2(k \mod 2^{w-1}) - (k \mod 2^w), & \text{otherwise.} \end{cases} \tag{3.5}$$

This is a valid model of bitvector arithmetic under the least positive residue definition of modulus [122].

$$\cdot \mod \cdot : \mathbb{Z} \times \mathbb{N} \to \mathbb{N} \tag{3.6}$$

Under these semantics, (3.1) has the following interpretation:

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

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 there is no 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* [2]. Therefore VeriLang includes a bitwidth annotation for the output of an operator

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

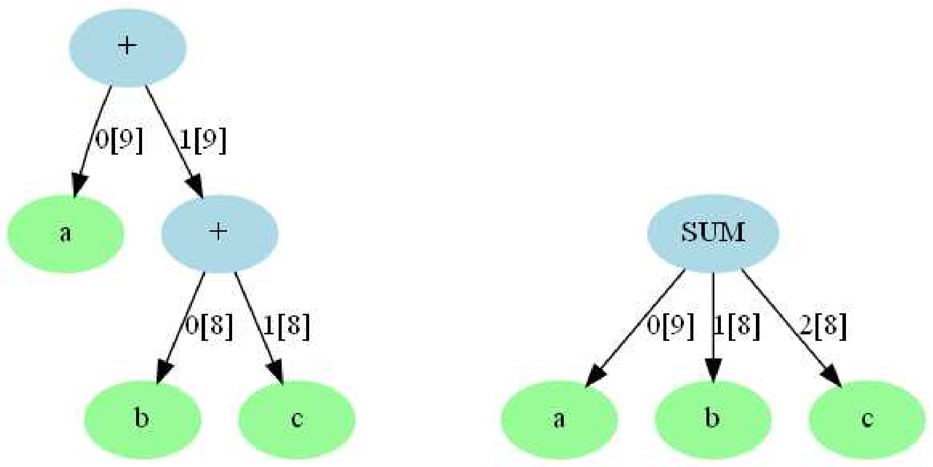
Grammar 3.1: VeriLang grammar definition. The terminal variable *var* is a symbol drawn from a set of expression variables, and *op* is an operation from the supported set of VeriLang operators as described in Table 3.1.

in VeriLang. Only the subset of VeriLang expressions comprised of concrete instances 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. Section 3.2 describes 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 VeriLang supports 29 of the Verilog defined operators as shown in Table 3.1, 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 3.1, 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 [21]. 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 3.2 shows two consecutive additions being reduced to a single `SUM` node. VeriLang includes two further merged operators, the familiar Fused Multiply-Add (FMA), which encodes the ability to construct the circuit for



(a) Consecutive adders (b) Merged adders encoded as `SUM`

Figure 3.2: There is no datapath leakage as the intermediate adder result retains the potential carry-out, therefore the adders can be merged. Edge labels show the operand's index and bitwidth in square brackets.

$a * b + c$ 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 [21]. The Muxed Mult Array will be discussed further in Section 3.2.

As shown in Figure 3.1, the input Verilog/System Verilog is first parsed by the open-source `slang` parser [123], 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, ROVER constructs a mapping from the original variable names to their corresponding VeriLang expressions. By attaching the variable name to the corresponding e-classes, ROVER retains information from the original RTL, which can be used during code generation (Section 3.3.3) to improve readability.

3.2 Rewrites

3.2.1 Specifying Rewrites

Rewrites define local equivalences between two expressions that, when chained, enable architectural exploration. Equivalence is defined as functional equivalence over `term`s in VeriLang. Namely, given `term`s 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 `term`s 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 2.4, `egg` matches a `term` in the e-graph returning a substitution that is an assignment of the variables in the `term` to e-classes. Following the VeriLang semantics ROVER processes this substitution, producing a `map` that is an assignment of (some of the) variables in the `term` to concrete values, mostly bitwidth and signage parameters. A partial evaluation of a `term` with respect to a `map` produces a new `term`,

$$[\![\cdot]\!]_{\cdot} : \texttt{term} \times \texttt{map} \rightarrow \texttt{term}.$$

As a first example, consider the unconditional commutativity rewrite that is always valid. Later examples will describe conditional rewrites. Commutativity of addition is defined as:

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

If applied to an e-graph containing (3.1), the e-matching process would return a `map`,

$$m = \begin{cases} w & \mapsto 9 \\ w_a & \mapsto 8 \\ s_a & \mapsto \texttt{unsign} \\ w_b & \mapsto 8 \\ s_b & \mapsto \texttt{unsign} \end{cases} \tag{3.7}$$

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 $\times_{32}$ [72], but in our setting this is impractical due to the large number of operators that would have to be supported. 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. Meaningful RTL transformations are defined via a set of conditionally applied rewrites specified as a triple (`cond`, `term`, `term`), where

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

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

$$\phi(m) \Rightarrow [\![lhs]\!]_m \cong [\![rhs]\!]_m. \tag{3.8}$$

Figure 3.3 provides 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

```
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;
```

Figure 3.3: Verilog associativity rewriting example. Signals `left1` and `right` are functionally distinct, because the carry-out is discarded in computing `add_8bit`, therefore `left1`$\not\to$`right`. The signals `left2` and `right` are functionally equivalently, therefore it is valid to rewrite `left2`$\to$`right`.

has sufficient precision to retain 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. Section 3.2.2 describes the construction of the conditions and returns to this example to construct a condition for this exact associativity rewrite.

The set of rewrites described in Table 3.2 captures optimizations learnt from Intel's Graphics Hardware Group, prior work [24] and logic synthesis documentation [23, 21]. All rewrites include the type annotations described in Section 3.1. No restrictions on the bitwidth and signage parameters are imposed in the rewrites, to ensure maximum generality of the rewrites. In Table 3.2 bitwidth and signage annotations as well as the conditions are omitted 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

Table 3.2: 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. To demonstrate the complexity, the condition for associativity of addition is given in Appendix B.

Class	Name	Left-hand Side	Right-hand Side	Condition
Bitvector Arithmetic	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	$a + 0$	$\texttt{slice}(a)$	†
	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	$(a \times b) + b$	$(a + 1) \times b$	†
Bitvector Logic	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
	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	$\sim \{a, b\}$	$\{(\sim a), (\sim b)\}$	†
Arithmetic Logic Exchange	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	$e?(a * b) : (c * d)$	$(e?a : c) * (e?b : d)$	†
	Sel Add Zero Left	$e?(a + b) : c$	$(e?a : c) + (e?b : 0)$	†
	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	$-a$	$(\sim a) + 1$	†
Merging Ops	Merge Additions	$a1 + (a2 + (a3 + ... + an)...)$	$\texttt{SUM}(a1, a2, ..., an)$	†
	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	$1 \times x$	$(2 \times x) - x$	†

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.* [24] 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. ROVER extends prior work on this subject [24], 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 3.1, allows ROVER to identify sub-circuits that synthesis tools will specifically optimize [23]. 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 [29]. 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 $b[i]$ represent bit i of b 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:

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

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, greatly helping to address the toolchain correlation problem.

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. Section 3.5 will describe such results, but will also demonstrate limitations of a rewriting approach for complex MCM problems. These rules allow ROVER to re-create and generalize results from the MCM literature described in Section 2.1. As in previous `egg` implementations, constant folding is implemented as an e-class analysis [15].

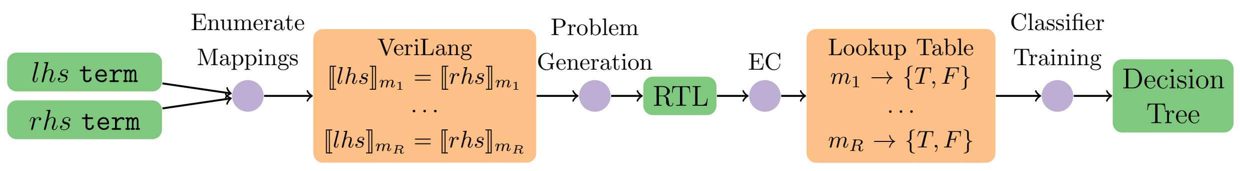
3.2.2 Synthesizing Rewrite Conditions

As described above, rewrites are encoded as triples $(\texttt{cond}, \texttt{term}, \texttt{term})$, where the `term`s may contain variable `width` and `signage` parameters. Not all assignments to these parameters produce valid rewrites, meaning the rewrite should be conditionally applied. Namely, in general, for rows in the table with † conditions, there exist mappings m such that $[\![lhs]\!]_{\texttt{m}} \ncong [\![rhs]\!]_m$. Figure 3.4 describes an automated condition synthesis flow, that greatly simplifies the addition of new rewrites to ROVER. The proposed flow automatically constructs a solution to the following problem. Given a pair of `term`s, (lhs, rhs), construct a `cond`, ϕ, such that for all maps m,

$$\phi(m) \Leftrightarrow [\![lhs]\!]_m \cong [\![rhs]\!]_m. \tag{3.9}$$

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 (3.9) is challenging since the space of all mappings is infinite. The data-driven condition synthesis approach used here does not guarantee that the generated ϕ satisfies (3.9), however any invalid rewrite applications are detected via a backend verification flow that will be described in Section 3.4.

Developers or design engineers can specify new ROVER rewrite rules as pairs of `term`s and run ROVER's condition synthesis flow to automatically generate a correct `cond`. This allows design



lhs term
rhs term
Enumerate Mappings
VeriLang
$[\![lhs]\!]_{m_1} = [\![rhs]\!]_{m_1}$
$\dots$
$[\![lhs]\!]_{m_R} = [\![rhs]\!]_{m_R}$
Problem Generation
RTL
EC
Lookup Table
$m_1 \rightarrow \{T, F\}$
$\dots$
$m_R \rightarrow \{T, F\}$
Classifier Training
Decision Tree

Figure 3.4: Flow diagram for the automated process of synthesizing rewrite conditions. The output is a decision tree that is translated into a Boolean expression.

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 [124], 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.

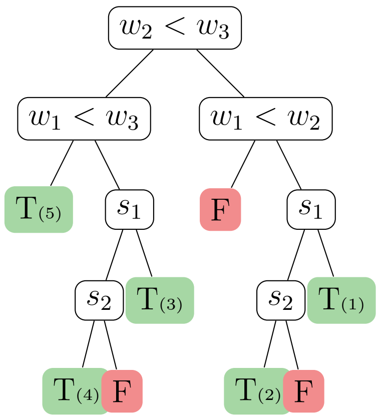
$$M = \{w_1 \mapsto \texttt{w1}, \dots w_H \mapsto \texttt{wH}, s_1 \mapsto \texttt{s1}, \dots s_G \mapsto \texttt{sG}$$
$$\mid \texttt{wi} \in \{1, \dots, 8\} \wedge \texttt{si} \in \{\texttt{unsign}, \texttt{sign}\}\} \,.$$

The flow enumerates the entire parameter space, M, constructing VeriLang expressions $[\![lhs]\!]_m$ and $[\![rhs]\!]_m$ for all $m \in M$, and determines, for each $m \in M$, whether these terms are equivalent. ROVER converts both $[\![lhs]\!]_m$ and $[\![rhs]\!]_m$ to Verilog then deploys commercial RTL EC as an oracle. This enables the re-use of the RTL generation framework (see Section 3.3.3) 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

$$T(m) = \begin{cases} \text{true}, & \text{if } [\![lhs]\!]_m \cong [\![rhs]\!]_m \\ \text{false}, & \text{otherwise.} \end{cases} \tag{3.10}$$

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

$$
\begin{array}{l}
(+\; w_3\; w_2\; s_2\; (+\; w_2\; w_1\; s_1\; \mathbf{a}\; w_1\; s_1\; \mathbf{b})\; w_1\; s_1\; \mathbf{c}) \rightarrow \\
(+\; w_3\; w_1\; s_1\; \mathbf{a}\; w_2\; s_2\; (+\; w_2\; w_1\; s_1\; \mathbf{b}\; w_1\; s_1\; \mathbf{c}))
\end{array}
$$



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

Figure 3.5: 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.

this ROVER fits a decision tree classifier [125] to determine a predicate, ϕ, such that

$$\forall m \in M,\ \phi(m) = T(m). \tag{3.11}$$

ROVER uses Python's sklearn library implementation to fit a decision tree classifier. The classifier learns based on Boolean features (3.12)-(3.17).

$$\begin{aligned} i = 1 \dots m, \quad & s_i == \texttt{unsign} & (3.12)\\ i, j, k = 1 \dots n, i \neq j \neq k, \quad & w_i == w_j & (3.13)\\ & w_i < w_j & (3.14)\\ & w_i \pm 1 < w_j & (3.15)\\ & w_i + w_j < w_k & (3.16)\\ & w_i + 2^{w_j} < w_k & (3.17) \end{aligned}$$

These features are relevant for the operators supported in VeriLang. For example, (3.14) indicates whether an addition of w_i-bit integers stored in a w_j-bit signal will retain a carry-out. Similarly, (3.16) relates to a multiplication of a w_i-bit integer and a w_j-bit integer stored in a w_k-bit signal. Lastly, (3.17) 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 (3.11) corresponding to zero classification error on the training set. Figure 3.5 takes a restricted associativity of addition rewrite as an example, where the variables a, b and c 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 3.4 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 3.5, 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 (3.11), 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 3.5. The minimum depth classifier satisfying (3.11) 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. This generates $|M| = 8^5 \times 2^4 = 2^{19}$, leading to a depth 9 decision tree classifier. For large enumeration spaces, a sampling approach may be necessary. The condition derived from this decision tree is given in Appendix B.

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 3.5 to an e-graph corresponding to the Verilog shown in Figure 3.3, e-matching detects two potential rewriting opportunities and returns two maps m_1 and m_2 corresponding to the expressions for `left1` and `left2`, respectively. The difference is highlighted in red.

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

Evaluating the `cond`, ϕ, shown in Figure 3.5

$$\phi(m_1) = \text{false} \qquad \phi(m_2) = \text{true}. \tag{3.18}$$

This agrees with the validity statements made in Figure 3.3. Note that $m_1, m_2 \notin M$, providing an example where the condition extrapolation is valid.

Since ROVER supports Verilog with signals exceeding 8-bit integers (the limit of the training data), ROVER extrapolates 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 were not observed in practice, are detected by the back-end verification, described in Section 3.4, preventing ROVER from delivering functionally incorrect RTL.

3.3 Extraction and Back-End

ROVER applies rewrites to the e-graph until saturation (defined in Section 2.4) 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. In this chapter, ROVER selects the minimum area design according to a theoretical area metric.

3.3.1 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 ROVER fixes a particular architecture from amongst the various possibilities. These architecture choices are described in Table 3.1 and are representative of operator architectures implemented by commercial synthesis tools [21]. When at least one operand is constant a different, constant specific, cost is used, 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 HLS tools use call-outs to logic synthesis engines to evaluate different circuit designs [6]. Such an approach is more computationally intensive thus limiting design space exploration. Section 3.6 evaluates the effectiveness of the theoretical cost metric.

3.3.2 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 yields sub-optimal solutions for minimal circuit area. An optimal circuit area design is extracted by casting extraction as an ILP problem, similar to the approach in prior work that optimized linear algebra expressions [111]. The ILP encoding correctly counts the cost of an operator once, irrelevant of how many times the resulting signal is utilized.

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 $n \in \mathcal{N}$, 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 (3.19). The program constraints ensure that a valid circuit description is extracted. The first constraint (3.20) ensures that at least one node from all child e-classes of a selected node is implemented. The final constraint ensures that for all output expressions found in the set of e-classes $\mathcal{S}$, the generated circuit produces that output.

$$\text{minimize: } \sum_{n\in\mathcal{N}} \text{cost}(n)x_n \text{ subject to:} \tag{3.19}$$

$$\forall(n,c)\in E.\ \sum_{n'\in\mathcal{N}_c} x_{n'} \geq x_n \tag{3.20}$$

$$\forall c\in\mathcal{S}.\ \sum_{n\in\mathcal{N}_c} x_n = 1. \tag{3.21}$$

Since e-graphs may contain cycles additional topological sorting variables associated with each class t_c are included. Let N denote the number of e-classes and $\mathcal{C}(n)$ be the e-class containing node n. The constraint (3.22) ensures that the output expression is acyclic.

$$\forall(n,k)\in E \quad t_{\mathcal{C}(n)} - Nx_n - t_k \geq 1 - N \tag{3.22}$$

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 ROVER deploys the CBC solver [126]. The ILP solution corresponds to a single VeriLang expression, that is a minimal circuit area implementation according to the theoretical area metric.

3.3.3 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 2.4. 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. Each e-class is assigned 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 signal assignment in the generated System Verilog. 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 user defined 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.

3.4 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 [86]. Debugging inconclusive proofs can be time consuming for verification engineers. To provide a robust verification flow, ROVER uses the `egg` proof production feature [107] described in Section 2.4, 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 3.1. 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. Whilst each of these steps is trivial to prove for the EC tool, verifying the combination of many transformations can yield inconclusive proofs. Section 3.5.4

demonstrates the value of proof decomposition. ROVER generates both the RTL and proof scripts, providing a proof certificate to the user which can be re-run to verify the RTL.

3.5 Results

ROVER has been used 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. The effectiveness of ROVER's datapath clustering optimizations are evaluated by studying the synthesis reports. Using the approach described in Section 3.4 the functional equivalence of the original and optimized architectures is verified. Each pair of designs is compared at two points along the area-delay trade-off curve using logic synthesis. The results for all benchmarks are summarized in Table 3.3. Figure 3.6 plots the complete area-delay profile comparing the original and ROVER optimized Media Kernel implementations across the delay spectrum. Table 3.3 compares the designs 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 3.6. 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 3.6. The evaluation will primarily focus on the area and delay impact since the cell count and power measurements are proportional to the area in this work.

The results are separated into two contributions. Firstly, ROVER is evaluated on a set of general RTL benchmarks. Then we will see how ROVER can optimize different instances of parameterizable RTL, generating a suite of tailored implementations.

3.5.1 Benchmark Selection

Each benchmark is comprised of a single module, implementing a datapath circuit using the ROVER supported operators described in Table 3.1. Typically, only designs containing multiple

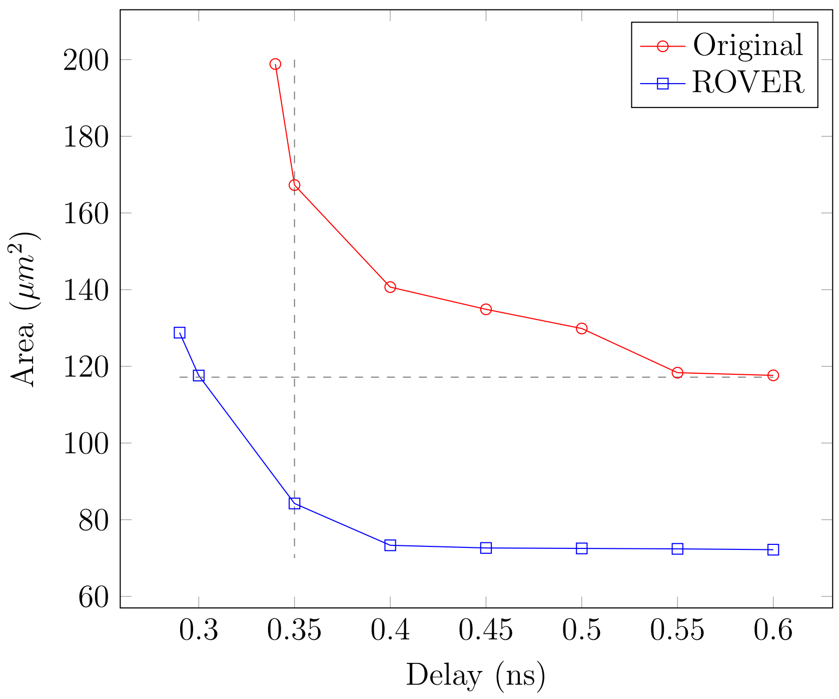


Figure 3.6: Area-delay profiles for the original and ROVER optimized Media Kernel designs. The dashed grey lines indicate the minimum area and delay comparison points used in Table 3.3.

sequential operations are amenable to the dataflow graph manipulations performed by ROVER, motivating the benchmark selection. The first two benchmarks are closed-source Intel designs taken from broader units that my Intel team were asked to optimize. These designs were easy for ROVER to consume, lacking additional System Verilog constructs such as generate loops, and exhibited datapath rewriting opportunities. The remaining benchmarks are all publicly available. The FIR Filter Kernel and ADPCM Decoder are taken directly from [24], which is the most relevant prior work. The Shifted FMA and Shift Mult are toy examples constructed for this evaluation, inspired by datapath optimizations performed in industry, which concisely demonstrate where ROVER exceeds [24]. The functional description of Shifted FMA is given below in (3.23) and the Verilog implementation of Shift Mult is given in Figure 3.7. The last collection of benchmarks are all taken from existing work on the MCM problem [127] and are included to demonstrate ROVER's ability to construct general solutions to specific problems. The area optimization approach presented in this chapter will be used as a baseline in Chapters 5 and 6 to evaluate the additional benchmarks listed in Appendix A.

Table 3.3: 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	0.12	620	197.4	41.8	**556**	**190.6**	**38.0**	-9%	20.8	**0.84**	**0.84**	0%
	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**	0%

Table 3.4: 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

3.5.2 Exploiting Datapath Optimizations

The first set of benchmarks in Table 3.3 are Intel RTL designs. The first benchmark is a kernel from the Intel media module. ROVER is able to automatically optimize the design and achieve comparable results to manual optimization by an RTL expert, discovering the opportunity to merge two multiplication arrays into a single array using the "Merge Mult Array" rewrite. The reports generated by the synthesis tool highlight the source of the area reduction when compared against the original baseline. 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 2.1) 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 the code contains constants, a fact the analysis performed by logic synthesis fails to capture because it has limited understanding of 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.* [24] 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 [24], 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 3.8a). Via the "Arithmetic Logic Exchange" rewrites, ROVER explores all the alternative arithmetic clustering opportunities extracting

```
module spec(A,B,M,N,O);
  input  [15:0] A, B;
  input  [ 3:0] M, N;
  output [62:0] O;
  wire   [30:0] D, E;

  assign D = A << M;
  assign E = B << N;
  assign O = D * E;
endmodule
```

Figure 3.7: Shift Mult benchmark implemented in Verilog. The design first shifts the two inputs then performs a multiplication.

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, potentially, 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 synthesis 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 [24]. Shifted FMA implements a simple circuit.

$$(a \times b) \ll S + c \tag{3.23}$$

ROVER 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, logic synthesis greedily clusters, such that it must perform a shift of a value represented in carry-save format. 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, therefore does not discovers this opportunity.

These ROVER optimizations are not reachable by [24], since their tool did not explore the interaction between multiplication and logic.

The “Constant Expansion” rewrites are valuable for the MCM benchmarks, where

$$MCM(a_1, a_2, ..., a_n) = \{a_1 \times x, a_2 \times x, ..., a_n \times x\}. \tag{3.24}$$

MCM(3,7,21) is an example taken from [27]. ROVER is able to match the operator count from [27], 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 3.3. 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, the logic synthesis tool has all datapath optimizations enabled to provide a baseline, leveraging state-of-the-art datapath optimization techniques. To quantify the significance of the datapath optimizations built in to the logic synthesis tool and those performed by ROVER, these optimizations were disabled and the designs were synthesized again. 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. These results are included to demonstrate how ROVER advances the state of the art, which already incorporates powerful optimizations.

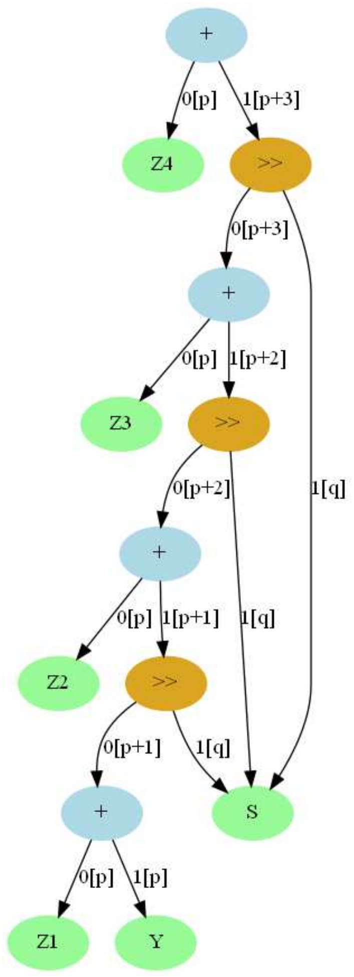
[1]Thank you to an anonymous reviewer of our journal paper for providing this benchmark.

3.5.3 Bitwidth Dependent Architectures

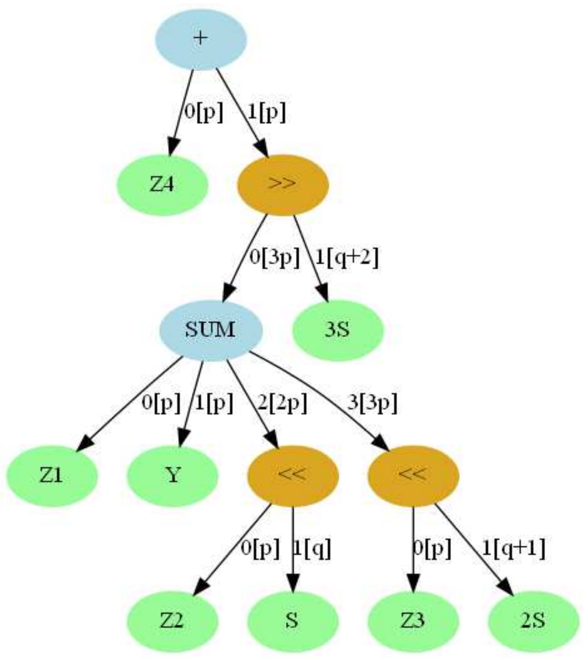
This section considers parameterizable RTL designs. As the complexity of integrated circuits grows, reusable and parameterizable hardware has become increasingly popular amongst engineers and architects as it facilitates faster development. Each instance of this RTL will be synthesized using the same 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, a 3-tap FIR filter with parameterizable input bitwidths was considered. Increasing the input bitwidth parameter from 4 to 64 allowed ROVER to explore the design space for each parameterization. As shown in Figure 3.8, ROVER extracted one of three distinct architectures. In the FIR kernel 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 and three logical shifts, Architecture 1 uses two carry-propagate adders and three logical shifts, whilst Architecture 2 uses only a single carry-propagate adder at the expense of one additional logical shift. ROVER automatically detects the point at which these trade-offs becomes favorable as bitwidth is increased.

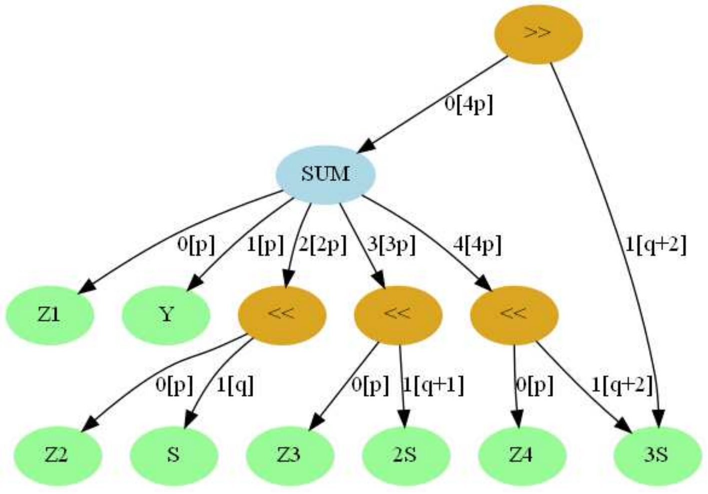
For each bitwidth, Architecture 0 and the distinct ROVER-generated RTL (which implements either Architecture 0, 1 or 2) are synthesized at the minimum delay target that both can meet. Figure 3.9 plots synthesis results at each bitwidth comparing against the baseline, Architecture 0 (Figure 3.8a). The 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 Section 3.6. Using ROVER to automatically generate an optimized design for each parameterization allows engineers to avoid manual customization without sacrificing IP quality.



(a) Architecture 0 {4,8}



(b) Architecture 1 {12,...,24}



(c) Architecture 2 {28,...,64}

Figure 3.8: Simplified FIR filter dataflow 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.

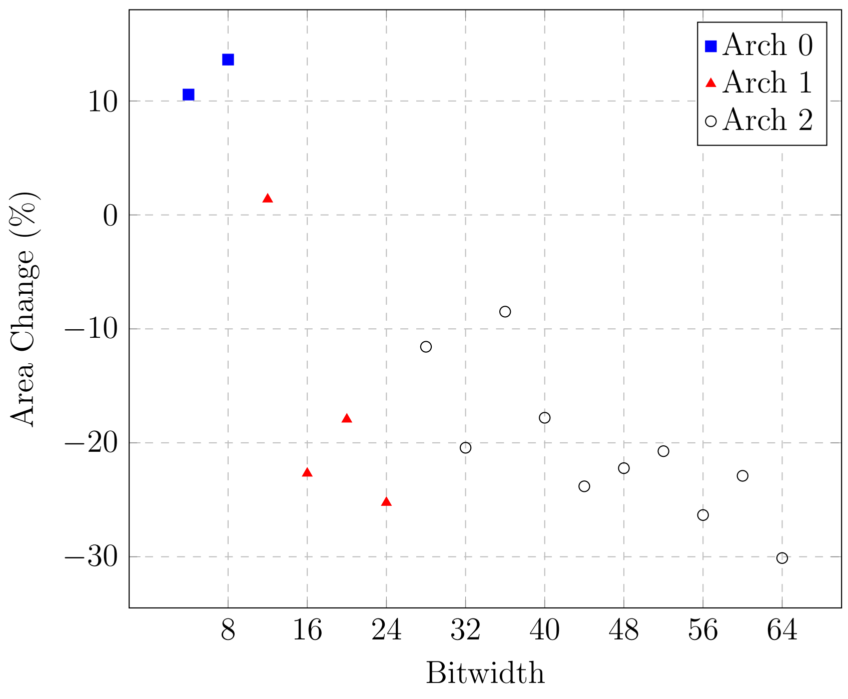


Figure 3.9: Synthesis results for the 3-tap FIR kernel at a range of different bitwidths. Both the ROVER-generated RTL and original RTL (Architecture 0) are synthesized with a minimum delay objective. The relative change in area and delay against the baseline is plotted.

3.5.4 Performance

Table 3.4 presents benchmark properties and optimization statistics. ROVER was run on SLES 12 on modern Intel Xeon CPUs. Since the evaluation does not compare runtimes against alternative approaches, a single runtime result is reported that does not account for small run-to-run variations. The ILP extraction method uses 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 large scale problems can typically be naturally partitioned. The faster greedy `egg` extraction method [15] was used for the ADPCM decoder since there was no scope to exploit common sub-expressions in this benchmark. Extraction method selection is left as a user specified option for ROVER. 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 and the type of operators it contains.

Highlighting the importance of the verification flow, for the Media Kernel and Shift Mult benchmarks, the commercial 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 3.4.

3.6 Cost Metric Evaluation

The primary objective of the theoretical cost metric is to steer the extraction process in order to generate an optimized architecture. Before determining the accuracy of a cost estimate, it is necessary to consider inherent variability of the logic synthesis process. Results such as those observed in Section 3.5.3, showed that in logic synthesis 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. In fact, VeriLang is unable to express such changes. The evaluation of the logic synthesis noise floor used an approach known as performance fuzzing [128, 129], that differs from the more traditional application of fuzzing to automated bug detection [128]. By automatically generating random mutations to a program, the variation in the results can be measured. Fuzzing the RTL allows two types of semantics-preserving mutations: variable renaming and swapping the order of always/assign blocks [2] in the code, modifications which one would not expect to have a meaningful impact on synthesis results. Variability of the results for the Media Kernel and 3-tap FIR Filter, synthesizing 30 fuzzed designs in each case at relevant delay targets, is shown in Figure 3.10.

Figure 3.11 highlights how noise can affect the choices made in Section 3.5.3. For 12-bit inputs the synthesis results for fuzzed Architectures 0 and 1 overlap, with Architecture 1 generating lower area on average whilst Architecture 0 obtains the minimum area. This noise is not captured by ROVER, as these fuzzed designs are identical when represented in VeriLang. Applying this to other bitwidth inputs there are cases where there is clearly an optimal choice.

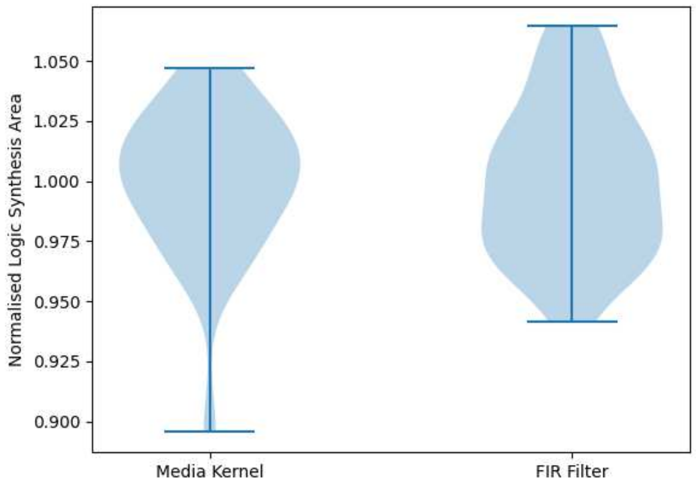


Figure 3.10: A violin plot depicting the logic synthesis area results for 30 fuzzed designs of the Media Kernel and the 3-tap FIR Filter at a 0.5ns delay target. For each violin, the area results are normalized by the mean.

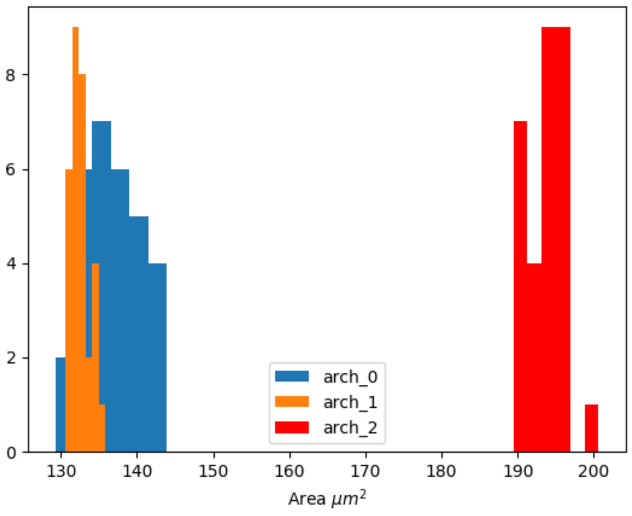


Figure 3.11: Histogram plot of logic synthesis area results for 30 fuzzed designs for each of the three FIR Filter architectures (Fig 3.8).

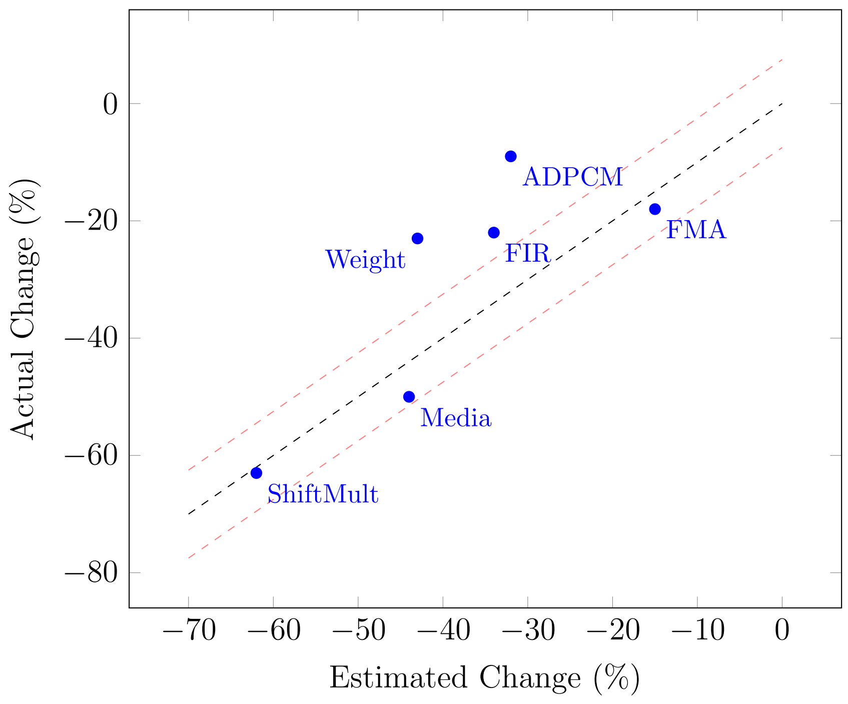


Figure 3.12: ROVER's predicted percentage change vs. the actual percentage change based on logic synthesis at the minimum delay target. Points above/below the diagonal indicate that ROVER over/under-predicts the area reduction. The MCM results are omitted. Red lines represent the synthesis noise window.

These results show up to a 15% difference in logic synthesis area, which ROVER's cost model cannot be expected to capture. Given its randomness, the variability is equally likely to benefit ROVER as it is to be detrimental for the results shown in Table 3.3. However, the overall benefit demonstrated by ROVER is statistically significant and explainable.

The accuracy of the cost model is evaluated by plotting the ROVER estimated circuit area reduction against the actual change seen in the logic synthesis results at the minimum delay target in Figure 3.12. 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, it is clear that the synthesis tool is already capable of manipulating this design to cluster the arithmetic operations limiting the observable benefit of "optimizations" performed by ROVER. 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.

3.7 Summary

By decomposing RTL optimization into a sequence of local-equivalence preserving transformations, engineers and automated tools can realize substantial circuit area reductions. This chapter lays the foundations of an RTL rewriting framework, leveraging equality saturation, and develops ROVER, an RTL optimization tool. Through a set of bitwidth dependent rewrites and custom operators, ROVER is able capture downstream synthesis optimizations. The equality saturation approach avoids any need to specify an order in which to apply transformations whilst maintaining bit-accurate functionality. ROVER's ability to generate an accompanying proof certificate ensures that engineers can deploy the ROVER-generated designs with confidence.

Automated rewriting techniques help engineers, allowing them to defer bug prone circuit area optimization to a tool that can generate verified implementations. The e-graph representation of RTL, rewrite synthesis framework and verification methodology form the foundations on which Chapters 5, 6 and 7 build.

Chapter 4

Combining Equality Saturation with Abstract Interpretation

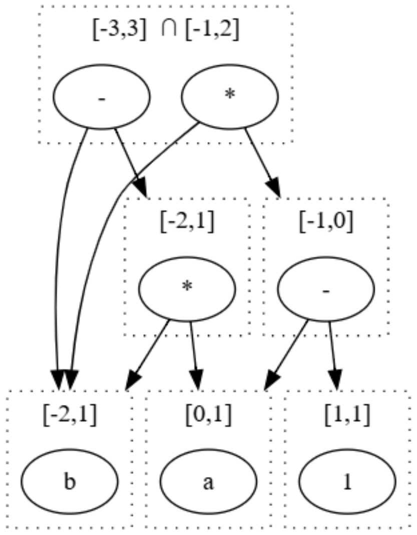
The previous chapter described how rewriting could be used to optimize a circuit design. However, rewriting alone is not capable of expressing every optimization that a skilled engineer may implement. Consider the following Verilog example.

```
input  wire [3:0] a, b, c, d;
       wire [4:0] add;
       wire [7:0] mul;
output wire [8:0] res;

assign add = a + b;
assign mul = c * d;
assign res = add + mul;
```

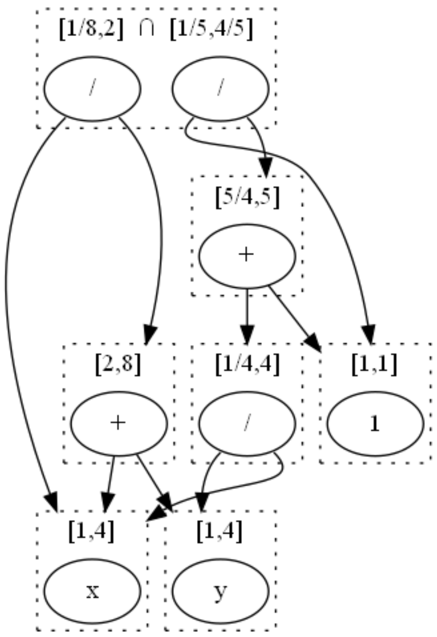
This implementation conservatively assigns the result of the final addition to a 9-bit result. This bitwidth determination follows the natural rules; the result of a multiplier occupies twice the bitwidth of its inputs and the result of an addition may produce a carry-out. However, by manually analyzing the arithmetic we can see that the maximum value of the result is $15 \times 15 + 15 + 15 = 255$. This can be stored using only 8-bits, highlighting an optimization opportunity. To automatically detect such optimization opportunities it is necessary to perform some computation, e.g. calculating the maximum value, to discover additional properties of the design. These properties cannot be determined from the syntax alone, suggesting that datapath optimization is not purely a rewriting problem. As described in Section 2.3, program analysis is the process of analyzing the behavior of a program (or a circuit) to learn useful properties. Using program analysis techniques it will be possible to automate the optimization described here and many more.

Program analysis is an essential component of many automated optimization and verification tools. In most automated tools, transformations and analyses are considered separately, yet recent work on phrasing compilation in terms of abstract interpretation (AI), a core theory underpinning much of program analysis, highlighted the strong interaction between them [96]. As ROVER began to incorporate program analysis, a similar advantage was observed when combining equality saturation and program analysis. The benefit comes from analyzing multiple equivalent representations, which help to mitigate the shortcomings of computationally efficient program analyses. To both simplify the exposition and demonstrate the broader applicability of the proposed formal framework, the remainder of this chapter will study real arithmetic expressions, rather than bitvector arithmetic. Despite the alternative setting, the theoretical results presented in this chapter will underpin the bitvector analyses that enable the deeper transformations performed throughout Chapters 5, 6 and 7.

Recall the example of two equivalent real arithmetic expressions provided in Section 2.3. Under the constraints $a \in [0, 1]$ and $b \in [-2, 1]$, an interval analysis of the two equivalent expressions, $a \times b - b$ and $(a - 1) \times b$, yields $[-3, 3]$ and $[-1, 2]$, respectively. An equality saturation optimizer may rewrite $a \times b - b \to (a - 1) \times b$, such that they reside in the same e-class, as shown in Figure 4.1a. Intuitively, if we now lift the interval analysis from expressions to e-classes of expressions, we know two facts about the expressions in that e-class. First, any expression in the e-class produces outputs in the range $[-3, 3]$, but also by equivalence, any expression in the e-class produces outputs in the range $[-2, 1]$. This chapter will show that these facts can be combined via their interval intersection since it is also true that any expression in the e-class produces outputs in the range $[-3, 3] \cap [-2, 1] = [-2, 1]$, providing a more precise abstraction



(a) An e-graph containing two equivalent expressions $a \times b - b$ and $(a-1) \times b$.



(b) An e-graph containing two equivalent expressions $\frac{x}{x+y}$ and $\frac{1}{1+(y/x)}$.

Figure 4.1: Each expression contains two variables subject to input range constraints, $a \in [0,1]$, $b \in [-2,1]$ and $x, y \in [1,4]$. The input intervals are propagated through the e-graph via an e-class analysis (described in Section 2.4.2). In the root e-classes two interval abstractions are combined taking an intersection.

than analysis of the original expression yielded. An e-graph demonstrating this analysis is shown in Figure 4.1a.

The examples above demonstrated how program analysis enables circuit optimizations and how equality saturation can produce a more precise abstraction. Taking this one step further, the rewrites used in equality saturation can be conditionally applied based on the results of a program analysis. By interleaving conditional e-graph rewriting and program analysis the exploration power of the e-graph can be greatly enhanced. Figure 4.2 visualizes this positive feedback loop. Essentially, as new representations are added to the e-graph via rewriting, the analysis is refined, potentially facilitating the application of further conditional rewrites.

This chapter formalizes the theoretical connection between equality saturation and AI, a core theory underpinning much of program analysis. These concepts are required to produce an e-class analysis, which was introduced in Section 2.4.2. The e-class analysis feature allows `egg` users to attach analysis data to each e-class. This data can be accessed during e-graph

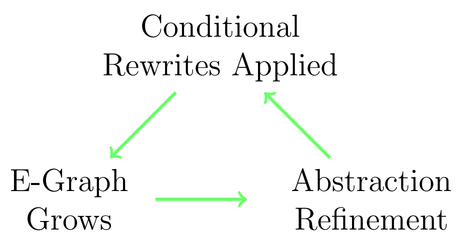


Figure 4.2: The positive feedback loop between e-graph exploration and abstraction refinement.

exploration to validate the applicability of conditional rewrites. Partitioning expressions into e-classes gives rise to a natural lattice-theoretic interpretation for AI, resulting in the generation of more precise abstractions. This chapter will use a simpler real-arithmetic setting for motivational examples and evaluation. As another motivational example, Figure 4.1b demonstrates how interval analyses of equivalent expressions are combined to produce tighter enclosing intervals. In this example, the expressions are actually only equivalent if x is always non-zero, a property that is true due to the interval constraints on x.

Section 4.1 develops the general theoretical underpinnings of AI on e-graphs, exploiting rewrites to produce more precise abstractions using a lattice-theoretic formalism. Section 4.2 demonstrates the benefits of combining equality saturation and AI, describing a general purpose real-arithmetic analysis tool that approximates the range of values an arbitrary arithmetic expression can take. The analysis capabilities of the tool are evaluated using the FPBench suite [130] in Section 4.3.

The work described in this chapter was published at SOAP in 2023 [131]. This chapter contains the following novel contributions:

- formalization of AI with e-graphs in lattice theory,
- relating fixpoints to e-graph cycles to automatically discover iterative abstract refinement methods,
- an interval arithmetic (IA) implementation with associated expression bounding results.

For simplicity of exposition, this chapter focuses on non-relational domains. However, it is important to note that the combination of non-relational domains with the *relational* information provided by rewrite rules provides a stronger analysis than classical non-relational domains. Chapter 5 introduces context into the e-graph and describe a mechanism to realize the abstraction refinements offered by this relational information.

4.1 Theory

4.1.1 Abstraction

From a theoretical viewpoint, AI [93] is concerned with relationships between lattices, defined via Galois connections.

Definition 4.1 (Lattice)**.** *A lattice is a partially ordered set (poset) $\langle L, \leq \rangle$, such that $\forall a, b \in L$ the least upper bound (join) $a \sqcup b$ and the greatest lower bound (meet) $a \sqcap b$ both exist. Meet- and join-semilattices only require the existence of the meet and join respectively.*

Definition 4.2 (Galois connection)**.** *Given a poset $\langle \mathcal{K}, \sqsubseteq \rangle$, corresponding to the concrete domain, and a poset $\langle \mathcal{A}, \preccurlyeq \rangle$, corresponding to the abstract domain, then a function pair $\alpha \in \mathcal{K} \to \mathcal{A}$, $\gamma \in \mathcal{A} \to \mathcal{K}$, defines a Galois connection iff*

$$\forall P \in \mathcal{K}.\ \forall \overline{P} \in \mathcal{A}.\ \alpha(P) \preccurlyeq \overline{P} \Leftrightarrow P \sqsubseteq \gamma(\overline{P}),$$

$$\textit{written } \langle \mathcal{K}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \mathcal{A}, \preccurlyeq \rangle.$$

The pair (α, γ) define the abstraction and concretization functions respectively, allowing us to over-approximate (i.e. abstract) concrete properties in $\mathcal{K}$ with abstract properties in $\mathcal{A}$.

Definition 4.3 (Sound abstraction [93])**.** *$\overline{P} \in \mathcal{A}$ is a sound abstraction of a concrete property $P \in \mathcal{K}$ iff $P \sqsubseteq \gamma(\overline{P})$.*

Consider expressions evaluated over a domain $\mathcal{D}$. By imposing a canonical ordering on the variable set, a defined subset $I \subseteq \mathcal{D}^n$, encodes any preconditions on the set of (input) variable values. Now consider a (concrete) semantics of expressions $[\![\cdot]\!]_\cdot \in \mathit{Expr} \to I \to \mathcal{D}$, where *Expr* denotes the set of expressions, so $[\![e]\!]_\rho$ denotes the interpretation of expression e under execution environment (assignment of variables to values) $\rho \in I$. Let $[\![e]\!] = \{[\![e]\!]_\rho \,|\, \rho \in I\}$. To clarify the notation, consider the following example. Let $\mathcal{D} = \mathbb{Z}$ and $I = \mathbb{N}^2$, restricting the two free variables a and b to be positive integers. If $\rho = \{a \mapsto 2, b \mapsto 3\}$, then $[\![a - b]\!]_\rho = 2 - 3 = -1$, and $[\![a - b]\!] = \mathbb{Z}$. The e-graph data structure encodes equivalence under concrete semantics, which can now be defined precisely using this notation.

Definition 4.4 (Congruence)**.** *Two expressions e_a and e_b are congruent, $e_a \cong e_b$, iff $[\![e_a]\!]_\rho = [\![e_b]\!]_\rho$ for all $\rho \in I$.*

Lemma 4.1. *If $e_a \cong e_b$ and $\overline{P}$ is a sound abstraction of $[\![e_a]\!]$, then $\overline{P}$ is a sound abstraction of $[\![e_b]\!]$.*

Proof. by definition of congruence. □

This lemma implies that a sound abstraction of one expression in an e-class is a sound abstraction of all expressions in the e-class. Precision refinement relies on the following, which is a specialization of the more general result [132].

Lemma 4.2. *For any two sound abstractions $\overline{P}_a$ and $\overline{P}_b$ of P, the meet $\overline{P}_a \sqcap \overline{P}_b$ is also a sound abstraction of P.*

Proof.

$$P \sqsubseteq \gamma(\overline{P}_a) \text{ (sound abstraction)} \Rightarrow \alpha(P) \preccurlyeq \overline{P}_a \text{ (Galois connection), and similarly}$$
$$P \sqsubseteq \gamma(\overline{P}_b) \text{ (sound abstraction)} \Rightarrow \alpha(P) \preccurlyeq \overline{P}_b \text{ (Galois connection)}$$

Therefore $\alpha(P) \preccurlyeq \overline{P}_a \sqcap \overline{P}_b$ (meet definition) and hence $P \sqsubseteq \gamma(\overline{P}_a \sqcap \overline{P}_b)$ (Galois connection). □

4.1.2 Application to E-graphs

Consider an e-graph. As in Chapter 3, let $\mathcal{C}$ denote the set of e-classes, and $\mathcal{N}_c$ the set of nodes in the equivalence class $c \in \mathcal{C}$. With each e-class, associate an abstraction $A \in \mathcal{A}$ and write $\mathcal{A}[\![c]\!] = A$. Interpreting an m-arity node n of function f with child classes $c_1, ..., c_m$, using an arbitrary sound abstraction $\bar{f}$:

$$\mathcal{A}[\![n]\!] = \bar{f}\left(\mathcal{A}[\![c_1]\!], ..., \mathcal{A}[\![c_m]\!]\right). \tag{4.1}$$

0-arity nodes are either constants with exact abstractions in $\mathcal{A}$ or variables with user specified abstract constraints.

For acyclic e-graphs, the known abstractions are propagated upwards using (4.1), taking the greatest lower bound (meet) across all nodes in the e-class.

$$\mathcal{A}[\![c]\!] = \bigsqcap_{n \in \mathcal{N}_c} \mathcal{A}[\![n]\!] \tag{4.2}$$

The propagation algorithm is described in Figure 4.3, where

$$\texttt{make}(n) = \mathcal{A}[\![n]\!] \text{ and } \texttt{meet}(A_1, A_2) = A_1 \sqcap A_2.$$

These functions are analogous to those described for an e-class analysis [15], but replace their `join` with a `meet`. Since Lemma 4.2 holds using a join, an e-class analysis can merge abstract elements using a `meet` or a `join` depending on whether the abstract poset $\langle \mathcal{A}, \preccurlyeq \rangle$ is a meet-semilattice or a join-semilattice. If $\langle \mathcal{A}, \preccurlyeq \rangle$ is a lattice, there is choice in how abstract elements are combined.

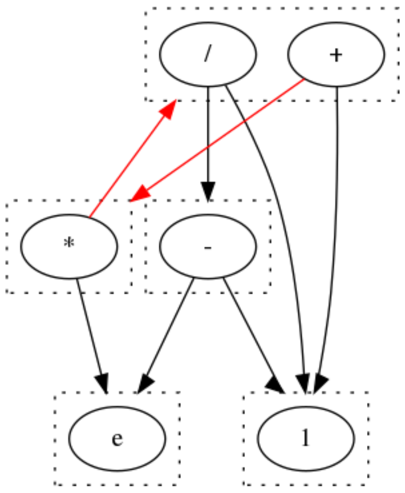
Lifting the abstract analysis from expressions to e-classes of expressions constructs a more precise analysis. In the abstract domain the notion of equivalence is different, $n_a, n_b \in \mathcal{N}_c \not\Rightarrow \mathcal{A}[\![n_a]\!] = \mathcal{A}[\![n_b]\!]$, which results in tighter abstractions since the `meet` corresponds to a more precise abstraction in $\mathcal{A}$. Figure 4.1 provided examples that demonstrate this precision refinement. In the algorithm in Figure 4.3, by initializing the `workqueue` with only the modified

```
workqueue = egraph.classes().leaves()

while !workqueue.is_empty()
  s = workqueue.dequeue()

  for n in s.nodes()
    skip_node = false

    for child_s in n.children()
      if child_s.uninitialized
        workqueue.enqueue(s)
        skip_node = true

    if skip_node
        continue
    elif s.uninitialized
        s.data = make(n)
        s.uninitialized = false
        workqueue.enqueue(s.parents())
    elif !(s.data <= meet(s.data,make(n)))
        s.data = meet(s.data,make(n))
        workqueue.enqueue(s.parents())
```

Figure 4.3: Pseudocode for abstract property propagation in an e-graph.

e-classes after application of a rewrite, the abstract properties of the e-graph can be evaluated on the fly. On-the-fly evaluation facilitates the application of additional conditional rewrites as more precise properties are discovered during construction. Section 4.3 quantifies the extent to which this enhances the reach of equality saturation.

A positive feedback loop is created by combining AI and e-graphs (Figure 4.2), enhancing the value of the interaction observed in [96]. A larger space of equivalent expressions is explored as more rewrites can be proven to be valid at exploration time. In turn, e-class abstractions are further refined by discovering more equivalent expressions, allowing even more valid rewrites, and the cycle continues. Further refinement comes from the fact that several equivalent expressions can contribute to the tight final abstraction. An interval arithmetic (IA) example of this is shown in Section 4.2, where one expression in the e-class contributes the tight lower bound whilst another *distinct* expression contributes the tight upper bound.



$$\frac{1}{1-e} \cong 1 + e * \frac{1}{1-e}$$

Figure 4.4: An example of a cyclic e-graph. The cycle is highlighted using red arrows.

4.1.3 Cyclic E-graphs and Fixpoints

Although rarely explored, cyclic e-graphs arise when an expression is equivalent to a sub-expression of itself with respect to concrete semantics. An example is shown in Figure 4.4. Given the possibility of introducing e-graph cycles, it is useful to define how the abstract elements propagate through a cyclic e-graph. Without loss of generality, let $e \cong e'$ where e appears as a subterm in e'. Let $f : \mathcal{D} \to \mathcal{D}$ be the interpretation of e' as a (concrete) function of $[\![e]\!]_\rho$, so that – in particular – $f([\![e]\!]_\rho) = [\![e]\!]_\rho$ due to the congruence and hence $[\![e]\!] = \{f([\![e]\!]_\rho) \mid \rho \in I\}$. Note that f may depend on other subterms, which have been absorbed into the function to simplify the notation. Abstracting f via a sound abstraction $\bar{f}$, yields the corresponding abstract fixpoint equation $a = a \sqcap \bar{f}(a)$ where the `meet` operation arises from (4.2) and a represents the abstract element associated with the e-class containing e.

Now consider the function $\tilde{f}(a) = a \sqcap \bar{f}(a)$. The decreasing sequence defined by $a_{n+1} = \tilde{f}(a_n)$ corresponds to applying the abstract property propagation around a cycle in the e-graph, given an initial sound abstraction a_0 of $[\![e]\!]$.

Lemma 4.3. *$\alpha([\![e]\!])$ is a fixpoint of $\tilde{f}$.*

Proof.

$$\begin{aligned} \alpha(\llbracket e \rrbracket) &= \alpha\left(\{f(\llbracket e \rrbracket_\rho) \mid \rho \in I\}\right) && \text{(congruence)} \\ &\preccurlyeq \bar{f}(\alpha(\llbracket e \rrbracket)) && \text{(sound abstraction)} \end{aligned}$$

Hence $\tilde{f}(\alpha(\llbracket e \rrbracket)) = \alpha(\llbracket e \rrbracket) \sqcap \bar{f}(\alpha(\llbracket e \rrbracket)) = \alpha(\llbracket e \rrbracket)$ (meet definition). □

Lemma 4.4. *a_n is a sound abstraction of $\llbracket e \rrbracket$ for all $n \in \mathbb{N}$.*

Proof. By induction, $\llbracket e \rrbracket \sqsubseteq \gamma(a_0)$ and assume $\llbracket e \rrbracket \sqsubseteq \gamma(a_n)$. $\llbracket e \rrbracket = \{f(\llbracket e \rrbracket_\rho) \mid \rho \in I\} \sqsubseteq \gamma(\bar{f}(a_n))$ (sound abstraction of f). Hence $a_{n+1} = a_n \sqcap \bar{f}(a_n)$ is a sound abstraction of $\llbracket e \rrbracket$ (Lemma 4.2) for all n. □

Collecting these results, for some fixpoint a^*

$$\alpha(\llbracket e \rrbracket) \preccurlyeq a^* \preccurlyeq \ldots \preccurlyeq a_1 \preccurlyeq a_0. \tag{4.3}$$

Thus computing abstractions around the loop refines the abstraction and is guaranteed to terminate if the lattice $\langle \mathcal{A}, \preccurlyeq \rangle$ satisfies the descending chain condition, as any finite abstract domain will [133]. Note that the fixpoint a^* is neither guaranteed to be greatest, least nor unique. This can be seen because the bottom element of the lattice is a fixpoint of $\tilde{f}$, but from (4.3) the computed fixpoint is bounded below by $\alpha(\llbracket e \rrbracket)$. Furthermore, for any $\bar{f}$ such that $\bar{f}(\top) = \top$, the top element of the lattice is also a fixpoint of $\tilde{f}$. The algorithm in Figure 4.3 will correctly apply abstract property propagation around loops, terminating if the sequence a_n converges in a finite number of steps. For abstract domains with infinite descending chains standard techniques such as widening/narrowing apply [134]. Section 4.3.3 shall demonstrate how e-graph cycles can be used to discover helpful iterative refinement loops.

4.2 Implementation

The theory described above is evaluated via an implementation of IA [135] for extended real valued expressions, $\mathcal{D} = \mathbb{R} \cup \{-\infty, +\infty\}$, as an e-class analysis using `egg` [15]. The implementation considers a concrete domain corresponding to sets of extended real numbers, i.e. $\mathcal{K} = \mathcal{P}(\mathcal{D})$ where $\mathcal{P}$ denotes the power set. Each expression is associated with a `binary64` (double precision) [80] valued interval (a finite abstract domain),

$$\mathcal{A} = \{[a,b] \mid a \leq b, a, b \in \texttt{binary64} \setminus \{\texttt{NaN}\}\} \cup \{\emptyset\}.$$

In this setting the abstraction and concretization functions are as follows (infima/suprema always exist in this setting as can always take $\pm\infty$):

$$\langle \mathcal{K}, \subseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \mathcal{A}, \subseteq \rangle \tag{4.4}$$

$$\alpha(X) = [\text{round\_down}(\inf X), \text{round\_up}(\sup X)] \tag{4.5}$$

$$\gamma([a,b]) = [a,b] \tag{4.6}$$

$$\alpha(\emptyset) = \emptyset, \gamma(\emptyset) = \emptyset \tag{4.7}$$

This work supports the following set of operators, +, -, $\times$, /, $\surd$, `pow`, `exp` and `ln`. Correctness is ensured through, the use of 'outwardly rounded IA' which conservatively rounds upper bounds towards $+\infty$ (round_up) and lower bounds towards $-\infty$ (round_down) [135, 136]. For the elementary functions, $\surd$, `exp` and `ln`, default library implementations are used. Since it is not possible to control the rounding mode, the implementation conservatively adds or subtracts one unit in the last place for upper and lower bounds respectively. If `NaN`s do not appear in the initial analysis of the input expression evaluation it is assumed they are not generated by the e-graph exploration. In the implementation, if a `NaN` occurs in the computation of the analysis it will halt execution, although this was not observed during the evaluation. Furthermore abstract intervals containing -0 shall be mapped by γ to sets containing $0 \in \mathcal{D}$.

The implementation uses an abstraction of a given function f, $\bar{f} = \alpha \circ f \circ \gamma$. For e-class c

Table 4.1: Additional IA optimization rewrites, beyond the standard arithmetic identities. Several rewrites are taken from [135].

Name	Left-hand Side	Right-hand Side	Condition
Subtract Common	$ab - b$	$(a-1)b$	True
Factorize	$ab \pm ac$	$a(b \pm c)$	True
Binomial	$1/(1-a)$	$1 + a/(1-a)$	$0 \notin [\![1-a]\!]$
Fractional Add	$b/c \pm a$	$(b \pm ac)/c$	$0 \notin [\![c]\!]$
Invert Division	a/b	$1/(b/a)$	$0 \notin [\![a]\!] \cup [\![b]\!]$
Expand Division	a/b	$1 + (a-b)/b$	$0 \notin [\![b]\!]$
Basic Quadratic	$a^2 - 1$	$(a-1)(a+1)$	True
Log Exponential	$\ln(e^a)$	a	True
Cancel Inverse	$a(1/a)$	1	$0 \notin [\![a]\!]$

under this interpretation, (4.2) uses the intersection operation, the meet operation of the lattice of intervals.

$$\mathcal{A}[\![c]\!] = \bigcap_{n \in \mathcal{N}_c} \mathcal{A}[\![n]\!] \tag{4.8}$$

This relationship generates monotonically narrowing interval abstractions. 0-arity nodes represent either constants associated with degenerate intervals or variables taking user defined interval constraints.

The classical problem of IA is the so-called 'dependency problem', arising because the domain does not capture correlations between multiple occurrences of a single variable. Consider an input $x \in [0,1]$, under classical IA:

$$\mathcal{A}[\![x - x]\!] = [0,1] - [0,1] = [0-1, 1-0] = [-1,1]. \tag{4.9}$$

The e-graph framework discovers, via term rewriting, $x - x \cong 0$ and by (4.8) the expression is now correctly abstracted by the (much tighter) degenerate interval $[0,0]$.

A set of 39 rewrites is used, defining equivalences of real valued expressions. The basic arithmetic rewrites are commutativity, associativity, distributivity, cancellation and idempotent operation reduction across addition, subtraction, multiplication and division. Conversion rewrites describe the natural equivalence between the power function and multiplication/division. Table 4.1 contains the remaining rewrites. Beyond these rewrites, more complex quadratic polynomial factorization rewrites using the quadratic formula and completing the square are included, checking that the polynomial has real roots to avoid complex numbers.

Conditional rewrites, e.g. "Invert Division", are only valid on a subset of the input domain. Via IA the e-graph can prove the validity of such rules. In (4.10) IA can confirm that $0 \notin [\![x+y]\!]$, in order to remove multiple occurrences of variables resulting in expression bound improvements.

$$\frac{x+y}{x+y+1} \to \ldots \to \frac{1}{1+\frac{1}{x+y}} \tag{4.10}$$

As noted earlier, multiple expressions in an e-class can independently contribute to a more precise abstraction. Consider the equivalent expressions shown in Figure 4.5 for variables $x \in [0,1]$ and $y \in [1,2]$. All three reside in the same e-class with associated interval $[-3, \frac{1}{3}] \cap [-2,0] \cap [-1,1] = [-1,0]$. The e-graph generates a tight interval enclosure using distinct, yet equivalent, expressions for the upper and lower bounds.

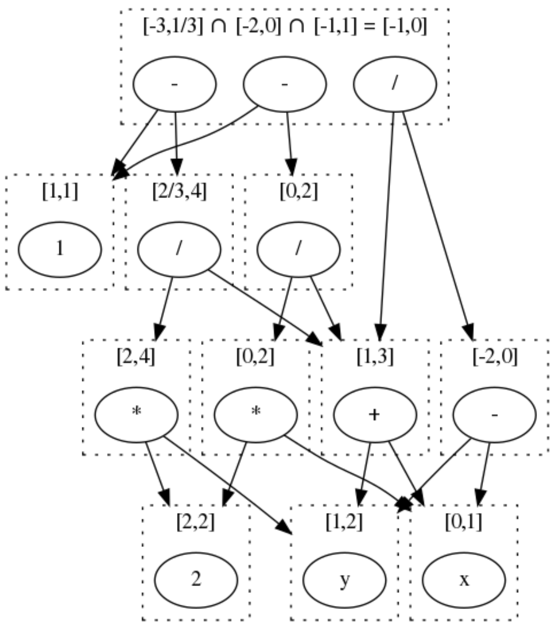
4.3 Results

Based on the theory introduced above a real valued expression bounding tool is developed in Rust using the `egg` library. The evaluation aims to answer the following research questions.

1. Does the interval analysis allow conditional e-graph rewriting to explore a larger space of equivalent designs?

2. Can combining interval arithmetic and equality saturation provide tighter interval bounds than a naive interval analysis?

3. Do cyclic e-graphs provide any practical benefit?



$$\begin{aligned} & 1-\frac{2y}{x+y} && \in \left[-3,\tfrac{1}{3}\right] \\ \cong\ & \frac{x-y}{x+y} && \in [-2,0] \\ \cong\ & \frac{2x}{x+y}-1 && \in [-1,1] \end{aligned}$$

Figure 4.5: An e-graph representing three equivalent expressions. Each expression contains two variables subject to input range constraints, $x \in [0,1]$ and $y \in [1,2]$. The input intervals are propagated through the e-graph via an e-class analysis (described in Section 2.4.2). In the root e-class three interval abstractions are combined, where distinct expressions contribute the most precise upper and lower interval bounds.

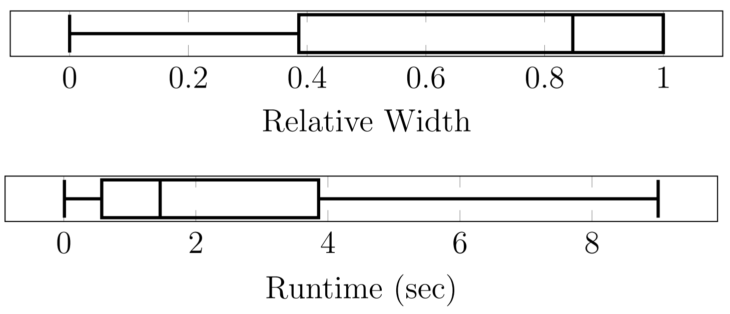


Figure 4.6: Relative interval width (optimized width/naive width) and runtime boxplots to demonstrate the distribution of results on the FPBench suite.

4.3.1 Benchmark Selection

The implementation is evaluated using 40 benchmarks from the FPTaylor [137] supported subset of the FPBench benchmark suite [130]. Four iterations of e-graph rewriting are used, as further rewriting iterations do not yield significant improvements in interval width on these modest benchmarks. All test cases were run on an Intel i7-10610U CPU.

4.3.2 Evaluation

Across these benchmarks, the inclusion of IA and domain specific rewrites increased the number of e-graph nodes by 4% on average but by up to 84% in one case. This demonstrates the additional rewrites that have been applied as a result of combining equality saturation and AI. The overhead of incorporating IA into the e-graph increased runtimes by less than 1% on average, although this will grow with the complexity of the analysis being performed. Figure 4.6 summarizes the distribution of the results, showing a modest average interval reduction over naive IA, but a substantial improvement in particular cases. There is little correlation between the runtime and bound improvement. Note that the exact impact of the additional nodes on the interval refinement was not evaluated.

4.3.3 Iterative Method Discovery

Section 4.1.3 discussed how cyclic e-graphs naturally lead to abstraction refinement loops. In program analysis such refinement loops can correspond to iterative refinement algorithms, evaluated until a user-defined limit. This section demonstrates how cyclic e-graphs can be used to rediscover known iterative refinement algorithms, specifically the Krawczyk method [135]. The Krawczyk method [135] is a known algorithm to generate increasingly precise element-wise interval enclosures of solutions of linear systems of equations $Ax = b$, where A is an n-by-n matrix and b is an n-dimensional vector. Letting X^0 be an initial interval enclosure of the solutions, the Krawczyk method uses an update formula of the form,

$$X^{k+1} = \big(Yb + (I - YA)X^k\big) \cap X^k, \text{ where } Y = \mathrm{mid}(A)^{-1}.$$

$\mathrm{mid}(A)$ is the element-wise interval midpoint of the matrix, A. This sequence, via interval extension and intersection, corresponds to a sequence of tightening bounds on the solution x, which converges provided the matrix norm $||I - YA|| < 1$.

Consider a specific instance of this problem,

$$\begin{pmatrix} 1 & y \\ y & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \text{ where } y \in \left[-\frac{1}{2}, \frac{1}{2}\right]. \tag{4.11}$$

$$X_1^{k+1} = \big(b_1 - yX_2^k\big) \cap X_1^k,\ X_2^{k+1} = \big(b_2 - yX_1^k\big) \cap X_2^k. \tag{4.12}$$

A naive solution in the concrete domain, $x = A^{-1}b$, yields,

$$x_1 = \frac{1}{1-y^2}(b_1 - b_2 y),\ x_2 = \frac{1}{1-y^2}(b_2 - b_1 y). \tag{4.13}$$

Initializing the e-graph with these expressions, the solution for x_1 can be automatically rewritten such that (4.12) arises in the abstract domain.

$$x_1 = \frac{1}{1-y^2}(b_1 - b_2 y) \qquad \textit{Binomial} \tag{4.14}$$

$$\to \left(1 + \frac{y^2}{1-y^2}\right)(b_1 - b_2 y) \qquad \textit{Distribute} \tag{4.15}$$

$$\to b_1 - \left(b_2 y + y\frac{1}{1-y^2}(b_2 y^2 - b_1 y)\right) \qquad \textit{Factorize} \tag{4.16}$$

$$\to b_1 - y\left(b_2 + \frac{1}{1-y^2}(b_2 y^2 - b_1 y)\right) \qquad \textit{Fractional Add \& Simplify} \tag{4.17}$$

$$\to b_1 - y\frac{1}{1-y^2}(b_2 - b_1 y) \qquad \textit{Merge E-Class} \tag{4.18}$$

$$\to b_1 - yx_2 \tag{4.19}$$

First, the "Binomial" rewrite from Table 4.1 introduces a loop into the e-graph, creating an iterative refinement loop. Following the sequence of arithmetic rewrites shown, the e-graph grows to eventually contain (4.18), implicitly performing the x_2 replacement shown in the final step. As a result, the e-class analysis on the e-class containing x_1, corresponds exactly to the iterative refinement loop shown in (4.12). The detailed algebraic rearrangements from this rewrite list are omitted to avoid unnecessary complexity. Initializing the e-graph with both equations in (4.13), the tool simultaneously discovers both of the iterative refinement equations in (4.12).

When a cycle is introduced, the IA update procedure will continue to iteratively evaluate the loop, taking the intersection with the previous iteration as described in Section 4.1.3. The ability to discover such iterative refinement methods within the framework not only facilitates tighter bound generation, but highlights the value in exploring e-graphs containing loops.

4.4 Summary

This chapter took steps towards a complete formalism of the combination of AI and equality saturation. Of key importance is the positive feedback loop between e-graph exploration and abstraction refinement, as the extra precision then permits the application of additional conditional rewrite rules. The additional applications introduce further new expressions, which may further improve abstraction precision. An exemplar IA implementation demonstrates the value of this pairing, even automating the discovery of a known iterative refinement algorithm. Many equality saturation applications could incorporate AI to extend their capabilities for relatively low overhead.

Whilst this chapter focused on the theoretical description of combining AI and equality saturation. The remaining chapters make practical use of the program analysis techniques formalized here, describing a bitvector value range analysis on the e-graph that estimates the range of possible outputs from an e-class following the VeriLang semantics. This analysis will help ROVER to capture new optimizations such as the bitwidth reduction opportunity described at the start of this chapter. Chapter 5 will evaluate the circuit optimizations enabled by this bitvector analysis. Chapters 5 and 6 will also introduce several additional analyses, all based on the formal framework described here.

Chapter 5

Automating Constraint-Aware Datapath Optimization using E-Graphs

The previous two chapters described the foundations of an equality saturation based RTL rewriting framework, and formalized the connection between equality saturation and program analysis. This chapter further extends the optimization capabilities of equality saturation, describing how to encode context within an e-graph setting in Section 5.1. Context encodes the constraints under which a circuit (or sub-circuit) executes, and is frequently exploited by engineers to apply deeper optimizations to their designs. For the remainder of this thesis, we shall call optimization that exploits context, constraint-aware optimization. Section 5.2 extends ROVER with a bitvector value range analysis based on the framework introduced in Chapter 4 and new constraint-aware optimization capabilities. This chapter will primarily focus on the constraint-aware optimization but also highlights a beneficial interaction between the context encoding and analysis.

Constraint-aware optimization provides an opportunity to exploit common control structures such as code branches. Such control structures are commonplace in hardware and software, but automating constraint-aware optimization is challenging, because one must infer the impact of a constraint on an arbitrary code segment. This involves automatically identifying variable correlations and performing a reasoning step.

```
(x>0 ?  f(abs(x))  :  0) ≅
(x>0 ?  f(x)       :  0)
```

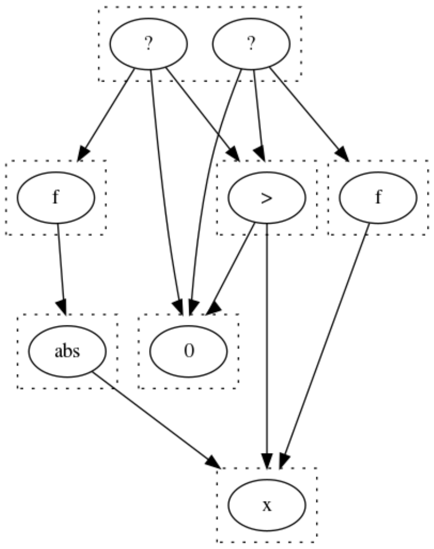


Figure 5.1: An e-graph representing two expressions, which are equivalent due to the constraint imposed by the conditional branch.

Constraint-aware optimization can be reduced to a combination of domain refinement and sub-domain equivalence. This perspective helps to bridge the gap between constraint-aware optimization and equality saturation. A fundamental limitation of the e-graph is that all rewrites define equivalence with respect to the same equivalence relation. This will be problematic in the context of constraint-aware optimization. Consider the equivalent expressions shown in Figure 5.1, containing an arbitrary function `f`. Whilst these two expressions are clearly equivalent, it is difficult to express via local rewrites since, in general, the rewrite $\texttt{abs}(\texttt{x}) \rightarrow \texttt{x}$ is not valid. However, the constraint-aware rewrite $x > 0 \Rightarrow \texttt{abs}(x) \rightarrow x$ is always valid.

Taking inspiration from the program analysis community [98], one general solution to encode sub-domain equivalences in an e-graph is via the introduction of *special* `ASSUME` operators that localize constraints. By defining rewrites over `ASSUME` operators it is possible to encode constraint-aware optimizations within an e-graph setting. This is valuable because these optimizations can be combined with existing work to enhance equality saturation based optimizers. Furthermore, the e-graph can efficiently explore a range of constraint-aware optimizations, deferring the profitability evaluation to a later implementation selection stage.

Constraint-aware optimization is of significant importance in datapath circuit design. Constrained sub-circuits occur naturally when a designer inserts a conditional mux, joining two branches. Such conditions are manually exploited by expert engineers as they often represent additional optimization opportunities. In fact, designers may even specifically insert case-splitting muxes to facilitate further performance optimization. State-of-the-art floating point adder architectures, utilize a carefully designed case-split [47, 46].

To demonstrate the value of constraint-aware optimization, the theory is applied to ROVER, showing that these manual optimizations can be automated and generalized by a constraint-aware e-graph rewriting framework. With these enhancements ROVER combines constraint-aware rewriting with a bit-accurate value range analysis. ROVER's cost model is also extended, allowing the tool to target critical path delay reduction for datapath designs, but also capable of balancing the area-delay trade-off generating a Pareto frontier of implementations.

The chapter is organized as follows. Section 5.1 first describes the key contribution of this work, the `ASSUME` operator, and how it enables the expression of sub-domain equivalence relations in the e-graph data structure. Section 5.2 describes the integration of constraint-aware optimization into ROVER, exploiting context in circuit designs. Lastly, Section 5.3 demonstrates the impact on circuit performance and area using this technique.

The work described in this chapter was first published at DAC in 2023 [138], and was later followed by a 2024 journal extension published in TCAD [139]. This chapter contains the following novel contributions:

- a general purpose encoding of multiple equivalence relations and constraint-aware optimization via e-graph rewriting,
- an accurate value range analysis for bitvector arithmetic in RTL designs,
- a theoretical model of circuit delay and an ILP method to balance the area-delay trade-off and generate a Pareto frontier of designs,
- detailed case-studies and benchmark results demonstrating the automation of sophisticated RTL optimization.

5.1 Localizing Constraint-Aware Optimization

Following the notation introduced in Section 4.1.1, (pg. 70), let *Expr* denote the set of all representable expressions of n variables, ranging over a domain $\mathcal{D}$. The inputs are restricted to $I \subseteq \mathcal{D}^n$ to encode constraints on each input variable, imposed by a type declaration or otherwise. The following then defines a concrete semantics of expressions

$$[\![\cdot]\!]. \in \mathit{Expr} \to I \to \mathcal{D}. \tag{5.1}$$

Under these semantics $[\![e]\!]_\rho$ denotes the interpretation of expression e under execution environment $\rho \in I$. For example, let $I = \mathbb{Z}^2$ and let a and b denote free variables. If $\rho = \{a \mapsto 2, b \mapsto 3\}$, then $[\![a + b]\!]_\rho = 2 + 3 = 5$.

5.1.1 Sub-Domain Equivalence

An e-graph represents a single congruence relation, as given in Definition 4.4 (pg. 71). This notion of congruence requires equality across the entire domain of possible inputs I. However, the presence of input constraints and code branches means that many sub-expressions are only ever evaluated (or are only utilized when evaluated) on a restricted subset of I, constituting a domain refinement. Consider again the example given in Figure 5.1 and let $\mathtt{x} \in \mathbb{Z}$. The true branch imposes a domain refinement, such that on this branch $\mathtt{x} \in \mathbb{N} \setminus \{0\}$. As a result, the requirement on such a strict notion of equivalence for such sub-expressions can be relaxed.

To specify equivalence on a restricted domain, consider $I' \subseteq I$ and define a sub-domain congruence relation.

Definition 5.1 (Sub-Domain Congruence). *Given $I' \subseteq I$, two expressions e_a and e_b are congruent in the sub-domain defined by I', $e_a \cong_{I'} e_b$, iff $[\![e_a]\!]_\rho = [\![e_b]\!]_\rho$ for all $\rho \in I'$.*

Note that, for $I' \subset I$, $e_a \cong_{I'} e_b \not\Rightarrow e_a \cong e_b$. Such sub-domain congruence relations form a lattice [105], where the top element is represented by $\cong$ and the bottom element is represented by $\cong_\emptyset$, as all expressions are congruent on the empty set. The meet and join of this lattice are defined as follows.

$$\cong_{I_1} \sqcap \cong_{I_2} := \cong_{I_1 \cap I_2} \tag{5.2}$$

$$\cong_{I_1} \sqcup \cong_{I_2} := \cong_{I_1 \cup I_2} \tag{5.3}$$

In practice, sub-domain equivalence relations become relevant in the presence of conditional branches and input constraints. Let *BoolExpr* $\subseteq$ *Expr* denote the subset of expressions that under evaluation map to $\{\texttt{True}, \texttt{False}\}$. Given $\texttt{c} \in$ *BoolExpr*, let $[\![\texttt{c}]\!]^T = \{\rho \in I \mid [\![\texttt{c}]\!]_\rho = \texttt{True}\}$, namely the sub-domain on which c evaluates to true. Vice versa, let $[\![\texttt{c}]\!]^F$ denote the sub-domain on which c evaluates to false. These sets correspond to congruence relations satisfying:

$$\cong_{[\![\texttt{c}]\!]^T} \sqcup \cong_{[\![\texttt{c}]\!]^F} = \cong . \tag{5.4}$$

The implementation of constraint-aware optimization utilizes the following observation:

$$e_a \cong_{[\![\texttt{c}]\!]^T} e_b \iff c \vDash e_a \cong e_b. \tag{5.5}$$

The $\vDash$ denotes entailment, which is true when the right-hand side logically follows from the left-hand side. Figure 5.1 demonstrates an example of a valuable sub-domain equivalence.

5.1.2 Sub-Domain Equivalence in an E-Graph

All e-classes within an e-graph are equivalence classes with respect to a single concrete notion of congruence, $\cong$, therefore defining any rewrite $e_a \to e_b \Rightarrow e_a \cong e_b$. To represent sub-domain congruence relations, consider a *special* `ASSUME` operator. The `ASSUME` operator takes two operands, an *Expr* to evaluate and a *BoolExpr* that defines the sub-domain of interest. To the input domain $\mathcal{D}$, append an additional *special* element, forming a new domain $\mathcal{D}' = \mathcal{D} \cup \{\bot\}$. Using this element, $\bot$, the semantics of `ASSUME` can be defined, such that only when the condition holds is the result of the expression of interest, and everywhere else, two

`ASSUME` operators with identical conditions will match.

$$[\![\texttt{ASSUME}(\texttt{x}, \texttt{c})]\!]_\rho = \begin{cases} [\![\texttt{x}]\!]_\rho & \text{if } [\![\texttt{c}]\!]_\rho, \\ \bot & \text{otherwise.} \end{cases} \tag{5.6}$$

Also define for any n-ary function f, if any $[\![\texttt{x}_i]\!] = \bot$:

$$[\![f(\texttt{x}_1, \ldots, \texttt{x}_n)]\!] = \bot. \tag{5.7}$$

It follows from (5.6) that:

$$\texttt{x} \cong_{[\![\texttt{c}]\!]^T} \texttt{y} \Leftrightarrow \texttt{ASSUME}(\texttt{x}, \texttt{c}) \cong \texttt{ASSUME}(\texttt{y}, \texttt{c}). \tag{5.8}$$

Crucially, for $\rho \in [\![\texttt{c}]\!]^F$, the equivalence in (5.8) is vacuously satisfied. Returning to the example in Figure 5.8. It is possible to encode the desired equivalence as

$$\texttt{ASSUME}(\texttt{abs}(\texttt{x}), \texttt{x} > \texttt{0}) \cong \texttt{ASSUME}(\texttt{x}, \texttt{x} > \texttt{0}). \tag{5.9}$$

Table 5.1 describes a set of rewrites, encoding the creation, propagation and refinement of `ASSUME` operators. In this work, `ASSUME` operators are derived purely from code branches. The "Propagate" rewrite allows ROVER to localize reasoning tasks, encoding the search to locate the constrained sub-expression. Consider the following example, where f and g represent arbitrary functions:

$$\begin{aligned} \texttt{ASSUME}(f(g(\texttt{x})), \texttt{x} == 1) &\rightarrow f(\texttt{ASSUME}(g(\texttt{x}), \texttt{x} == 1)) \\ &\rightarrow f(g(\texttt{ASSUME}(\texttt{x}, \texttt{x} == 1))). \end{aligned}$$

Propagating an `ASSUME` operator through an e-graph duplicates the sub-e-graph, a potentially costly overhead. In circumstances where there is no constrained sub-expression in the branch, this adds useless nodes to the e-graph and is unlikely to yield profitable optimization opportunities. By maintaining a live e-class analysis on the e-graph it is possible to guide sub-e-graph duplication, only propagating `ASSUME` operators when it may be profitable. To track live e-classes, with each e-class associate a list of e-classes reachable via any path starting from that e-class. More formally, let E denote the set of e-graph edges and $L(x)$ return the live e-classes associated with e-class x. Then L is calculated as follows.

$$L(x) = \{x\} \cup \bigcup_{\substack{n \in x \\ (n,y) \in E}} L(y)$$

Note that this e-class analysis uses the lattice join (union) to combine abstract elements in the domain of sets of e-classes, as opposed to the lattice meet used in Section 4.2. To utilize this analysis, the "Create" rewrites, defined in Table 5.1, are conditionally applied when the live-variables of the constraint e-class and evaluated expression have a non-empty intersection. More formally, for an expression over e-classes a, b and c,

$$\begin{aligned}&\text{if } L(a) \cap L(b) \neq \emptyset \\ &\quad a \mathbin{?} b : c \rightarrow a \mathbin{?} \texttt{ASSUME}(b, a) : c.\end{aligned}$$

In hardware design, control signals, such as resets, do not offer datapath optimization opportunities since they are typically not correlated with the data signals. Following the same approach, the "Propagate" rewrite is also conditionally applied based on the live class analysis. Such a conditional `ASSUME` creation and propagation approach only introduces the overhead of e-graph duplication when it has the potential to be profitable. Empirical evidence to support this is provided in Section 5.3.

In the presence of nested mux structures, constraints on a sub-expression can be accumulated via the "Combine" rewrite. The combination of constraints may yield conflicts, which, after applying Boolean rewrites, produces an $\texttt{ASSUME}(a, \texttt{False})$ expression. This expression corresponds to dead-code, which can be eliminated by rewriting to something trivial e.g. $\texttt{ASSUME}(0, \texttt{False})$. This is shown as the first "Refine" rewrite in Table 5.1. The last two "Refine" rewrites represent other dead-code elimination optimizations.

Table 5.1: `ASSUME` node rewrites, describing creation, propagation and refinements. `op` represents any operator defined in the expression language. The two argument `op` extends naturally to arbitrary arity. Boolean negation is represented by $\sim$.

Name	Left-Hand Side	Right-Hand Side
Create	$a\,?\,b\,:\,c$	$a\,?\,\texttt{ASSUME}(b,\ a)\,:\,c$
	$a\,?\,b\,:\,c$	$a\,?\,b\,:\,\texttt{ASSUME}(c,\ \sim a)$
Propagate	$\texttt{ASSUME}((a\ \texttt{op}\ b),\ c)$	$\texttt{ASSUME}(a,\ c)\ \texttt{op}\ \texttt{ASSUME}(b,\ c)$
Combine	$\texttt{ASSUME}(\texttt{ASSUME}(a, b),\ c)$	$\texttt{ASSUME}(a,\ b\,\&\,c)$
Refine	$\texttt{ASSUME}(a, \texttt{False})$	$\texttt{ASSUME}(0, \texttt{False})$
	$\texttt{ASSUME}((a\,?\,b\,:\,c),\ a)$	$\texttt{ASSUME}(b,\ a)$
	$\texttt{ASSUME}((a\,?\,b\,:\,c),\ \sim a)$	$\texttt{ASSUME}(c,\ \sim a)$

The general rewrites in Table 5.1 can be extended to encompass application specific optimizations, for example (5.9). To illustrate the advantage of combining constraint-aware optimization with equality saturation, consider this toy example.

$$\texttt{x} == \texttt{y}\,?\,\texttt{x} + \texttt{y}\,:\,\texttt{z} \tag{5.10}$$

Whilst a simplified expression is obvious to a human, this may prove challenging for a traditional optimizer. However, the simplification can be realized via constraint-aware optimization and equality saturation using just three rewrites.

```
x == y  ?  x + y                 :  z →
x == y  ?  ASSUME(x + y, x==y)   :  z →
x == y  ?  ASSUME(y + y, x==y)   :  z →
x == y  ?  ASSUME(2y, x==y)      :  z
```

5.1.3 Program Analysis Refinement

`ASSUME` operators encode sub-domain equivalences via specifically defined rewrites. However, the origins of `ASSUME` operators lie in abstract interpretation [98], where they modify an abstract element assuming the associated constraint holds. This hints at a connection to the theory described in Chapter 4. As in Chapter 4, for a given expression, e, abstract interpretation associates an abstract element $\mathcal{A}[\![e]\!]$ with that expression, and similarly, given an e-class

c, associates an abstract element $\mathcal{A}[\![c]\!]$ with each e-class. In an e-graph representation of the `ASSUME` operator, its children are in fact e-classes which, via an e-class analysis, have an associated abstract element. Therefore, the abstract interpretation definition of `ASSUME` can be naturally transferred and applied to an e-graph. Furthermore, by lifting the analysis to the e-class level, the complex equational reasoning about how the constraint affects the analysis is simplified to a syntactic check. This will be illustrated via an example.

Consider a simple example based on interval arithmetic.

$$\texttt{x>y ? x - y : y - x} \tag{5.11}$$

Let $\mathcal{A}[\![\texttt{x}]\!] = \mathcal{A}[\![\texttt{y}]\!] = [0, 255]$ and let $\mathcal{A}[\![\texttt{c ? a : b}]\!] = \mathcal{A}[\![\texttt{a}]\!] \sqcup \mathcal{A}[\![\texttt{b}]\!]$. A naive interval analysis yields an overapproximation.

$$\mathcal{A}[\![\texttt{x - y}]\!] = \mathcal{A}[\![\texttt{y - x}]\!] = [-255, 255] \Rightarrow$$
$$\mathcal{A}[\![\texttt{x>y ? x - y : y - x}]\!] = [-255, 255]$$

Introducing `ASSUME` operators as described in Section 5.1.2 yields two new expressions to interpret.

$$\mathcal{A}[\![\texttt{ASSUME(x - y}, \texttt{x>y)}]\!] = ? \tag{5.12}$$
$$\mathcal{A}[\![\texttt{ASSUME(y - x}, \sim \texttt{x>y)}]\!] = ? \tag{5.13}$$

Whilst the reasoning required to infer the implication of the assumed condition on the abstract element is not difficult, it is non-trivial. In contrast, by applying additional rewrites to an e-graph containing these `ASSUME` operators, as shown in Figure 5.2, the reasoning can be

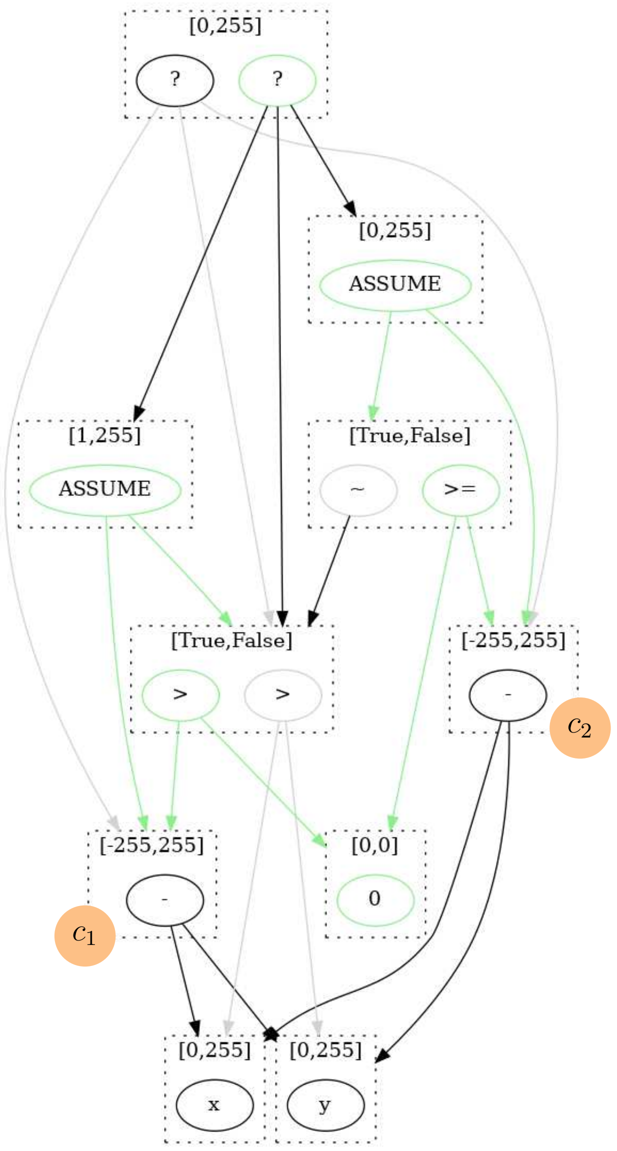


Figure 5.2: An e-graph demonstrating abstract element refinement via `ASSUME` nodes. The original design is marked by light grey arrows. The green nodes represent the additional nodes added via rewriting that enable the interval refinement. Each e-class is associated with an interval of possible outputs. Two e-classes are annotated with additional labels c_1 and c_2, as these represent the constrained e-classes.

simplified.

$$\begin{aligned}
(5.12) = \mathcal{A}[\![\texttt{ASSUME}(c_1, c_1 > 0)]\!] &= [-255, 255] \cap [1, \infty) \\
&= [1, 255] \\
(5.13) = \mathcal{A}[\![\texttt{ASSUME}(c_2, c_2 \geq 0)]\!] &= [-255, 255] \cap [0, \infty) \\
&= [0, 255]
\end{aligned}$$

Here, c_1 and c_2 denote the e-classes in Figure 5.2 containing `x-y` and `y-x`, respectively. These are the two subtraction classes in Figure 5.2. Recall from Section 4.2, that the intersection is the meet of the lattice of intervals, and that the meet of two valid abstract approximations is a valid abstract approximation. For an interval analysis, such as that described in Chapter 4, define the abstraction of the `ASSUME` operator.

$$\mathcal{A}[\![\texttt{ASSUME}(e, c)]\!] = (\mathcal{A}[\![e]\!] \cap I)\,, \text{ where for arbitrary constant } k$$

$$I = \begin{cases} (-\infty, k) & \text{if } (e < k) \quad \in c \\ (-\infty, k] & \text{if } (e \leq k) \quad \in c \\ (k, \infty) & \text{if } (e > k) \quad \in c \\ [k, \infty) & \text{if } (e \geq k) \quad \in c \\ [k, k] & \text{if } (e == k) \in c \\ (-\infty, k) \cup (k, \infty) & \text{if } (e \neq k) \quad \in c \\ (-\infty, \infty) & \text{otherwise.} \end{cases} \tag{5.14}$$

To perform the analysis shown in (5.14), requires the detection of which expressions are contained in the set of expressions represented by the constraint e-class c. This e-class c may contain compound expressions, such as $(e < k)\&(e > l)$, or several different representations of the same constraint. In Figure 5.1, after applying the standard comparator rewrites shown in Table 5.2, one constraint e-class contains both $\sim$ `x` $>$ `y` and `y-x` $\geq$ `0`. Only this second representation is considered analysis-friendly. To simplify the identification of constraint ex-

Table 5.2: Comparator rewrites to simplify `ASSUME` operator analysis refinement. We use `op` to denote any operator in $\{\leq, <, ==, \neq, \geq, >\}$.

Class	Left-hand Side	Right-hand Side	Condition
Comp to Subtract	a `op` b	$a - b$ `op` 0	$a \neq 0$ & $b \neq 0$
	a `op` b	0 `op` $b - a$	$a \neq 0$ & $b \neq 0$
Invert Comp	$\sim (a == b)$	$a \neq b$	True
	$\sim (a > b)$	$a \leq b$	True
	$\sim (a \geq b)$	$a < b$	True
	$\sim (a < b)$	$a \geq b$	True
	$\sim (a \leq b)$	$a > b$	True

pressions, e.g. $(e \leq k) \in c$, an analysis-friendly representative is extracted from the e-class c. The analysis-friendly representative prioritizes the expressions shown in (5.14), reducing the search to a simple pattern match.

Propagating the abstract elements associated with `ASSUME` operators will yield an overall program analysis refinement. As shown in Chapter 4 and later realized in an RTL optimization setting in Section 5.3, a tighter program analysis may enable additional rewriting opportunities. The precise abstract interpretation of `ASSUME` will depend upon the particular abstract domain, but the reasoning simplification due to the e-graph representation will be common to all implementations. Section 5.2 will demonstrate the benefits of abstraction refinement via a bitvector value range analysis which is added to ROVER.

5.2 RTL Performance Optimization

The previous section described how to theoretically encode constraint-aware optimization via e-graph rewriting. This section demonstrates a practical realization of the theory by extending ROVER with constraint-aware optimization capabilities. Figure 5.3 provides an overview of the tool flow, where the components affected by the extension are highlighted. This chapter first extends VeriLang to incorporate an `ASSUME` operator with the semantics described in Section 5.1. Next a set of RTL rewrites is introduced that realizes the constraint-aware datapath optimizations described above. To perform an RTL based bitvector value range analysis,

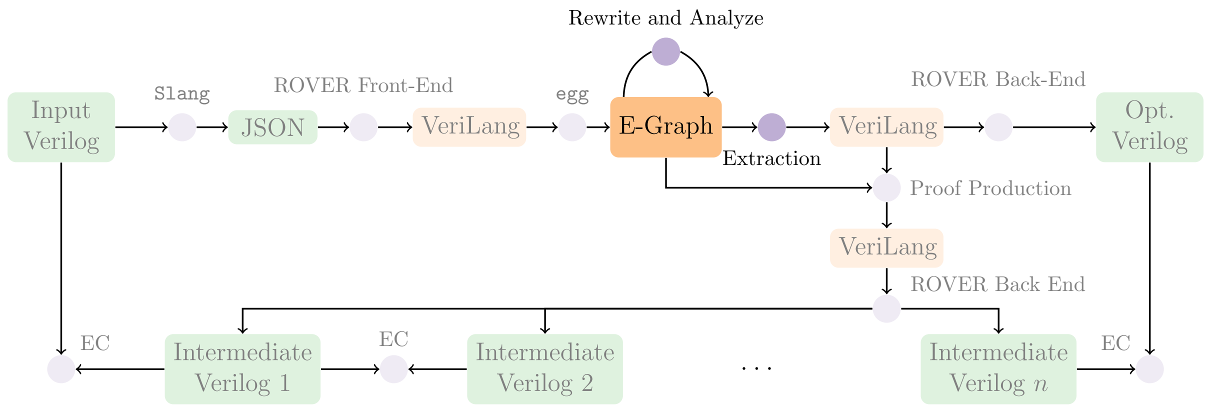


Figure 5.3: Flow diagram describing the operation of constraint-aware ROVER. Components that remain largely unmodified from Chapter 3 are grayed out.

ROVER implements an e-class analysis on the VeriLang e-graph. Lastly, to target performance optimization, ROVER gains an RTL delay model, that captures downstream logic synthesis optimizations. ROVER again focuses on combinational RTL designs in this work, but, as we will see in Chapter 6, can easily be generalized to pipelined designs.

5.2.1 Value Range Analysis

Each signal within the design and in the corresponding e-graph represents a bitvector. ROVER conservatively estimates the range of possible values of each signal using a finite union of positive integer intervals. To be precise, with each expression ROVER associates an element of the set:

$$\mathcal{A} = \left\{ \bigcup_{i=1}^{n} [a_i, b_i] \mid a_i \leq b_i,\ a_i, b_i \in \mathbb{N},\ n \in \mathbb{N} \right\}.$$

ROVER encodes a value range analysis as an e-class analysis (as described in Section 2.4), associating an element of the set $\mathcal{A}$ with each e-class. In RTL, each module input, corresponding to a variable in the e-graph, is defined with an associated bitwidth. ROVER conservatively assumes that all inputs are possible, therefore initializes the e-class analysis for an n-bit input to $[0, 2^n - 1]$. Input value constraints could be captured using an alternative initialization.

Bitwidth truncation is encoded using a conservative approximation to modular intervals.

$$[l,u] \mod p = \begin{cases} [l \mod p, u \mod p] & \text{if } \left\lfloor \frac{l}{p} \right\rfloor == \left\lfloor \frac{u}{p} \right\rfloor \\ [0, p-1] & \text{otherwise.} \end{cases}$$

The outputs of each RTL operator are approximated using an interval arithmetic that follows the Verilog semantics. ROVER stores unions of unsigned integer intervals for all signals, even those defined to be signed. Signage is encoded in the interval analysis of the individual operators. To implement this, define two conversion functions, each parameterized by target bitwidth $n \in \mathbb{N}$. S_n mapping an unsigned interval to its 2's complement interpretation, and U_n mapping an arbitrary integer interval to its unsigned interpretation in n-bits.

$$\begin{aligned} S_n : \{[a,b] \mid a,b \in \mathbb{N}, a \leq b < 2^n\} &\quad \to \left\{ \bigcup_{i=1} [a_i,b_i] \mid a_i, b_i \in \mathbb{Z}, -2^{n-1} \leq a_i \leq b_i < 2^{n-1} \right\} \\ U_n : \{[a,b] \mid a,b \in \mathbb{Z}, a \leq b\} &\quad \to \left\{ \bigcup_{i=1} [a_i,b_i] \mid a_i, b_i \in \mathbb{N}, a_i \leq b_i < 2^n \right\} \end{aligned}$$

$$S_n([x,y]) = \begin{cases} [x,y] & \text{if } y < 2^{n-1} \\ [x-2^n, y-2^n] & \text{elif } x \geq 2^{n-1} \\ [x, 2^{n-1}-1] \cup [-2^{n-1}, y-2^n] & \text{otherwise} \end{cases}$$

$$U_n([w,z]) = \begin{cases} [0, 2^n-1] & \text{if } \left\lfloor \frac{w}{2^n} \right\rfloor < \left\lfloor \frac{z}{2^n} \right\rfloor - 1 \\ [0, z^*] \cup [w^*, 2^n-1] & \text{elif } \left\lfloor \frac{w}{2^n} \right\rfloor < \left\lfloor \frac{z}{2^n} \right\rfloor \\ [w^*, z^*] & \text{otherwise} \end{cases}$$

$$\text{,where } z^* = z \mod 2^n, w^* = w \mod 2^n.$$

These conversion functions demonstrate the need for unions of intervals, rather than simple intervals alone. Consider the following System Verilog example that demonstrates the effect of bit-slicing and using signed signals.

```
wire [7:0] a;
wire [3:0] b;
wire [5:0] add;
assign add = $signed(a[4:0]) + $signed(b);
```

Initializing $\mathcal{A}[\![\texttt{a}]\!] = [0, 255]$ and $\mathcal{A}[\![\texttt{b}]\!] = [0, 15]$, compute

$$\begin{aligned}
&\mathcal{A}[\![\texttt{\$signed(a[4:0])} + \texttt{\$signed(b)}]\!] \\
&= S_5\left([0, 255] \mod 2^5\right) + S_4\left([0, 15] \mod 2^4\right) \\
&= S_5([0, 31]) + S_4([0, 15]) \\
&= [-16, 15] + [-8, 7] = [-24, 22], \\
&\mathcal{A}[\![\texttt{add[5:0]}]\!] = U_6([-24, 22]) = [0, 22] \cup [40, 63].
\end{aligned}$$

Even in this simple example, the analysis identifies that the `add` signal can only take a subset of the integer values that can be represented in six bits. Intuitively, ROVER computes the bitvector interval analysis by converting the bitvectors to integers, performing integer interval arithmetic, then converting back to bitvectors.

Since Verilog is a context-determined language, the signage of an operator is determined by the signage of its operands, as specified in the language reference manual (LRM) [2]. The following procedure is used to calculate the interval abstraction of an arbitrary operator.

1. Determine operator signage according to LRM.
2. Truncate unsigned operand intervals to operand bitwidth and convert to integer intervals according to signage.
3. Compute integer interval arithmetic abstraction of operator.
4. Convert computed interval to unsigned interpretation using the assigned to bitwidth.

To handle unions of intervals, given an interval abstraction function f, approximate $f\left(\bigcup_i^n [a_i, b_i]\right)$ by $\bigcup_i^n f\left([a_i, b_i]\right)$. A basic algorithm reduces a given union of intervals to the minimum number

Table 5.3: Additional VeriLang operators, extending the set described in Table 3.1.

Operator	Symbol	Arity	Architecture
Leading-Zero Count	`LZC`	2	Lookup Table
Absolute Difference	`ABSDIFF`	2	Compound Adder [38]

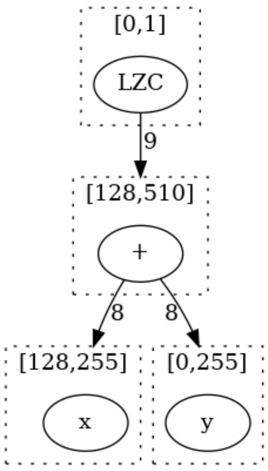
of unions required. For example, $[0,3] \cup [4,7] = [0,7]$.

The value range analysis data enables deeper rewrites that go beyond simple syntactic rewrites, such as bitwidth reduction. These rewrites will be discussed next. Furthermore, when combined with the `ASSUME` node rewriting, ROVER benefits from the associated program analysis refinement described in Section 5.1.3.

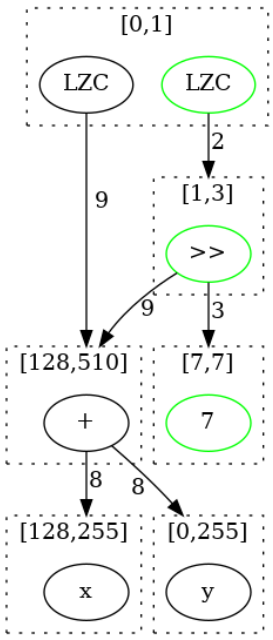
5.2.2 Constraint and Value Range Aware RTL Rewriting

Table 5.3 introduces new VeriLang operators that describe custom hardware components. In particular, VeriLang gains a leading-zero count (LZC) operator that takes two arguments, an input bitvector and a default value that zero is mapped to. Such an expressive LZC operator allows ROVER to capture hardware optimizations that exploit the fact that the LZC of zero is typically undefined. The second addition, the absolute value operator, allows ROVER to express optimizations that rely on a compound adder [38]. Rewrites map typical Verilog implementations of these components to their VeriLang operators. For example, an LZC is commonly implemented using a Verilog lookup table. Section 5.3 shall demonstrate that adding the set of constraint-aware rewrites shown in Table 5.4 to the existing ROVER rewrite set, described in Chapter 3, greatly expand ROVER's ability to capture deep optimizations.

For the newly added rewrites, the value range analysis data is accessed whilst rewriting, to determine the validity of conditional rewrites. For example, let $\mathcal{A}[\![a]\!] = [a_1, a_2]$ and $\mathcal{A}[\![b]\!] = [b_1, b_2]$, then for an unsigned comparison $a > b \rightarrow$ `1'b1` is valid if $a_1 > b_2$. Similarly, the value range analysis can be used by dynamic rewrites to construct the equivalent right-hand side based on the analysis data. The most valuable example of this is for bitwidth reduction rewrites, where ROVER is able to shrink the bitwidth of a given operator if the value range analysis proves that the values which that operator can take can be represented in fewer bits.



(a) Initial e-graph represents LZC($x + y$).



(b) LZC(a) $\to$ LZC($a \gg 7$). Green nodes are newly added.

Figure 5.4: An e-graph before and after rewriting. The LZC node denotes a leading-zero counter. Edges are labeled with bitwidths. The rewrite is valid due to the input constraint, $x \geq 128$, which implies that LZC($x + y$)≤ 1, namely $x + y$ has at most one leading zero.

For example, consider the following Verilog.

```
fma[6:0] = a[2:0] * b[2:0] + c[2:0]
```

A naive analysis may consider each operation independently, first a 3-bit multiplication generating a 6-bit result, followed by a 6-bit addition producing a 7-bit result. In contrast, a value range analysis shows that $\mathcal{A}[\![\texttt{fma}]\!] = [0, 56]$, which implies that `fma` can be reduced to six bits. According to ROVER's circuit area model, introduced in Chapter 3, a 6-bit adder is 19% smaller than a 7-bit adder. Logic synthesis results support ROVER's model, showing a 17% reduction in combinational cells. The general form of the bitwidth reduction rewrite is shown in Table 5.4.

The "Shrink LZC" rewrite, defined in Table 5.4, is a specific instance of bitwidth reduction, that exploits specific knowledge of the LZC operator. An example application is shown in Figure 5.4, where the diagram is simplified by omitting the default argument. The value range analysis proves that the result of the LZC is in $[0, 1]$. This proves that the leading one must be

Table 5.4: Dynamic and conditional rewrites that introduce domain refinement opportunities or exploit domain knowledge. Where ROVER performs bitwidth modifications, we use left subscript notation, ${}_pa$, to denote a bitvector a of length p. We use op to denote any VeriLang operator and use $\geqq$ to denote both $>$ and $\geq$. $\texttt{max}(\mathrm{e})$ denotes the maximum value in $\mathcal{A}[\![e]\!]$. The bw function returns the minimum width bitvector required to store the largest value in a union of positive integer intervals.

Class	Left-hand Side	Right-hand Side
Bitwidth Reduction	${}_r({}_pa \text{ op } {}_qb)$	${}_{r'}({}_{p'}a \text{ op } {}_{q'}b)$
	$r' = \mathrm{bw}(\mathcal{A}[\![({}_pa \text{ op } {}_qb)]\!]),\ p' = \mathrm{bw}(\mathcal{A}[\![a]\!]),\ q' = \mathrm{bw}(\mathcal{A}[\![b]\!])$	
Parallelism	$a \text{ op } (b\,?\,c : d)$	$b\,?\,a \text{ op } c : a \text{ op } d$
	$(b\,?\,c : d) \text{ op } a$	$b\,?\,c \text{ op } a : d \text{ op } a$
Special Case	$a - (b \gg c)$	$(c > 0)\,?\,a - (b \gg c) : a - b$
Alignment	$(a \ll \mathit{const}) \gg b$	$((a \ll \texttt{max}(b)) \gg b) \ll (\mathit{const} - \texttt{max}(b))$
		if $\mathit{const} > \texttt{max}(b)$
Shrink LZC	$LZC({}_pa, d)$	$LZC({}_m({}_pa \gg c), d)$ if $c > 0$
	$m = \max\left(\mathcal{A}[\![LZC({}_pa)]\!]\right),\ c = p - m$	
Simplify Mux	$c\,?\,a : b$	a if $\mathcal{A}[\![c]\!] == [1,1]$
	$c\,?\,a : b$	b if $\mathcal{A}[\![c]\!] == [0,0]$
Absolute Difference	$a \geqq b\,?\,a - b : b - a$	$\texttt{ABSDIFF}(a, b)$
	$\texttt{ABSDIFF}(a, b)$	$\texttt{ABSDIFF}(b, a)$
	$\texttt{ASSUME}(a - b, a \geqq b)$	$\texttt{ASSUME}(\texttt{ABSDIFF}(a, b), a \geqq b)$

in one of the two most significant bits and hence the LZC taking a 9-bit signal can be replaced by an LZC considering only the two most significant bits of the input.

In addition to bitwidth reduction, ROVER benefits from a set of hardware specific optimizations, many of which rely the on value range analysis to prove the correctness of the rewrite. The "Special Case" rewrite specifically introduces a case-split, which from designer intuition is known to be valuable. The value of this rewrite will be demonstrated in Section 5.3.1. The "Absolute Difference" rewrites map expressions to `ABSDIFF` operators, which can be efficiently implemented using a compound adder [38]. Section 5.3.4 will demonstrate the application of these rewrites.

ROVER exploits constraints present in the design itself, which are typically expressed via muxes in RTL, therefore `ASSUME` operators are introduced and propagated by applying the rewrites introduced in Table 5.1. ROVER further benefits from constraint-aware program analysis refinement facilitating further bitwidth reductions, ensuring that minimal bitwidths can be utilized throughout the design.

ROVER's rewrites are mostly learnt from Intel engineers or are derived from efforts to decompose manual optimizations into a sequence of generally applicable transformations. Key conditional and dynamic rewrites are summarized in Table 5.4. In total ROVER uses 179 rewrites, of which 122 are newly added in this Chapter. Since several rewrites are duplicated to match different bitwidths, there are only 60 new distinct transformations encoded by these 122 rewrites.

5.2.3 Extraction

The extraction procedure is similar to that described in Section 3.3. ROVER applies rewrites to the e-graph until saturation or a user defined iteration limit is reached. The extraction process then selects a set of e-classes to implement and within these e-classes chooses the best node to implement that particular e-class. The extraction procedure described in Chapter 3 minimized circuit area, however the procedure does not differentiate between designs with the same circuit area but different circuit delay. For example, the existing procedure does not differentiate between equivalent combinational circuits $((A\&B)\&C)\&D$ and $(A\&B)\&(C\&D)$. Both use three gates, but the first circuit has a delay equal to three gates, whilst the second circuit has a delay equal to two gates.

Building upon the theoretical area model introduced in Chapter 3, ROVER gains a theoretical delay model that estimates the number of two input gate delays on the critical path of each operator, as a function of the input and output parameters. The fixed component architectures are given in Tables 3.1 and 5.3. Delay accumulates through combinational paths in the design, therefore the total delay to an operator's output is the maximum delay across its operands plus its own delay.

Since design delay depends only on the delay of the longest delay path, a greedy approach to extraction, selecting the minimal delay operator from each e-class, will produce the minimum delay implementation. This is implemented using the default `egg` extraction method. Such an approach does not differentiate between implementations that achieve the same delay but occupy different circuit areas. For example, a greedy delay extraction does not differentiate between equivalent combinational circuits $A\&(B\|C)$ and $(A\&B)\|(A\&C)$. Both have a delay equal to two gates, but the first uses two gates, whilst the second uses three gates.

To correctly select designs with efficient area-delay tradeoffs, the ILP encoding, introduced in Section 3.3.2, of minimum area extraction is extended with additional delay constraints that constrain the maximum design delay. With each e-class $c \in \mathcal{C}$ associate a new integer variable, d_c and define a new set $\mathcal{L} \subseteq \mathcal{N}$, of leaf nodes with no children. For each $n \in \mathcal{L}$ associate a, potentially zero, input delay $\text{input\_delay}(n)$. To correctly model pipelined circuits, define $E_R \subseteq E$ of edges originating from register operators. To meet a given delay target $d > 0$, add the following constraints to the ILP.

$$\forall (n,c) \in E \setminus E_R.\ d_{\mathcal{C}(n)} \geq d_c + x_n \times \text{delay}(n) - (1 - x_n) \times K \tag{5.15}$$

$$\forall n \in \mathcal{L}.\ d_{\mathcal{C}(n)} \geq \text{input\_delay}(n) \tag{5.16}$$

$$\forall c \in \mathcal{C}.\ 0 \leq d_c \leq d \tag{5.17}$$

Constraint (5.15) ensures that delay accumulates throughout combinational paths in the design and (5.16) encodes the input delays. The large arbitrary constant K ensures that (5.15) is vacuously satisfied if $x_n = 0$. Lastly, (5.17) ensures that all paths in the extracted circuit meet the delay target d. ROVER deploys the CBC solver [126] yielding the minimum area implementation that meets the total delay constraint d.

Using the greedy extraction method described earlier, ROVER obtains d_{min}, the minimum delay that any design in the e-graph can meet. The minimum area implementation in the e-graph is obtained by solving the original ILP problem, omitting the new delay constraints (5.15), (5.16) and (5.17). This minimum area design has corresponding delay d_{max}. Solving an ILP problem for each delay target $d \in [d_{min}, d_{max})$ yields a complete Pareto frontier of implementations minimizing circuit area at each delay target.

Solving the ILP problems can be time consuming, but ROVER aides the solver by providing the initial greedy solution as an initial feasible solution. Seeding with an initial solution can help to guide the ILP solver and avoids time-outs producing infeasible solutions. The impact of seeding is not evaluated in this work. ROVER also terminates its delay sweep from d_{min} to d_{max} if, for some $d < d_{max}$, the extracted circuit matches the minimum area circuit obtained at d_{max}.

The extraction process generates a set of VeriLang expressions, from which ROVER's back-end generates a set of System Verilog implementations. For each generated implementation, ROVER produces an accompanying sequence of intermediate Verilog designs that can be used to verify that each implementation matches the input design, as shown in Figure 5.3. ROVER's back-end and verification method are unchanged from Chapter 3.

5.3 Results

To demonstrate the value of constraint-aware optimization, an in-depth case-study is provided, with analysis showing exactly how ROVER optimized a floating-point component. A second case-study, a component from a floating-point dot product, is provided to analyze the results of the Pareto frontier extraction method, described in Section 5.2.3. Finally, to show its applicability more generally constraint-aware ROVER is used to optimize a range of benchmarks.

To evaluate the quality of the ROVER's optimizations, the baseline and ROVER generated designs are synthesized using a commercial synthesis tool for a TSMC 5nm cell library. The commercial logic synthesis tool includes state-of-the-art datapath optimizations by default [21, 37]. The commercial tool, optimizing the baseline design is used as a comparison point, as in Chapter 3. The functional equivalence of the original and ROVER generated designs is formally verified using the approach described in Chapter 3.

5.3.1 Case-Study: Floating-Point Subtract

To illustrate the new optimization capabilities in ROVER, consider a complex datapath block, a floating-point subtractor computing $2^{ea} \times 1.ma - 2^{eb} \times 1.mb$ and producing a floating-point output. The subtraction case raises the potential for cancellation and hence is more complex than the pure addition case [46]. The architecture shown in Figure 5.5a is the core datapath of a baseline half-precision (FP16) floating-point subtractor computing the output mantissa. The core datapath takes sorted mantissas Max and Min, along with an ExpDiff$= |ea - eb|$. This architecture is easy to write and simple to verify. It consists of three main components: alignment, subtraction and normalization. The baseline architecture essentially converts the input mantissas to a 42-bit aligned integer representation, performs a 42-bit subtraction, then normalizes using an LZC, truncating the resulting signal.

Floating-point addition operators have been well-studied in academia and industry, providing optimized single-path and dual-path implementations. The dual-path architecture, also known as the near-path/far-path optimization, is derived from the observation that no input fully exercises the critical path of the baseline architecture [46]. The dual-path architecture inserts a case-split using a positive constant c, taking the near-path for small ExpDiff values (ExpDiff $\leq c$) and an alternative far-path for large ExpDiff values (ExpDiff $> c$). On the near-path, close floating point values are subtracted yielding a small alignment shift, since ExpDiff is representable in fewer bits. On the far-path, catastrophic cancellation can be avoided, reducing the renormalization logic.

To explore alternative case-splitting constants c, a mux on the output of the baseline architecture (Figure 5.5a) is manually inserted.

$$\text{ExpDiff} > c \,?\, \text{Out} \;:\; \text{Out} \tag{5.18}$$

Such a redundant case-split would be optimized away by most tools, but with ROVER's constraint-aware optimizations, `ASSUME` operators are propagated down each branch, triggering a chain of branch specific optimizations that result in efficient dual-path implementations.

In Figure 5.6, ROVER is fed dual-path architectures for each value of $c \in \{1, 2, 4, 8\}$. ROVER's cost-model suggests that the optimal split uses $c = 1$, in agreement with computer arithmetic literature [46]. Figure 5.5b presents the ROVER generated dual-path architecture for $c = 1$. The design uses two small 12-bit subtractors, replacing the larger 42-bit subtractor in the baseline architecture. To understand how ROVER generated the dual-path design, a description of the rewrites ROVER applied to transform Figure 5.5a into Figure 5.5b is provided.

In a pre-processing pass, ROVER propagates `ASSUME` operators through the e-graph, duplicating the datapath to create two separate near-path and far-path sub-e-graphs. On the near-path ROVER constrains the ExpDiff ≤ 1, allowing the initial alignment shift in Figure 5.5a to be shrunk, using the bitwidth reduction rewrite (with $i = 1$) that exploits the domain refinement described in Section 5.1.3.

$$\mathcal{A}[\![y]\!] \subseteq [0, 2^i - 1] \Rightarrow x \gg y \to x \gg y[i-1:0] \tag{5.19}$$

Having reduced the shift, ROVER can then apply the "Alignment" rewrite given in Table 5.4, with *const* equal to 31.

$$\texttt{max}(y) < 31 \Rightarrow (x \ll 31) \gg y \to ((x \ll \texttt{max}(y)) \gg y) \ll (31 - \texttt{max}(y)),$$

where $\texttt{max}(y)$ returns the maximum value of the signal y according to ROVER's value range analysis, $\mathcal{A}[\![y]\!]$. These rewrites lead ROVER to discover the following sub-expression, that yields an opportunity to reduce a 42-bit subtraction to a 12-bit subtraction:

$$\begin{aligned}
&(\text{Max} \ll 31) - ((\text{Min} \ll 1 \gg \text{NearExpDiff}) \ll 30) \to \\
&(\text{Max} \ll 1 - (\text{Min} \ll 1 \gg \text{NearExpDiff})) \ll 30, \\
&\text{where NearExpDiff} = \texttt{ASSUME}(\text{ExpDiff}, \text{ExpDiff} \leq 1).
\end{aligned}$$

This shifted subtraction is passed into a normalization circuit (including an LZC). The constant shift and normalization order are swapped by ROVER, reducing the critical path.

$$(x \ll 30) \ll LZC(x \ll 30) \rightarrow (x \ll LZC(x)) \ll 30 \tag{5.20}$$

This sequence of rewrites has essentially pulled the initial alignment shifts through the design until they reach the output, where they meet the truncation shift. With a final rewrite these constant shifts cancel out, resulting in the final shift by one shown in Figure 5.5b.

On the far-path, via `ASSUME` node propagation, ROVER constrains ExpDiff>1. This fact is propagated through the e-graph via the value range analysis, which leads to the automated discovery that the leading zero count on this branch is less than two, since cancellation due to subtraction is limited. This discovery allows ROVER to reduce the 42-bit LZC to an LZC on the most-significant bit only, as in Figure 5.4. Simultaneously, ROVER deploys another shift transformation.

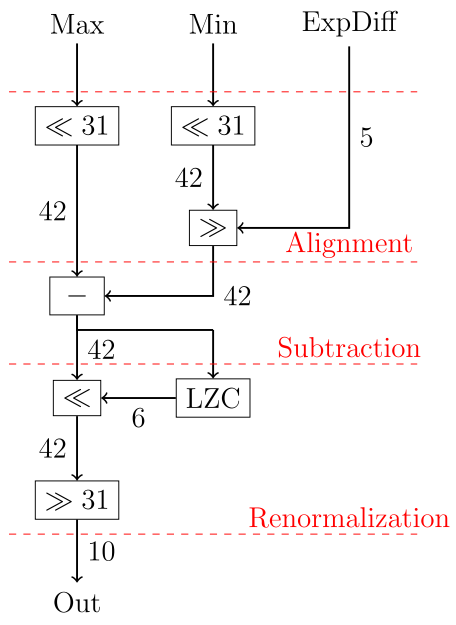
$$(x \ll y) \gg 31 \rightarrow ((x \gg 31 - \max(y)) \ll y) \gg \max(y)$$

In this example, y is the LZC output, therefore $\texttt{max}(y) = 1$. These transformations leave ROVER with the following expression in the e-graph, where x_{30} and y_{30} denote truncation to the least-significant 30 bits:

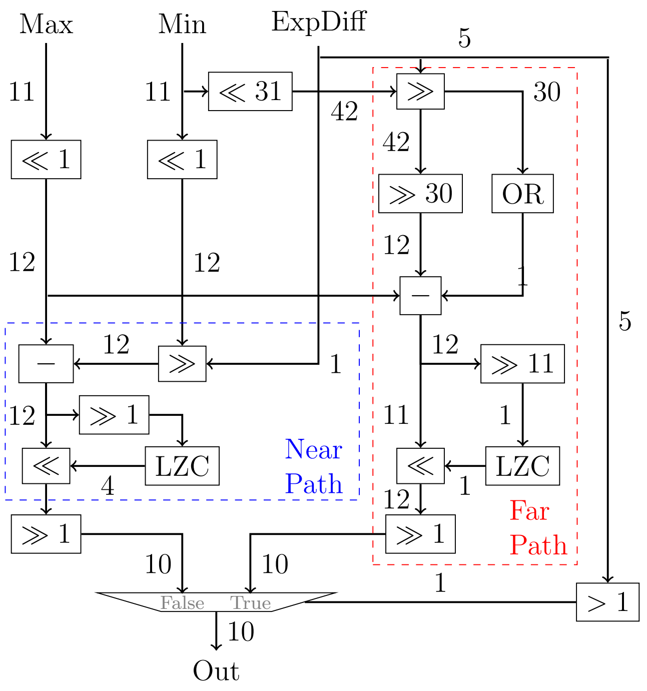
$$(x - y) \gg 30 \rightarrow (x \gg 30) - (y \gg 30) - (x_{30} < y_{30}). \tag{5.21}$$

In the floating-point subtraction context, the final term of the subtraction can be further reduced by ROVER to an or reduction since $x_{30} = (Max \ll 31)_{30}$, which is zero. Whilst rewrites including specific constant values have been shown, ROVER's rewrites dynamically adapt to arbitrary constant values.

The combination of rewrites applied to each branch by ROVER substantially reduces the critical path delay, leading to the dual-path architecture shown in Figure 5.5b. The ROVER generated architecture re-discovers optimizations typically implemented by hand by experienced hardware engineers. As mentioned above, ROVER was able to automatically explore alternative case-splitting constants which are synthesized in Figure 5.6. ROVER can also optimize the single-



(a) Baseline architecture, using a 42-bit subtraction.



(b) Dual-path optimized architecture, using two 12-bit subtractors. Near-path uses a one bit alignment shift, followed by a larger renormalization stage. Far-path uses a 5-bit alignment shift and a single bit renormalization.

Figure 5.5: Half-precision floating-point subtractor architectures. Input mantissas have the implicit one appended. Edge labels represent bitwidths. The floating-point inputs are sorted, such that Max $\geq$ (Min$\gg$ExpDiff). In the optimization, input sorting and exponent difference calculation blocks were omitted to focus on the core datapath.

path architecture, shown in Figure 5.5a, directly, generating the "Single-Path" implementation plotted in Figure 5.6. To achieve this result, an additional "Special Case" rewrite was required purely to aide the value range analysis.

$$x - (y \gg z) \to z > 0 \; ? \; x - (y \gg z) : x - y \tag{5.22}$$

This transformation treats $z = 0$ as a special case, helping ROVER's value range analysis to recognize some of the optimization opportunities that were utilized in the dual-path case. As we can see in Figure 5.6, the dual-path architectures are considered superior by ROVER for constants $c \leq 4$.

The dual-path architecture is generated by ROVER in less than 10 seconds, growing from an initial 100 node e-graph to an e-graph containing over 400 nodes. The majority of the

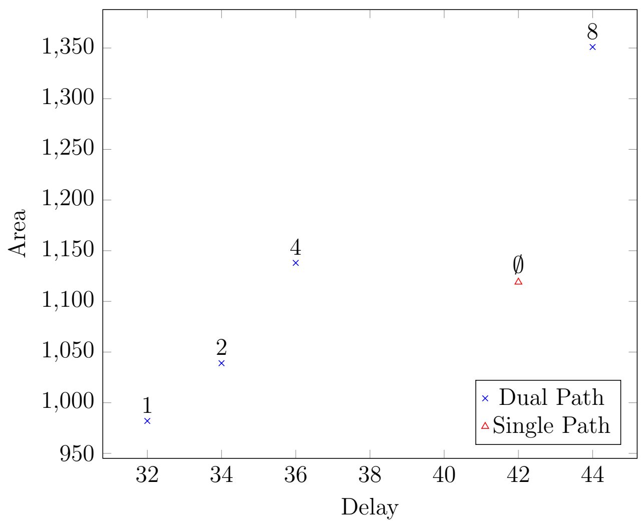


Figure 5.6: ROVER optimized designs for different case-split values, the label above each point represents the value on which the corresponding design case-splits. The area and delay values are those calculated by ROVER's cost models.

runtime is spent in the rewrite application and analysis phases. Figure 5.7 plots the area-delay curves generated from synthesizing four different architectures. The baseline, two ROVER generated single-path designs optimizing for minimal area and minimal delay, respectively, and one ROVER generated dual-path design (Figure 5.5b). The results clearly show that the ROVER generated dual-path architecture is superior, offering a 20% performance improvement for a 3% area penalty, when compared against the baseline architecture optimized by logic synthesis. In contrast, despite exhibiting significant architectural differences, the performance of the two single-path designs and baseline design are very close. This implies that the commercial synthesis tool is able to express most of the optimizations present in the ROVER generated single-path designs. However, the commercial synthesis tool does not appear to be capable of constraint-aware optimizations, the key component required to reproduce the dual-path architecture.

5.3.2 Multi-Objective Optimization

Section 5.2.3 described an algorithm to extract a Pareto frontier of implementations, balancing the area-delay trade-off. The value of such an approach is demonstrated via another case-study.

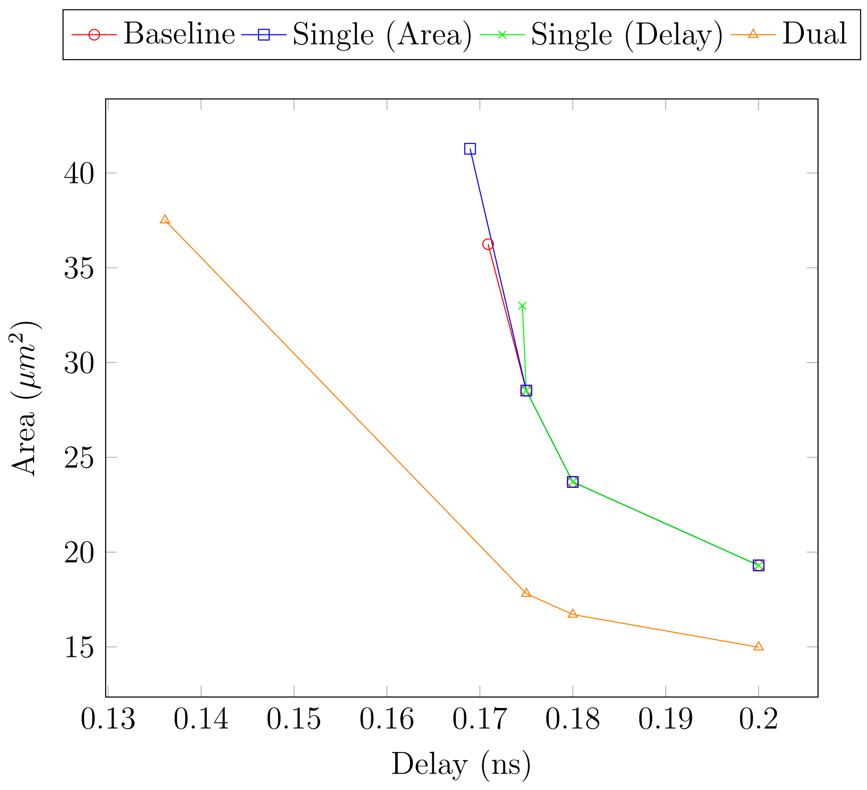


Figure 5.7: Area-delay profiles for baseline FP16 Subtractor and three ROVER generated implementations. Single (Area/Delay) are ROVER's single-path designs optimizing for area or delay, respectively. Dual represents the ROVER generated design shown in Figure 5.5b.

Consider a $2K$-input floating-point dot product.

$$2^{ea_1}ma_1 \times 2^{eb_1}mb_1 + \cdots + 2^{ea_K}ma_K \times 2^{eb_K}mb_K =$$
$$2^{ea_1+eb_1}(ma_1 \times mb_1) + \cdots + 2^{ea_K+eb_K}(ma_K \times mb_K)$$

The hardware implementation of this equation computes each product, producing K product exponents $(ea_i + eb_i)$ and associated product mantissas $(ma_i \times mb_i)$. To perform the final accumulation, the hardware must determine the maximum exponent and compute an alignment factor for each product. This corresponds to a maximum comparison tree followed by K subtractions, as shown in Figure 5.8.

Feeding the baseline implementation into ROVER, it manipulates the mux tree structure, introducing additional subtraction and comparison operators, increasing the degree of speculation. ROVER's e-graph explores all degrees of parallelism available, leaving the extraction phase to determine the optimal implementation based on the delay target. `ASSUME` node rewrites are

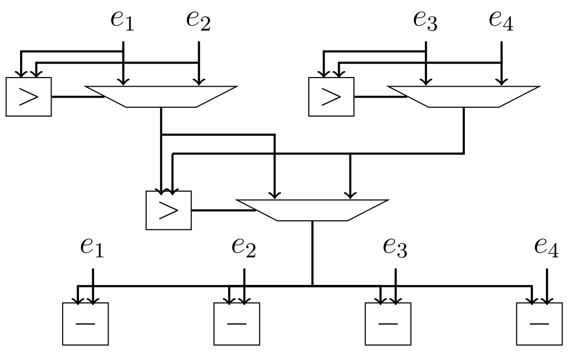


Figure 5.8: Component from a floating-point dot product design, calculating the alignment factor for each product. The hardware computes, for $i = 1, 2, 3, 4$, $\max_j e_j - e_i$.

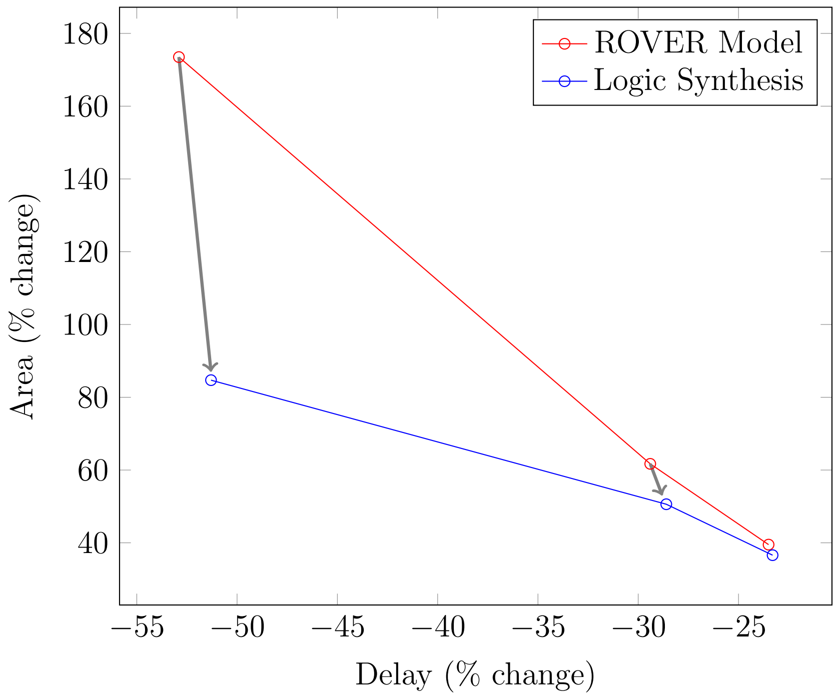


Figure 5.9: Pareto frontier of design costs from a 4-input maximum comparison tree based on ROVER's theoretical cost models and logic synthesis. The percentage change comparing against the baseline implementation is plotted. Grey arrows connect results derived from the same implementation, when not overlapping.

omitted here as the large number of correlated muxes in the design leads to excessive e-graph duplication. The correlation between the mux select signals and the conditional branches, means that the live e-class analysis described in Section 5.1.1 does little to prevent the growth. Whilst the example does not leverage ASSUME nodes it is still relevant to demonstrate how ROVER explores the area-delay trade-off using the ILP model introduced in Section 5.2.3. ROVER generates a 202 node e-graph during exploration, from which multiple implementations are selected.

Given the max tree in Figure 5.8, ROVER generates 18 distinct delay constrained ILP problems, which take 8 seconds to solve in total. These solutions yield a Pareto frontier containing four distinct implementations. The minimum area architecture returned is unchanged from the input shown in Figure 5.8. The estimates given by ROVER's cost model for each implementation are shown in Figure 5.9, using the input architecture as the baseline. The synthesis results closely match the ROVER predicted trend, with one exception. For the implementation with the highest performance, ROVER over-predicts the area penalty, since ROVER's area model fails to capture all the sharing opportunities exploited by logic synthesis.

5.3.3 Benchmark Selection

From the results in Table 5.5, only the "Media Kernel" is carried over from Chapter 3, however the design is slightly modified here, replacing a variable clamping threshold with a constant clamping threshold. The remaining benchmarks from Chapter 3 are not included as they do not present any opportunities to exploit design constraints or the value range analysis computed by ROVER. A number of new benchmarks are introduced that specifically provide opportunities to exploit constraints on the data signals within the designs. The "Unorm8 to FP32", "FP16 to Unorm11" and "Normalization" benchmarks are provided by Intel, whilst the remaining three benchmarks are standard datapath designs that are described within this paper. The approach presented in Chapter 3 is used as an additional comparison point.

Table 5.5: Logic synthesis results under a zero delay synthesis constraint. We compare the baseline, ROVER (Ch. 3), ROVER with only value range analysis (+ VRA) and Constraint-Aware ROVER with value range analysis (+ VRA). Delay and area are measured in ps and μm^2, respectively. We bold the best result for each metric and report percentage improvement against the baseline. † denotes that ROVER was run in two passes.

Benchmark	E-Graph		Baseline		ROVER (Ch. 3)		ROVER + VRA		Constraint-Aware ROVER + VRA			
	Initial	Final	Delay	Area	Delay	Area	Delay	Area	Delay		Area	
Media Kernel	40	645	288.6	192.2	267.5	199.2	**241.4**	**156.7**	**241.4**	-16.3%	**156.7**	-18.5%
Unorm8 to FP32	32	88	36.7	11.2	27.0	9.1	33.9	**8.0**	**23.2**	-36.8%	8.2	-26.5%
FP16 to Unorm11	44	71	54.3	12.8	54.5	15.0	**31.5**	**3.0**	**31.5**	-41.9%	**3.0**	-76.6%
FP16 Sub	79	416	170.9	36.2	176.5	**34.7**	180.9	37.7	**136.1**	-20.3%	37.5	+3.5%
FP8 Sub	86	405	68.9	13.6	68.9	13.6	68.6	**11.7**	**58.3**	-15.4%	14.1	+3.8%
Normalization	134	493	173.5	54.0	167.6	**52.1**	168.6	**52.1**	**135.2**	-22.0%	77.6	+43.5%
Max Tree (†)	43	960	173.7	**33.9**	89.8	51.2	89.8	51.2	**78.1**	-55.0%	56.4	+66.5%

5.3.4 Delay Optimization Evaluation

Having analyzed two case-studies, ROVER can now be used to demonstrate that constraint-aware optimization is generally applicable. Table 5.5 reports synthesis results for a range of benchmarks, where ROVER was used to optimize the designs. The commercial logic synthesis tool with all datapath optimizations enabled, including datapath clustering, constant folding and common sub-expression sharing [21] is used for a baseline (as in Chapter 3). To highlight the benefits of both the value range analysis and the constraint-aware optimizations, an ablation study compares versions of ROVER with and without these features. Across the entire benchmark set, constraint-aware ROVER with value range analysis reduces critical path delay by 30% on average combined with a 1% average circuit area reduction, when compared to the baseline. Total circuit power is omitted, but this is generally closely correlated with circuit area, seeing an average 1% increase. The ablation study demonstrates that, when compared against Chapter 3, the value range analysis alone reduces delay by 3% on average, whilst adding constraint-aware optimizations leads to a 20% average reduction. To understand the resulting improvement, each benchmark is analyzed in detail.

As in Chapter 3 the “Media Kernel” is a kernel from the Intel media module, computing an interpolation between four pixels then clamping the result. For particular constant clamping thresholds, ROVER’s value range analysis proves that the threshold can never be breached, allowing ROVER to remove the clamping. This example demonstrates the advantage of combining rewriting with value range analysis. Before applying rewrites, ROVER’s value range analysis fails to prove that the threshold cannot be breached. After several rewriting iterations, the value range analysis is refined sufficiently to prove the desired property, triggering the removal of the clamping. This optimization only relies on the value range analysis, so constraint-aware ROVER produces an identical implementation.

The “Unorm8 to FP32” benchmark converts a value in a Unorm8 format to an FP32 format, implementing round towards zero. The baseline implementation handles zero inputs on a separate path. Constraint-aware ROVER exploits this to exclude zero from the input domain of the alternative path, leading to a set of constraint-aware optimizations that reduce the critical path. The “FP16 to Unorm11” benchmark performs a reverse conversion, taking an input in FP16 format, and converting it to a Unorm11 format. ROVER re-uses several of the subtraction and rounding rewrites introduced in Section 5.3.1. A lack of data dependent muxes means that constraint-aware optimization offers no further benefit.

Both 8-bit and 16-bit floating-point subtractors, based on the architecture described in Section 5.3.1, benefit from the constraint-aware optimizations, with the FP8 benchmark showing a slightly smaller advantage from the dual-path implementation. In both examples, circuit power consumption increases since the dual-path architecture executes two computational paths in parallel. The “Normalization” benchmark is a kernel from the Intel elementary function unit, normalizing a floating-point value. As in the “FP Sub” benchmarks, a manually inserted case-split is automatically optimized by ROVER, producing an efficient dual-path implementation.

The “Max Tree” was introduced in Section 5.3.2. Due to the large number of muxes in the design, `ASSUME` node rewrites excessively duplicate the e-graph. To combat this ROVER is applied in two passes. First ROVER establishes an efficient mux structure, and then, by enabling the `ASSUME` nodes on a second pass, ROVER discovers resource sharing opportunities via the absolute difference rewrite introduced in Section 5.2.2.

Section 5.1.1 described a live-class analysis and claimed that it limited the propagation of `ASSUME` operators. To evaluate this, only the rewrites in Table 5.1 were applied to the “FP16 to Unorm11” benchmark. Using the live-class analysis to conditionally create and propagate `ASSUME` operators, 48% fewer nodes are added to the e-graph at saturation.

The experiments were run on SLES 12 on modern Intel Xeon CPUs. Since the evaluation does not claim runtime improvements when compared to alternative approaches, a single runtime result is used that does not account for small run-to-run variations. Across the benchmark set, ROVER ran for an average of 69 seconds, whilst the commercial logic synthesis tool ran for between one and two minutes in each cases. The ROVER generated e-graphs were 11 times larger than the initial e-graph, on average. The input benchmarks contained an average 65 lines of System Verilog. For several benchmarks the extraction phase dominated the runtime, hitting the 120 second ILP timeout limit. Efficient e-graph extraction is the subject of a broader community effort to which ROVER benchmarks are actively contributed.[1]

5.4 Summary

This chapter presented a theoretical framework for expressing constraint-aware optimizations in an e-graph. A new `ASSUME` operator was introduced, for which a complete semantics was provided. This theoretical description allows e-graph users to exploit context that can be derived directly from the programs they wish to optimize. Crucially, the chapter described an approach to express sub-domain equivalences and a way to take advantage of the natural program analysis refinements generated by the `ASSUME` nodes. Applying the theoretical contributions to ROVER demonstrated substantial gains in circuit performance across a set of Intel and open-source benchmarks. Through a floating-point subtraction case-study, ROVER demonstrated how constraint-aware optimization can be leveraged to re-discover efficient designs, previously only implemented manually by expert designers. A second case-study highlighted how ROVER can use a single e-graph to explore the area-delay trade-offs.

The techniques presented here represent a substantial leap in datapath circuit optimization capabilities, as no other work has taken advantage of circuit context to match manual design optimization. The next chapter will re-use key ideas behind the `ASSUME` operator, describing how to exploit circuit context to reduce dynamic power consumption by avoiding redundant computation.

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

Chapter 6

Combining Power and Arithmetic Optimization via E-Graph Rewriting

Throughout this thesis, the scope of ROVER has expanded to capture an increasing spectrum of optimization capabilities. This chapter completes the PPA acronym, applying ROVER to circuit power optimization. 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 EDA industry are based on randomly generated or real-world simulation stimuli [58, 59]. ROVER targets dynamic power consumption as that is influenced heavily by RTL design. Leakage power is more heavily influenced by operating voltage and cell selection, which is determined by logic synthesis. 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.

Encoding power optimizations such as clock gating and operand isolation as a set of local

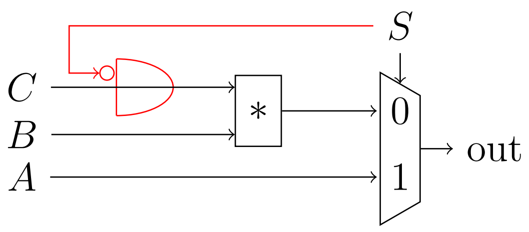


Figure 6.1: An operand isolation opportunity. The input to the multiplier can be data gated when the select signal is one, as shown by the red gate. The negated select signal, $\sim S$ is a common input to an array of AND gates equal to the bitwidth of C.

rewrites allows them to be combined them with the existing arithmetic rewrites, facilitating the exploration of new design spaces and the discovery of novel power efficient designs. An example of the optimizations captured is given in Figure 6.1. By forcing one multiplier input to zero when the computation is redundant the overall power consumed by the multiplier circuit is reduced. The compact e-graph representation also offers advantages to power estimation, via a computationally efficient simulation of design candidates based on the formal framework for analysis presented in Chapter 4. ROVER is extended with power optimization capabilities that allow the tool to customize an implementation based on representative workloads.

This chapter is organized as follows. Section 6.1 describes a set of RTL rewrites that encode power optimizations in ROVER. Section 6.2 describes ROVER's power model and how the compact representation provided by the e-graph yields efficiency gains for RTL simulation. Lastly, Section 6.3 demonstrates ROVER's impact on power consumption on a set of Intel provided and open-source benchmarks.

The work described in this chapter was published at ARITH in 2024 [140]. The chapter contains the following novel contributions:

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

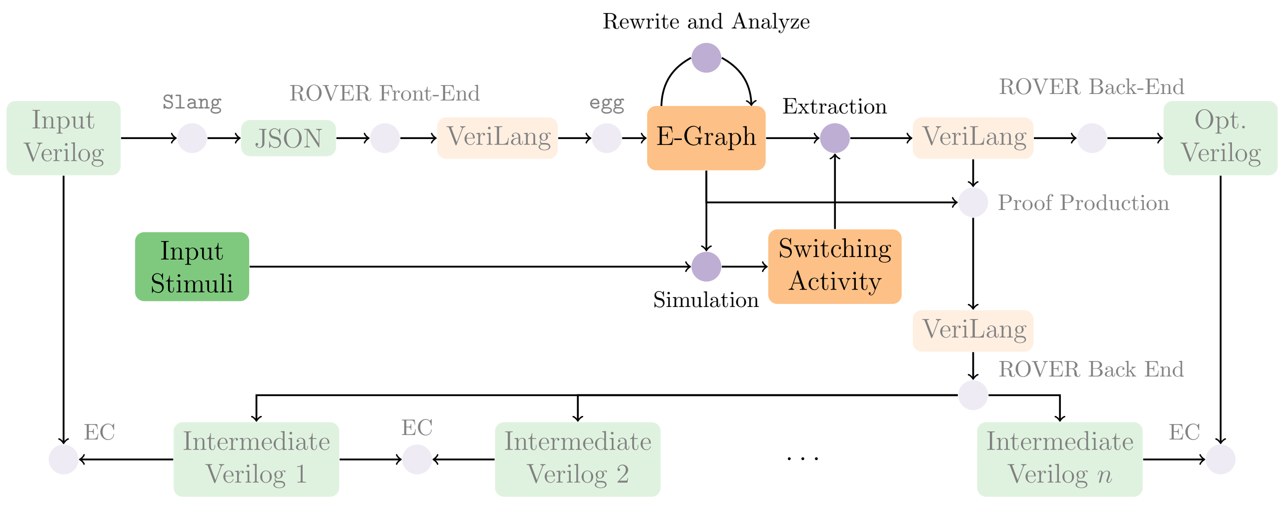


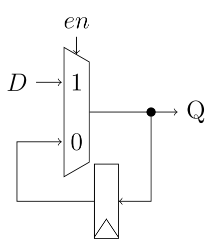
Figure 6.2: ROVER's power optimization tool flow. In addition to an input Verilog design, users provide input stimuli via simulation data or switching activity statistics. ROVER consumes this in a simulation flow, providing the switching activities from which ROVER estimates circuit power consumption.

6.1 Encoding Power Optimizations

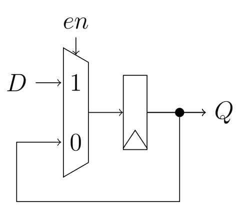
Figure 6.2 visualizes the extensions that allow ROVER to optimize a design to reduce power consumption. As in previous chapters, ROVER's front-end converts an input Verilog design to VeriLang, which `egg` uses to initialize an e-graph. ROVER then applies a set of power optimization rewrites, described in Sections 6.1.1 and 6.1.2, to the e-graph, constructing a set of implementation candidates.

When VeriLang was introduced in Chapter 3, its semantics were defined as operating over Boolean values. In this chapter, the semantics of VeriLang are modified to consider input variables as *streams* of Boolean data, such that a new data point enters the module every clock cycle. This work assumes a single clock domain, for simplicity. Now considering streams of data, it means that 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.

Two new VeriLang operators are included, `REG`, which describes a register with an enable signal, and `TREG`, representing a transparent register. The corresponding circuits are shown in Figure 6.3. Given input a and enable signal en with associated data streams a_i and en_i, $\texttt{REG}(a, en)$



(a) A transparent register (TREG).



(b) An enabled register (REG).

Figure 6.3: Circuit diagrams of the TREG and REG operators.

and TREG(a, en) have the following semantics, which assume that a register is initialized to zero:

$$\texttt{REG}(a_i, en_i) = \begin{cases} 0 & \text{, if } i == 0 \\ a_{i-1} & \text{, if } en_{i-1} \\ \texttt{REG}(a_{i-1}, en_{i-1}) & \text{, otherwise.} \end{cases} \tag{6.1}$$

$$\texttt{TREG}(a_i, en_i) = \begin{cases} a_i & \text{, if } en_i \\ \texttt{TREG}(a_{i-1}, en_{i-1}) & \text{, elif } i > 0 \\ 0 & \text{, otherwise} \end{cases} \tag{6.2}$$

6.1.1 Data Gating

This section describes a set of rewrites that encode the operand isolation optimizations described in Section 2.1.3. A key challenge in expressing operand isolation via local rewrites, is that having identified a redundant computation from a functional perspective the circuit does not care what value is produced under certain conditions. Therefore, from a functional perspective, the circuit can generate any value we choose. However, these chosen values have a significant impact upon power consumption.

Before progressing, it is necessary to define some notation. Let w_x denote the bitwidth of a bitvector variable x and let w_o denote the output bitwidth of an operation. Use $\{w\{S\}\}$ to denote w-fold replication of a bitvector S, which will usually be a single bit. Lastly, use $\sim S$ to denote the bitwise logical complement of a bitvector S.

Table 6.1: A set of RTL rewrites encoding operand isolation and clock gating optimizations. We define four sets of operators such that `op` is any arithmetic or logical VeriLang operator, `op1` $\in \{*, \ll, \gg, +, -\}$, `op2` $\in$ `op1` $\setminus \{+, -\}$ and `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	$s\,?\,b\,:\,c$	$s\,?\,b\,:\,(c\,\&\,\{w_c\{\sim s\}\})$
	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	$(s_1\,?\,a\,:\,b)\,\&\,\{w_o\{s_2\}\}$	$s_1 \Vert \sim s_2\,?\,(a\,\&\,\{w_a\{s_2\}\})\,:\,b$
	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	$s\,?\,b\,:\,c$	$s\,?\,b\,:\,\texttt{TREG}(c, \sim s)$
	Transp Reg Mask	$a\,\&\,\{w_a\{s\}\}$	$\texttt{TREG}(a, s)\,\&\,\{w_a\{s\}\}$
	Transp Reg Saturate	$a \Vert \{w_a\{s\}\}$	$\texttt{TREG}(a, \sim s) \Vert \{w_a\{s\}\}$
	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	$\texttt{TREG}(\texttt{REG}(a, en), \texttt{REG}(b, en))$	$\texttt{REG}(a, en\,\&\,b)$

In one approach to perform operand isolation data gating can be applied 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 6.1 contains rewrites to create initial data gating operations. Two "Gate" rewrites are included as it may be preferable 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 (6.3) to (6.4) illustrates the creation of a mask and data gating of a mux branch, in order to avoid dynamic power in the multiplier. Table 6.1 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. These "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 (6.4) and (6.5).

$$S\ ?\ A\ :\ (C * B) \qquad \rightarrow \textit{(Gate Right)} \tag{6.3}$$

$$S\ ?\ A\ :\ (C * B)\ \&\ \{w_o\{\sim S\}\} \qquad \rightarrow \textit{(Propagate Left)} \tag{6.4}$$

$$S\ ?\ A\ :\ (C\ \&\ \{w_c\{\sim S\}\}) * B \tag{6.5}$$

For such operators, gating just one operand may use half the number of gates but achieve the majority of the power saving when compared to gating both inputs. For example, when both operands are switching at the same frequency, gating just one multiplier operand, as shown in Figure 6.1, yields 91% of the power saving obtained by gating both inputs.

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

$$(s\,?\,f(a)\ :\ b) + g(f(a)), \tag{6.6}$$

the computation of $f(a)$ appears redundant when s is zero, however $f(a)$ is always 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 [54] 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 6.1, ROVER deploys the arithmetic and area optimization rewrites described in Chapters 3 and 5, 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 that provide an efficient area-power trade-off.

6.1.2 Clock Gating

The previous section described how data gating rewrites can encode operand isolation. This section shall describe a set of local equivalence preserving rewrites that create transparent registers providing an alternative way to achieve operand isolation.

The second group in Table 6.1 contains a set of rewrites, similar to the first group, that encode the creation and propagation of `TREG` operators. ROVER improves upon approaches based on mux tree analysis by including the "Transp Reg Mask/Saturate" rewrites that detect redundant computation. ROVER also creates transparent registers from register enable signals, since disabled registers correspond to redundant computation. Similar to the data gating case, the "Combine Transp Reg" rewrite allows ROVER to construct and possibly simplify complex *observability don't care* signals.

The final group in Table 6.1 describes 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. It is possible to prove the equivalence of

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

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 $R_k = L_k$,

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

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 limited retiming of Boolean operators is implemented.

As described in Figure 6.2, 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 is described next.

6.2 Power Estimation and Extraction

To accurately model per implementation power consumption, ROVER simulates the entire e-graph based on user configured input stimuli as shown in Figure 6.2. The user configured input stimuli provide, for every module input, a sequence of bitvectors that are fed one per clock cycle. 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 6.2.2. The power model is used by ROVER to determine the optimal implementation, producing a VeriLang expression. The ROVER back-end and proof production flows are as described in previous chapters.

6.2.1 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, described in Figure 6.4, 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. In the context of the formal analysis framework described in Chapter 4, analyzing any node in an e-class yields the same abstraction, so the combination is trivial and not necessary. 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 [72], as shown for the example of Figure 6.1

```
# Random initial bit
last_bit   = randint(0,1)
stimuli    = vec![last_bit]
threshold = L * switching_activity
while stimuli.len() < L:
    # random integer in range [0,L-1]
    r = randint(0, L-1)
    if r < threshold: # transition
        last_bit = not last_bit
    stimuli.push(last_bit)
```

Figure 6.4: An algorithm to generate simulation stimuli of length L for an individual bit at a given `switching_activity` $\in [0, 1]$.

in Figure 6.5. 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. Observe that the number of e-classes in Figure 6.5 does not grow monotonically. In later rewriting iterations e-classes get merged due to proof of equivalence generated by ROVER reducing the number of classes. For example, consider an e-graph containing two expressions, $f(a + b)$ and $f(b + a)$, upon application of commutativity $a + b$ is proven equivalent to $b + a$. By congruence, the e-graph now merges the two e-classes containing $f(a+b)$ and $f(b+a)$ into a single e-class. 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 all the bits of the output word of that e-class. For example, for a 3-bit word, 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. Note that the average switching activity is identical for all nodes within an e-class because they produce identical values.

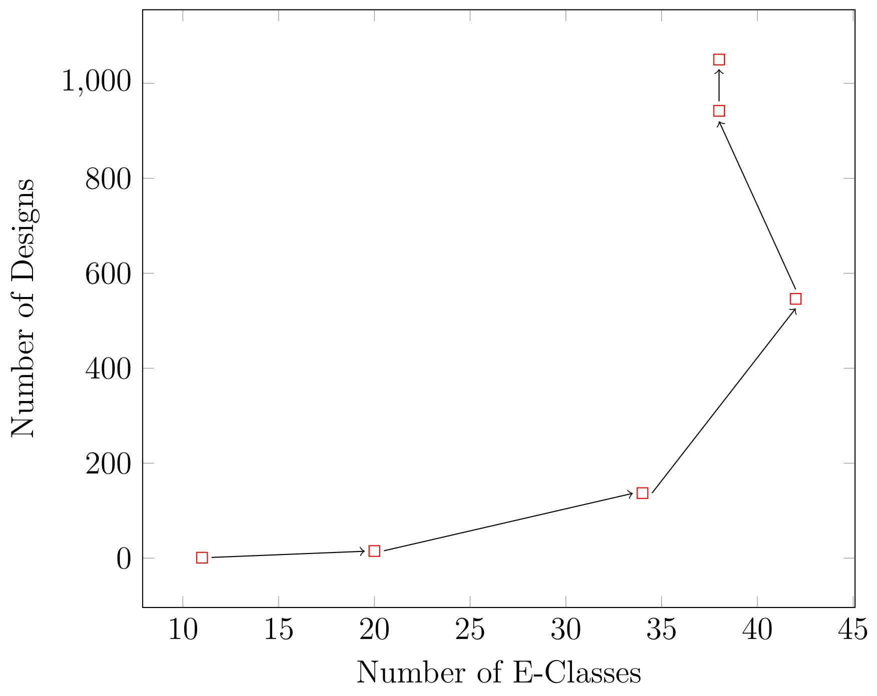


Figure 6.5: The number of designs vs. the number of e-classes after each iteration of rewriting the design in Figure 6.1. Simulation complexity scales with the number of e-classes but evaluates all designs in the e-graph.

6.2.2 Operator Power Model

The purpose of the power model is to order the candidate implementations so that ROVER can select the most power 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 [141]. Chapter 3 described how ROVER encodes certain high-level datapath optimizations, such as arithmetic clustering [21], in the e-graph. To model power, ROVER combines the theoretical circuit area model from Chapter 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 $c_0, \cdots, c_{k-1}$, compute a power estimate, $P(n)$.

$$P(n) = A(n) \times \frac{1}{k+1}\left(T_c + \sum_{i=0}^{k-1} T_{c_i}\right) \tag{6.7}$$

where, $A(n)$ is the theoretical gate area model from Chapter 3, and T_{c_i} and T_c are the operator's input and output toggle frequencies, respectively. The power model assigns an equal weight to input and output switching activities to approximate the proportion of the gates which transition each cycle. The model does not capture wire power consumption. Section 6.3 evaluates how accurately ROVER's model is able to estimate power consumption.

Once again, to correctly account for common sub-expressions, extraction is formulated as an ILP problem, replacing the objective function in the ILP problem described in Section 3.3.2, with the following objective:

$$\sum_{n \in N} x_n \times P(n). \tag{6.8}$$

The rest of the constraints are unmodified to ensure that a valid implementation computing all the module outputs is extracted.

6.3 Results

For each design, ROVER takes the original System Verilog design, which does not contain any existing power optimizations, along with a JSON file that specifies the input switching activities. ROVER is run twice generating an area optimized and a power optimized design in System Verilog along with estimates of the power reduction according to ROVER's power model. Using a commercial logic synthesis tool targeting a TSMC 5nm cell library, the original and ROVER generated designs are synthesized at a range of delay targets to mitigate the impact of logic synthesis noise that was observed in Section 3.6. The commercial tool is provided with the same switching activity configuration as given to ROVER. Table 6.2, shows 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, the cycle-accurate equivalence of the original and ROVER generated designs is verified using a commercial formal EC tool.

Table 6.2: Logic synthesis results comparing the average total power consumption (μW) and average area (μm^2) 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 include the number of nodes in the initial e-graph for each benchmark.

Source	Benchmark	Nodes	Baseline		Area Optimized				Power Optimized			
			Area	Power	Area		Power		Area		Power	
Intel	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%
Open-Source	Pipe. Mux Add Tree	23	**38.6**	852.3	**38.6**	0.0%	852.3	0.0%	44.1	+14.9%	**615.3**	-27.2%
	Dual Op ALU	17	**6.5**	186.9	**6.5**	0.0%	186.9	0.0%	7.5	+15.1%	**146.8**	-21.3%
	Sequential Reg	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	62	**27.8**	1097.1	**27.8**	0.0%	1097.1	0.0%	29.0	+3.5%	**929.4**	-14.9%

6.3.1 Benchmark Selection

ROVER's impact on dynamic power consumption is evaluated on two sets of benchmarks as shown in Table 6.2. Two benchmarks are carried over from previous chapters, that instantiate a pipeline offering power optimization opportunities. The "Weight Calculation" benchmark, introduced in Chapter 3, is a production two-stage pipelined design computing pixel offsets in the graphics pipeline. The "Dual-Path FP Sub", is a pipelined version of the floating point subtractor with a near-/far-path split that ROVER generated. The newly introduced benchmarks are taken from internal Intel low power training materials or existing power optimization literature to to aid comparison with prior work. As in Chapter 5, the approach presented in Chapter 3 is used as an additional comparison point.

The "Combinational Mux Add Tree", is taken from Intel low power training materials and comprises 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 consists of two adders, a multiplier and a pair of muxes.

The second set of benchmarks are taken from prior publications [55, 54, 138, 60]. The "Pipelined Mux Add Tree" [54] is similar to the "Combinational Mux Add Tree" but introduces a distinct pipelined structure. It consists of two adders, three muxes and a pair of registers. The "Dual Op ALU" design [55] can optionally perform either a shift or addition. Next, the "Sequential Reg" benchmark is used in the PowerPro white paper to demonstrate that tool's sequential clock gating capabilities. It is a combination of registers and a mux.

6.3.2 Dynamic Power Reduction

Table 6.2 compares total power and area results for each of the benchmarks before and after ROVER optimization. ROVER reduces total power consumption by up to 33.9% and 23.6% on average at the expense of an average 5.0% increase in circuit area. In power constrained designs this is likely a favorable tradeoff, especially given the area overhead imposed by previous solutions. This will be discussed in the subsequent evaluation. 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, the evaluation is based on total power, including leakage power.

The benefit of exploring arithmetic, area and power in tandem, is highlighted through a deeper analysis of an open-source benchmark. Figure 6.7 shows the circuits corresponding to the baseline "Pipelined Mux Add Tree", the design proposed in [54] and the ROVER generated version. The optimizations proposed in [54] 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. The ROVER generated design contains a three input adder, which, thanks to ROVER's comprehension of logic synthesis optimizations, is recognized as only a single carry-save adder. ROVER then inserts area efficient data gating on the adder inputs to save power. Synthesizing the design proposed in [54], the ROVER generated architecture is strictly better, consuming 11% less power within 17% less area, as shown in Figure 6.6.

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 [55] and [60], demonstrating ROVER's generalization of prior work. Lastly, ROVER recognizes the distinct computational paths in the "Dual-Path FP Sub" and inserts the appropriate clock gating for each path.

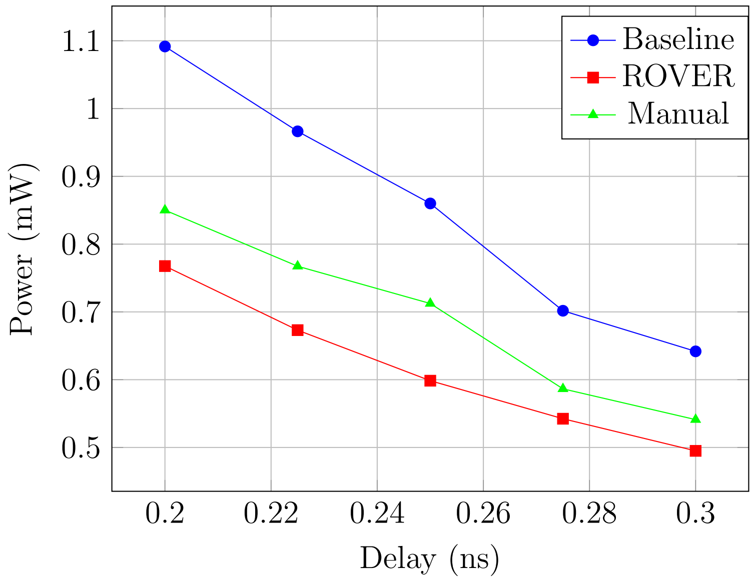
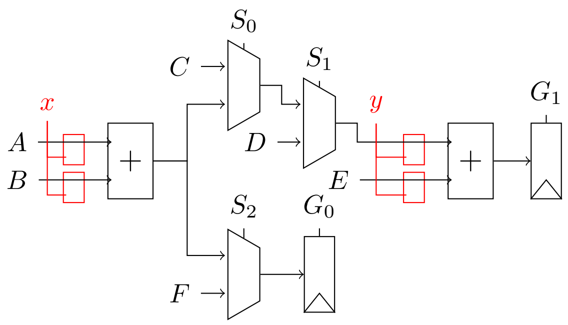
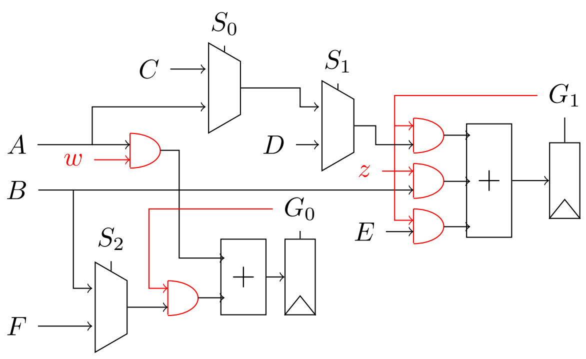


Figure 6.6: “Pipelined Mux Add Tree” delay-power profiles for the baseline, ROVER power optimized and manually optimized designs.



(a) Baseline design (black). In [54] the authors add the transparent registers (red), where $x = (\sim S_2 \& G_0) \| (S_0 \& \sim S_1 \& G_1)$ and $y = G_1$.



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

Figure 6.7: Circuit diagrams of the “Pipelined Mux Add Tree” benchmark with power optimizations from prior work and from ROVER.

The experiments were run on SLES 12 on modern Intel Xeon CPUs. Since the evaluation does not claim runtime improvements when compared to alternative approaches, a single runtime result is used that does not account for small run-to-run variations. 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, the model provides a relevant estimate of the power reduction for a group of five benchmarks, 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. This can be attributed 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, not capturing the relative impact of inputs switching on different operands.

6.3.3 Data Dependent Design

To demonstrate how ROVER is capable of tailoring the implementation to the computation, the switching activities of mux select and register enable signals are modified and ROVER's design choices are studied. Consider the "Pipelined Mux Add Tree" as shown in Figure 6.7. In Table 6.3, four different switching activity configurations are given to ROVER and the associated design choices are highlighted, using different colors to denote distinct optimizations. Given Cfg. 1, ROVER elects to insert data gating using the S_0, S_1, S_2 and G_1 signals, as indicated by the green cells. Given Cfg. 3, where the switching activity for all signals increases, ROVER instead elects to insert a single transparent register, as indicated by the purple cell. In Cfg. 4, ROVER uses the G_0 signal to data gate (green), which was not see in other configurations. The final columns show how the power benefit of ROVER's optimizations varies with switching activities.

Table 6.3: Each row represents a different switching activity configuration (Cfg.) for the mux select and register enable signals in the "Pipelined Mux Add Tree" (Figure 6.7a). For each Cfg., if ROVER inserted a data gate using one of these signals, the corresponding cell is colored green, whilst if ROVER inserted a transparent register, the corresponding cell is colored 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%)

6.4 Summary

This chapter completed the PPA axes, describing how to encode power optimizations, such as operand isolation and clock gating, as local equivalence preserving rewrites over streams of data. By phrasing power reduction as a rewrite problem it can be combined with existing arithmetic rewrites to explore both power and area in tandem, using ROVER. The e-graph representation of design candidates enabled efficient simulation of many functionally equivalent implementations as only one node from each class needs to be simulated. Optimizing a set of benchmarks using ROVER demonstrated a 23.6% reduction in total average power consumption for just an average circuit area increase of 5.0%. These results highlighted the importance of ROVER's understanding of downstream logic synthesis optimizations, leading to designs not seen in prior work.

This chapter concludes our work on ROVER and its application to datapath circuit optimization. In the final technical chapter, the focus shifts to the datapath circuit verification challenge. In particular, we will see that an effective optimization engine can, with some modification, provide the foundations for an equally effective verification engine.

Chapter 7

Formal Verification and Bug Fixing via E-Graph Rewriting

The previous chapters have primarily focused on datapath optimization, measuring improvements in PPA metrics. In every case, there was an associated verification challenge, namely, using trusted EC tools to prove that ROVER has not modified the functional behavior of the circuit. The same challenge remains when a human design engineer provides a new implementation or optimizes an existing circuit description in RTL. ROVER resolved this challenge by automatically decomposing the verification problem into a sequence of simpler sub-problems. For the human design engineer, producing and, even worse, maintaining such a proof decomposition is time consuming and error prone. Compounding this problem, the design is typically passed on to a verification engineer, who is tasked with verifying the implementation. Intentionally, the verification engineer has little knowledge of how the design works, making their task very difficult should their verification tools fail to converge out-of-the-box. One objective in this chapter is to provide verification engineers with a verification assistant capable of automatically decomposing an equivalence checking problem into a sequence of simpler sub-problems.

As a secondary objective, this chapter addresses the datapath debugging challenge. As described in Section 2.2, when two designs are not equivalent, existing verification tools return a single counter-example. From this counter-example, an RTL engineer may be able to isolate the source of the bug, but the tools provide no guidance on where the bug may be or how to fix it. An extension of the existing RTL rewriting framework provides a tool capable of automatically generating a minimal fix to the implementation design, for relatively simple bugs.

To motivate the first objective, Figures 7.1a and 7.1b show functionally equivalent specification and implementation designs, written in Verilog. Despite commercial tools orchestrating a suite of solver technologies [86], including SAT, SMT and BDDs, several industrial EC tools return an inconclusive result. By manually decomposing the problem, it is possible to aide these tools in reaching convergence. Alongside the Verilog is a simplified SMT encoding in the theory of bitvectors [142]. In this representation, it is necessary to explicitly zero extend operands because the bitvector theory operators are only defined for uniform bitwidth operands. Once again, leading SMT solvers, CVC5 [89] and Z3 [90] are unable to solve the simplified encoding, even when given hours of computation time.

Certain limitations of existing EC tools are described in an overview of the technology behind Synopsys' industry leading Datapath Validation (DPV) tool [86]. DPV orchestrates a suite of techniques and solvers to prove the equivalence of input designs. One of these techniques is a set of rewrite engines. In [86], the authors state that certain rewrite sets "are only applied selectively" or their application "can be counter-productive". As a result these rewrite engines are heuristic and may not explore the required space. Such a description suggests that an e-graph approach may yield more reliable results.

To tackle objective one, the underlying e-graph-based RTL rewriting framework is modified to produce a formal verification assistant, called ROVERIFY. Intuitively, instead of initializing the e-graph using a single design, ROVERIFY starts with an e-graph representing two designs that are to be proven equivalent. Applying rewrites to the e-graph simultaneously transforms the two designs such that they converge to some (unknown) intermediate design. The approach differs from ROVER primarily in the e-graph initialization and extraction stages. The proposed verification assistant is able to exceed the capabilities of the industrial state of the art, reduce verification runtimes and decrease the complexity of the EC problem.

The approach taken here is similar to that of Stepp, Tate and Lerner, who initially developed an

```
module spec(A,B,M,N,O);
  input   [15:0] A, B;
  input   [3:0]  M, N;
  output  [62:0] O;
  wire    [30:0] D;
  wire    [30:0] E;

  assign D = A << M;
  assign E = B << N;
  assign O = D * E;
endmodule
```

(a) Specification design.

```
module impl(A,B,M,N,O);
  input   [15:0] A, B;
  input   [3:0]  M, N;
  output  [62:0] O;
  wire    [31:0] C;
  wire    [4:0]  P;

  assign C = A * B;
  assign P = M + N;
  assign O = C << P;
endmodule
```

(b) Implementation design.

```
(set-logic QF_BV)
(declare-fun A () (_ BitVec 16))
(declare-fun B () (_ BitVec 16))
(declare-fun M () (_ BitVec 4))
(declare-fun N () (_ BitVec 4))

(declare-fun spec () (_ BitVec 16))
(declare-fun impl () (_ BitVec 16))

(declare-fun zeros_12 () (_ BitVec 12))
(assert (= zeros_12 #b000000000000))

(assert (= spec (bvmul (bvshl A (concat zeros_12 M))
                       (bvshl B (concat zeros_12 N)))))
(assert (= impl (bvshl (bvmul A B)
                       (bvadd (concat zeros_12 M) (concat zeros_12 N)))))

(assert (not (= spec impl)))
(check-sat)
```

(c) SMT bitvector theory encoding of equivalence between spec and implementation.

Figure 7.1: A motivational example, where existing EC tools fail to prove the equivalence of these two designs. The simplified SMT encoding in the theory of bitvectors only compares the least significant 16-bits of the outputs. A complete encoding is shown in Appendix C.

e-graph based LLVM optimizer [14], and later modified it to perform translation validation [143]. The approach here differs from this previous work in that ROVERIFY validates numerically intense optimizations at a lower abstraction level often performed by a human rather than a compiler. The approach presented is sound, as each intermediate step is checked using a trusted EC tool.

For objective two, the framework is further modified to produce ROVERIFIX, a tool that automatically proposes minimal fixes to a buggy implementation design. Again, the e-graph initialization is adjusted such that the specification design gets rewritten to close the gap between the two designs. The implementation is not rewritten since the aim is to propose the fix with the minimal modification to the implementation. This can be phrased as finding the nearest design that is functionally equivalent to the specification. ROVERIFIX is currently able to propose relatively simple fixes for designs that have undergone minor modifications.

This chapter is organized as follows. First, Section 7.1 describes how word-level e-graphs can be applied to produce a verification assistant, ROVERIFY. Section 7.2 applies ROVERIFY to the motivational example shown in Figure 7.1. Section 7.3 describes ROVERIFIX and applies it to a common error made in RTL design. Finally, Section 7.4 presents results on a set of benchmarks demonstrating overall verification runtime improvements when deploying ROVERIFY.

The work described in this chapter was published at FMCAD in 2023 [144]. This chapter contains the following novel contributions:

- a word-level e-graph framework that composes a set of sub-problems from local rewrites to assist FV tools,
- an e-graph extraction method minimizing the 'distance' between two designs,
- test cases showing an enhancement in capabilities over industrial tools, reducing the need for manual FV effort.

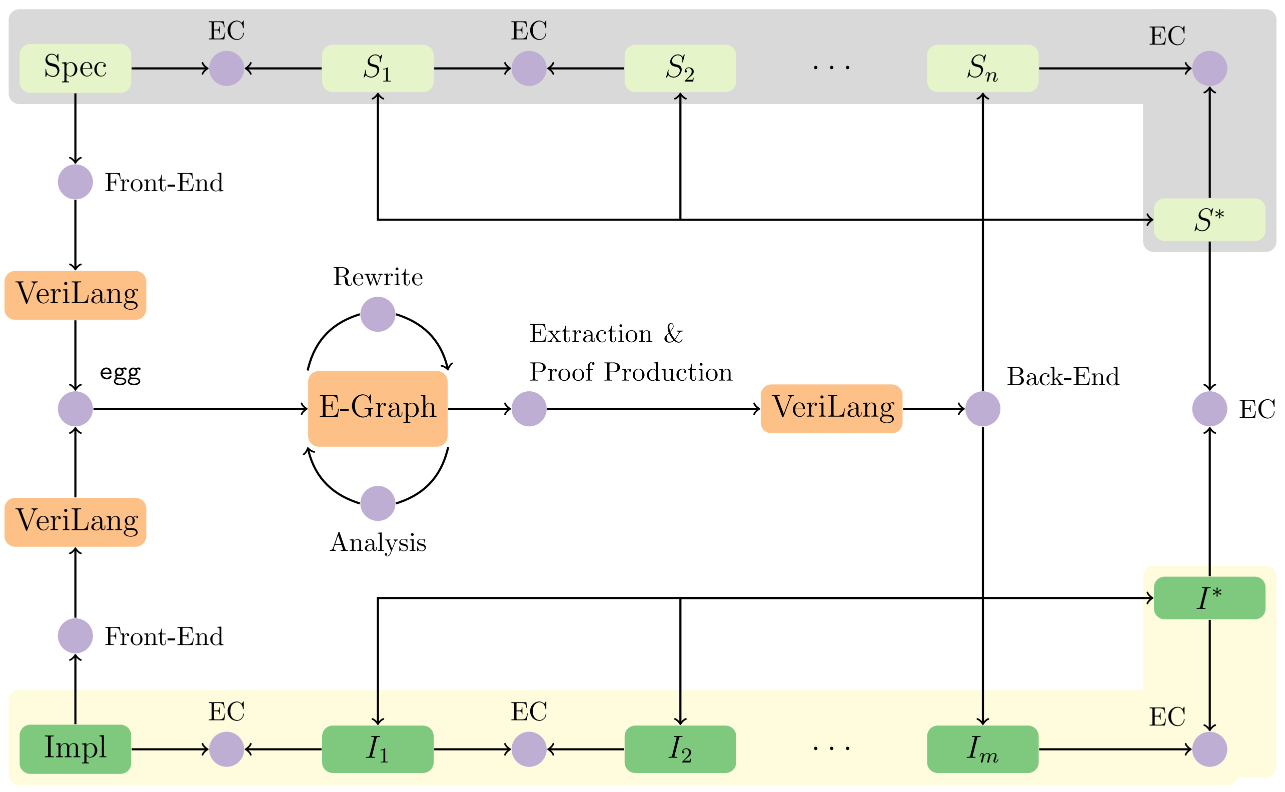


Figure 7.2: Flow diagram for the verification assistant, taking a specification, S, and implementation circuit design, I, in (System) Verilog. The designs are parsed and an e-graph is constructed. From the rewritten e-graph, two designs S^* and I^* are extracted along with intermediate designs forming a verification waterfall.

7.1 ROVERIFY: A Formal Verification Assistant

This section tackles the following problem, given two RTL designs, a specification, S, and an implementation, I, prove them equivalent or reduce the original EC problem to a simpler one to solve. Given this objective, e-graph rewriting can provide an efficient solution. Figure 7.2 illustrates the overall flow of the assistant. In this work, a particular commercial EC tool is used throughout, but any RTL2RTL EC tool could be substituted in its place.

7.1.1 E-Graph Initialization

Using the framework developed in the previous chapters, both RTL designs are parsed producing VeriLang expressions for S and I. In most e-graph applications built using egg, the e-graph is initialized with a single expression representing the design to be optimized. Here, the e-graph is initialized with both S and I. The common nodes are automatically shared by egg.

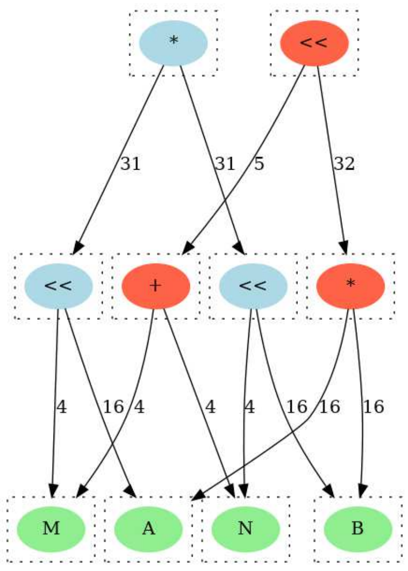


Figure 7.3: Initial e-graph representing two designs shown in Figure 7.1, a specification (blue) and implementation (red). Shared nodes are colored green. Edge labels denote bitwidths. All e-classes (dashed boxes) initially contain a single node.

Figure 7.3 represents the two designs shown in Figure 7.1 in a single e-graph. Colors indicate the design in which each node is used. Note that the designs initially only share the input variables and no intermediate signals. The following sections will discuss how, as the e-graph is grown, common intermediate signals can be discovered. Initializing the e-graph with both designs means that both designs can be simultaneously rewritten to find a common equivalent.

7.1.2 Bitwidth Dependent Verification Rewrites

The rewrites described throughout Chapters 3, 5 and 6 targeted optimization. It is natural that the verification rewrite set should include many of the optimization capabilities, since a key FV use case is for verifying the correctness of optimizations. However, the rewrite set should also incorporate additional verification specific rewrites that 'undo' optimizations. For example, it may be productive to include transformations that introduce redundant logic that enables further sharing. Whilst reversing optimization rewrites seems intuitive, this is non-trivial for conditional and dynamic rewrites that construct the right-hand side as a function of the matched pattern. The rewrite rules specifically designed to improve correlation with downstream logic synthesis tools are omitted.

Table 7.1: An example set of bitwidth dependent datapath verification rewrites. All rewrites are conditionally applied to ensure correctness. Bitwidth and signage information of operators and operands is omitted here for concision.

Name	Left-hand Side	Right-hand Side
Unmerge Shift	$a \ll (b + c)$	$(a \ll b) \ll c$
Mult Left Shift	$a \times (b \ll c)$	$(a \times b) \ll c$
Shift to Mult	$a \ll \text{const}$	$a \times 2^{\text{const}}$
Mult to Add	$a \times 2$	$a + a$

Table 7.1 describes the small set of additional verification specific rewrites learnt from experience using commercial EC tools. Several of these rewrites are the reverse of rewrites targeting optimization. The space of rewrites that 'undo' optimizations is less intuitive, so selecting valuable rewrites to include is challenging. The rewrites selected are relevant for the test cases presented here. The assistant is designed such that it is simple for users to extend the rewrite set with their own transformations that are applicable to their designs.

One important consideration is to ensure that few rewriting opportunities are missed. This is achieved by parameterizing the pattern matching left-hand side and applying the rewrites conditionally, following the condition synthesis flow described in Section 3.2.2. Missed rewriting opportunities can be the difference between a proven equivalence check and an inconclusive result, contributing to an observable brittleness of existing EC tools. We will see this in Section 7.2.

A challenge for RTL verification is that functional behavior is bitwidth dependent, for example the addition of two 8-bit values stored in an 8-bit and a 9-bit result differ in general but may be equivalent under certain design constraints. This challenge is resolved by reusing the interval analysis and bitwidth reduction rewrites described in Chapter 5. These rewrites detect and reduce operators to the minimum bitwidth required to store the result, hence normalizing the operations. Such techniques are also deployed in commercial tools [86], but, as observed in Chapter 4, program analysis on e-graphs is able to provide more precise abstractions.

Having defined a set of rewrites, ROVERIFY applies them to the e-graph initialized as described in Section 7.1.1. Rewrites are applied to both the specification and implementation designs simultaneously with the objective being to discover equivalent sub-expressions across the two designs.

7.1.3 Maximal Sharing Extraction

Once the e-graph has saturated or reached a user defined iteration limit, the e-graph represents two (potentially identical) sets of equivalent designs, one set of designs equivalent to the specification and one set of designs equivalent to the implementation. From the e-graph, ROVERIFY extracts two designs, $S^* \cong S$ and $I^* \cong I$ that share the maximum number of common nodes. If S and I are found in the same e-class, namely the tool found a path of rewrites between the two designs, then S^* and I^* are syntactically identical. If they are found in different e-classes, ROVERIFY extracts syntactically distinct S^* and I^* sharing as many of the common sub-expressions as is feasible from the e-graph.

To extract S^* and I^*, ROVERIFY first identifies which e-classes in the e-graph are associated with each design. As before, let $\mathcal{C}$ denote the set of all e-classes. Given a root e-class, r, a simple algorithm recursively constructs an associated $C_r \subseteq \mathcal{C}$. Starting from $C_r = \emptyset$, it iterates through each node in r, adding its children e-classes to C_r. The process continues, recursively visiting each of the child e-classes and iterating through the contained nodes until C_r stops growing. This construction is guaranteed to terminate for an e-graph comprised of a finite number of e-classes.

Letting s and i denote the e-classes containing S and I, respectively. The algorithm described above is used to construct $C_s \subseteq \mathcal{C}$, $C_i \subseteq \mathcal{C}$ and the shared e-class set, $C_{\text{shared}} = C_s \cap C_i$, which is used to identify the S^* and I^* that share the most common nodes. Define $C'_s = C_s \setminus C_{\text{shared}}$ and $C'_i = C_i \setminus C_{\text{shared}}$. In Figures 7.3 and 7.5, C'_s is highlighted in blue, C'_i is highlighted in red and C_{shared} in green.

The previous chapters sought to minimize hardware specific, PPA cost functions in the extraction phase. To maximize shared sub-expressions, a simple objective function is used that does

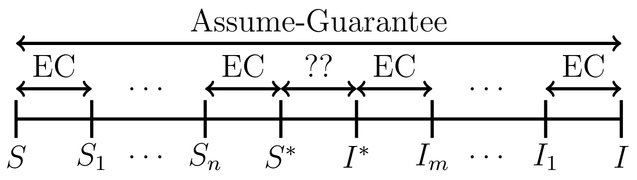


Figure 7.4: ROVERIFY extracted waterfall. An EC tool checks the equivalence of each intermediate step. The central equivalence check between S^* and I^*, which may not be true, may not be provable using the EC tool, but represents a simplified problem.

not depend on the particular type of an e-graph node $n \in \mathcal{N}$:

$$\text{shared}(n) = \begin{cases} K, & \text{if } \mathcal{C}(n) \in C_{\text{shared}}, \\ -1, & \text{otherwise}, \end{cases} \tag{7.1}$$

where $\mathcal{C}(n)$ returns the e-class containing the node n and K is the total number of e-classes in the e-graph. Maximizing this objective function ensures that the maximum possible number of nodes are shared. The negative scoring of unshared nodes ensures that amongst designs sharing the same number of nodes, the simplest one is prioritized. Once again, extraction is formulated as an ILP problem to correctly account for sub-expression reuse, modifying the objective function as shown in (7.2). The ILP introduced in Section 3.3.2 is extended, but the formulation is repeated below to avoid confusion. As before, define $\mathcal{N}$ to be the set of nodes and $E \subseteq \mathcal{N} \times \mathcal{C}$ the set of edges. Let $\mathcal{N}_c$ to denote the set of nodes in a given e-class c and introduce P_c to denote the set of parent nodes of c. In this case define the root set $\mathcal{S} = \{s, i\}$. With these definitions the problem formulation is the following:

$$\text{maximize} \quad \sum_{n \in N} \text{shared}(n) \cdot x_n \text{ subject to:} \tag{7.2}$$

$$\forall (n, c) \in E. \sum_{n' \in \mathcal{N}_c} x_{n'} \geq x_n \tag{7.3}$$

$$\forall c \in \mathcal{S}. \sum_{n \in \mathcal{N}_c} x_n = 1. \tag{7.4}$$

$$\forall c \in \mathcal{C}. \sum_{n \in \mathcal{N}_c} x_n \leq 1 \tag{7.5}$$

$$\forall c \in \mathcal{C} \text{ s.t. } P_c \neq \emptyset. \sum_{n \in \mathcal{N}_c} x_n \leq \sum_{n' \in P_c} x_{n'}. \tag{7.6}$$

Constraints (7.3) and (7.4) are as in Section 3.3.2 and guarantees that only valid designs are extracted. Additionally, since the objective function is now maximized, (7.5) is necessary to ensure at most one node in each e-class is implemented, whilst (7.6) avoids unused signals in the generated RTL. As before topological sorting variables avoid cycles in the e-graph.

As described in Section 2.4.2, `egg` includes a default greedy extraction method, which can be used to minimize a comparable objective function. Such an approach is faster but fails to correctly account for common sub-expressions so may generate designs that are not as 'close' as the ILP approach. The ILP approach is recommended for solving EC problems that will require manual intervention. This choice is left as a user-defined option.

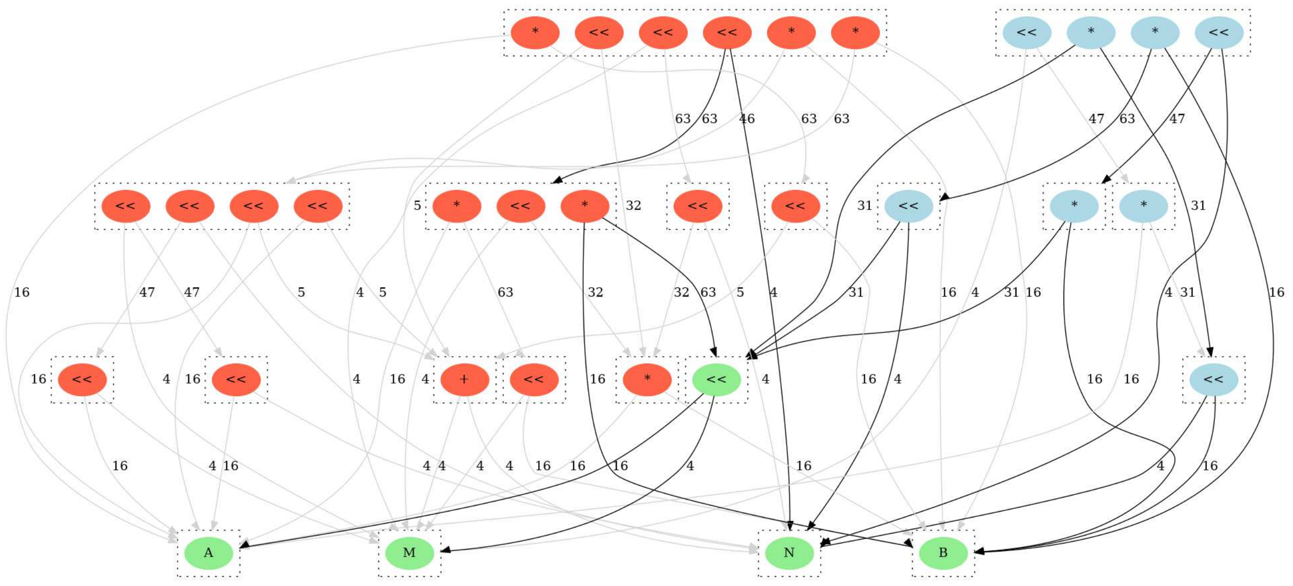
Using the proof production feature in `egg`, described in Section 2.4.2, two sequences of VeriLang expressions, separated by a single rewrite, are produced such that

$$S \cong S_1 \cong \ldots \cong S_n \cong S^* \text{ and } I \cong I_1 \cong \ldots \cong I_m \cong I^*.$$

To verify the correctness of each step and remove the need to trust ROVERIFY, the assistant converts each VeriLang expression to System Verilog. ROVERIFY then deploys the EC tool to formally verify the equivalence at each step as shown in Figure 7.4. If the EC tool can prove each step including $S^* \cong I^*$, then there is a chain of reasoning proving the equivalence of S and I. To ensure soundness of the chain of reasoning, a final "Assume-Guarantee" lemma proving $S \cong I$ is included, which uses all of the intermediate proofs (assuming they passed). This provides confidence that no gaps were left in the reasoning.

It is expected that all proofs of the form $S_j \cong S_{j+1}$ and $I_j \cong I_{j+1}$ will be trivial for the EC tool, since each step represents a single rewrite application. The $S^* \cong I^*$ proof may be more complex. If the EC tool is unable to prove $S^* \cong I^*$ then human intervention is required. However, provided ROVERIFY made some progress, the EC problem is simplified, as these designs share more common signals than the original S and I.

Proof decomposition also aids performance. Since each intermediate proof is independent they can be proven in parallel. The solver configuration is customized for each intermediate proof,


(a) E-graph after two iterations of rewriting. Designs sharing an intermediate signal are highlighted with black arrows.

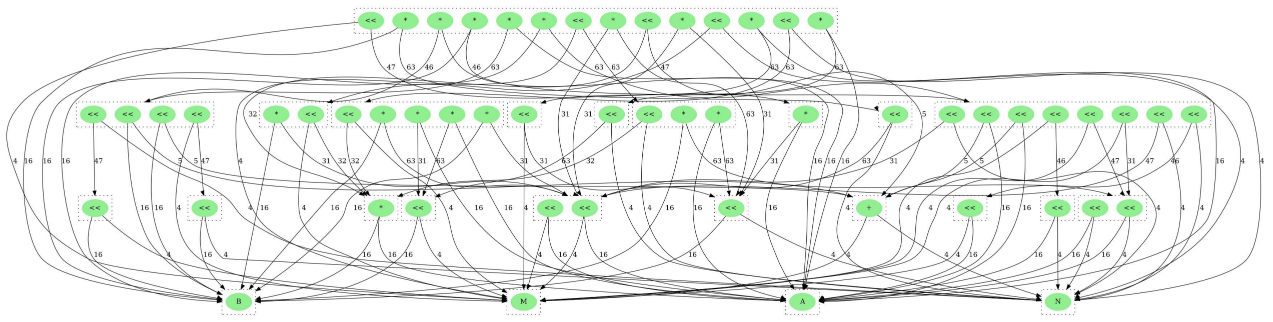

(b) E-graph after three iterations of rewriting (77 nodes), where S and I have been merged into the same e-class.

Figure 7.5: Stages of e-graph growth starting from the initial e-graph in Figure 7.3.

since it is simple to map each rewrite to an optimal solver setup. For example, the commercial tool provides a set of solve scripts that handle proof orchestration with different capabilities. These scripts can be enabled by the user. ROVERIFY contains a mapping from rewrites to the most efficient solve script. With limited effort ROVERIFY can target additional solvers.

7.2 ROVERIFY Case-Study

A case-study of a real world problem gives an example of where this technique proves beneficial. The following results use an up-to-date version of the commercial EC tool running on SLES 12 on Intel Xeon W-2155 CPUs.

The designs shown in Figure 7.1 are alternative ways to implement floating-point multiplication of denormal numbers. More precisely, given two denormals $2^{1-bias} \times 0.\text{mant}_a$ and $2^{1-bias} \times 0.\text{mant}_b$, the product of their mantissas is often reduced to a standard non-denormal multiplication by normalizing the values, expressing it as either $(\text{mant}_a \ll m) \times (\text{mant}_b \ll n)$ or equivalently as $(\text{mant}_a \times \text{mant}_b) \ll (m+n)$, where $m = LZC(\text{mant}_a)+1$, $n = LZC(\text{mant}_b)+1$ and $LZC(\cdot)$, as in Chapter 5, is the leading zero counter function.

In three iterations of rewriting ROVERIFY applies a sequence of rewrites such that the specification and implementation are found within the same e-class. The progress of the e-graph can be seen in Figures 7.5. After two iterations of rewriting the first shared signal is detected, see the green left-shift in Figure 7.5a, where the initial specification and designs sharing the green node are highlighted with brighter arrows. The e-graph shown in Figure 7.5b, after three iterations of rewriting, contains only green nodes, since ROVERIFY was able to apply a sequence of rewrites such that the original root nodes of S and I were merged into the same equivalence class. As a result, all e-classes are shared, meaning $\mathcal{C}_{\text{shared}} = \mathcal{C}$.

From the final e-graph, Figure 7.5b, ROVERIFY then extracts identical S^* and I^* along with the sequence of rewrites that were applied to reach it. The rewrites transforming S to I are summarized below, omitting trivial bitwidth alteration and commutativity steps.

$$\begin{aligned}
& (A \times B) \ll (M+N) \rightarrow && (7.7)\\
\textit{Unmerge Left-Shift} \quad & ((A \times B) \ll N) \ll M \rightarrow && (7.8)\\
\textit{Left-Shift Mult} \quad & (A \times (B \ll N)) \ll M \rightarrow && (7.9)\\
\textit{Left-Shift Mult} \quad & (A \ll M) \times (B \ll N) && (7.10)
\end{aligned}$$

ROVERIFY runs in 0.14 seconds, growing an e-graph comprised of 77 nodes. Unfortunately, the commercial EC tool is unable to prove the "Left-Shift Mult" and "Mult Left-Shift" transformations when non-uniform bitwidths are used. ROVERIFY automatically resolves this by further decomposing this proof, automatically inserting an additional intermediate step with standardized bitwidths. The additional step helps the EC tool to converge in less than 0.1

second, therefore, it may be the case that there is a rewrite rule that only gets applied under certain parameterizations in the internal EC proof engine.

Including commutativity and bitwidth alteration rewrites, ROVERIFY generated a total of 20 intermediate equivalence checks (including the "Assume-Guarantee" lemma). All intermediate proofs and the completeness lemma are proven in 0.1 seconds by the EC tool. When passed the original EC problem, $S \cong I$, the tool did not converge within 24 hours.

7.3 ROVERIFIX: Automatic Bug Fixing

In contrast to ROVERIFY, which assists existing tools with proof convergence, ROVERIFIX can automatically provide bug fixes for broken implementations. More precisely, given a specification, S and an implementation, I, that are proven to not be equivalent, $S \ncong I$, ROVERIFIX finds the fixed design I' nearest to I, such that $I' \cong S$. The aim is to minimize the distance between I and I' because an implementation typically contains optimizations that it is desirable for the fixed design, I', to retain. This is similar to the intent behind corrections applied for an Engineering Change Order (ECO).

Figure 7.6 describes the ROVERIFIX flow, that includes two phases of e-graph rewriting. First, an e-graph is initialized with the specification S, and the same ROVERIFY rewrites are applied to this e-graph. The objective here is to explore a large space of design candidates that are equivalent to S. To then localize the fix, VeriLang is extended with a new operator, `FIX`, that takes two arguments, a correct expression and an expression that is to be replaced with the correct expression.

In the second phase, $\texttt{FIX}(S, I)$ is added to the e-graph, and only the rewrites given in Table 7.2 are applied. These rewrites propagate the `FIX` operators down through the e-graph, maximally sharing sub-expressions between expressions equivalent to S and I. Intuitively, pushing `FIX` nodes through the e-graph localizes the source of the bug. Once the e-graph has saturated, the extraction process minimizes the area cost function introduced in Chapter 3, giving each `FIX` node a cost of one, since this corresponds to the minimum area bug fix. Since the `FIX`

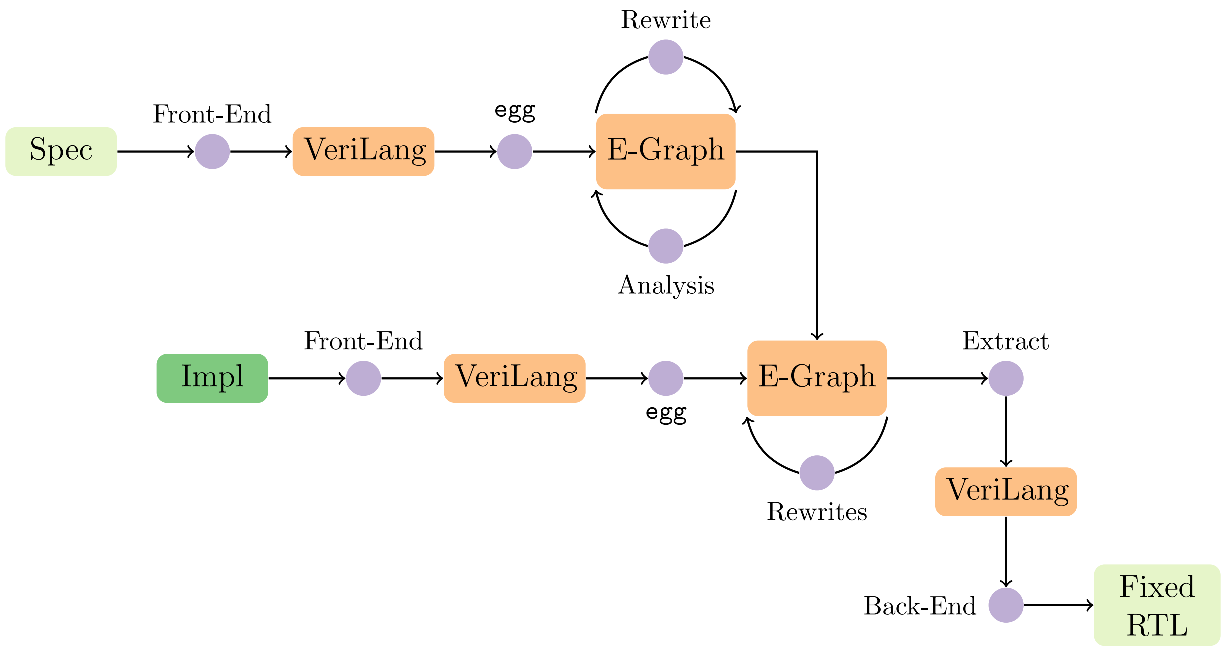


Figure 7.6: Flow diagram for ROVERIFIX, taking a specification, S, and implementation circuit design, I, in System Verilog. Two separate phases of e-graph rewriting are applied, and the minimal fix design, I', is extracted.

Table 7.2: `FIX` propagation and removal rewrites, that remove `FIX` nodes when no correction is necessary. `op` denotes any VeriLang operator.

Name	Left-hand Side	Right-hand Side
Propagate Fix	$\texttt{FIX}(a \texttt{ op } b, c \texttt{ op } d)$	$\texttt{FIX}(a,c) \texttt{ op } \texttt{FIX}(b,d)$
Fix Same	$\texttt{FIX}(a,a)$	a

operations do not directly map to any Verilog operator, ROVERIFIX generates Verilog for only the first operand of each `FIX` node, since this corresponds to a design that is functionally equivalent to the specification. ROVERIFIX also reports the applied corrections to the user via VeriLang expressions.

The extracted VeriLang expression may contain multiple `FIX` operators, corresponding to several independent corrections. There is no limit on the size of a correction, and if no sharing was possible then the minimal ROVERIFIX correction will be to replace the implementation by the original specification. The difference in the area cost, between the implementation I and fixed design I', gives an estimate of the size of the correction.

Figure 7.7 provides a motivational example, where the designer has made a common mistake, by using an incorrect bitwidth. In particular, add_8bit is defined to be an 8-bit signal, such that

```
module spec(A,B,C,out);
input logic  [7:0] A, B, C;
output logic [9:0] out;
wire         [8:0] add_right;

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

endmodule
```

(a) Specification design.

```
module impl(A,B,C,out);
input logic  [7:0] A, B, C;
output logic [9:0] out;
wire         [7:0] add_8bit;

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

endmodule
```

(b) Buggy implementation design.

Figure 7.7: Motivational example for ROVERIFIX, where the specification and implementation designs are not functionally equivalent.

the carry-out of the addition is discarded. In the first rewriting phase, ROVERIFIX applies associativity of addition to the specification, then, in the second rewriting phase, `FIX` operators are pushed through the design. The minimal extracted correction in this case, is given by `FIX`$(9, 8)$, which corresponds to incrementing the bitwidth of the add_8bit signal.

A thorough evaluation of ROVERIFIX is still to be completed, but to demonstrate real-world value, ROVERIFIX was applied to an Intel developed floating-point norm component calculating $1/\sqrt{1+x^2}$. Simulation discovered that the component incorrectly processed input NaNs. A correct specification and the buggy implementation were passed to ROVERIFIX, which correctly identified a FIX minimizing the circuit area overhead in just 0.19 seconds.

7.4 ROVERIFY Results

The results section will only focus on the evaluation of ROVERIFY. Having demonstrated how ROVERIFY can provide the intermediate steps transforming a previously inconclusive proof into one solved in under one second, ROVERIFY's impact on verification runtimes is now evaluated across a datapath optimization benchmark set.

Table 7.3: A comparison of commercial EC tool performance with and without ROVERIFY generated intermediate proofs. We report the baseline EC tool performance, solving the original EC problem. We compare this to the sum of the runtime of ROVERIFY and the runtime of the assisted EC proofs. Runtimes are in seconds. The Box Filter benchmark used the greedy extraction method, which reduces the extraction runtime mitigating the significant e-graph growth.

Benchmark	Baseline EC	ROVERIFY	EC with assistance	Total	Speedup
ADPCM Decoder	**0.68**	0.38	0.49	0.87	0.78
H-264 VBSME-4	7.93	7.04	0.71	**7.75**	1.02
H-264 VBSME-8	93.13	14.3	0.20	**14.50**	6.42
FIR Filter	5.50	3.49	0.79	**4.28**	1.29
Box Filter	79.56	16.10	1.61	**17.71**	4.49
Case Study	-	0.14	0.10	**0.24**	-

7.4.1 Benchmark Selection

Benchmarks are taken from [24] and original and optimized RTL is implemented for those designs that are fully described in this paper. The depth-8 FIR Filter and ADPCM Decoder are also used in Chapter 3, but this chapter uses hand-written rather than ROVER-generated optimized RTL, to ensure a fair evaluation. The chapter introduces two instances of a kernel from the H.264 VBSME (variable block size motion estimator), which correspond to absolute difference summation trees of size four and eight, $\sum_i |a_i - b_i|$. This design was not used throughout the previous chapters as the logic synthesis results were difficult to explain using the limited reports available. The case study, as described above, was also optimized in Chapter 3. The Box Filter is an Intel provided benchmark, not optimized in previous chapters due to the scalability concerns highlighted by the MCM examples in Chapter 3. The same scalability concerns will also be apparent in this chapter. The Box Filter is a reconfigurable square filter, sampling four pixels at a time. The dataflow graph for this design is shown in Figure 7.8. The optimized design deploys constant factorization and mux rewriting which is relatively challenging for the EC tool to prove.

The benchmarks include a range of arithmetic and logical operators, representative of typical RTL optimizations that may be performed by-hand or by a specialized datapath optimization tool. For each benchmark, ROVERIFY is run until either it discovers a complete path between specification and implementation or it deploys five iterations of rewriting, whichever comes first.

Since the e-graph applies all rewrites in parallel at each iteration, many parts of the designs can be simultaneously transformed in each iteration.

In these results, the EC tool does not report any increase in the initial compilation time, which is less than a second for all cases presented here. The runtime is reported from when the solvers start running. For the baseline, all the EC tool's solvers are deployed in parallel taking the minimum proof time. For the ROVERIFY results, the maximum time taken to solve a single sub-problem is reported, since each proof can be run in parallel. In practice, the industrial tool's multi-processor environment introduced runtime overhead that was not related to the proof. Namely, running a proof on a server grid produced unpredictable runtime results due to the license checks and interactions with the workload management software.

Table 7.3 presents the performance impact of the assistant on the total verification time, whilst Table 7.4 provides statistics that describe ROVERIFY's resource utilization. In the first example, ADPCM Decoder, the EC tool efficiently proves the correctness of the two designs, meaning that the overhead of ROVERIFY is detrimental, increasing total runtime. It is worth noting that the intermediate proofs do help to reduce the solve time. The two H-264 designs suggest that ROVERIFY's advantage may grow with the design complexity.

In the remaining benchmarks, ROVERIFY has a net positive impact on total runtime. The introduction of intermediate proofs reduces the EC solve time by up to 465x, when just comparing the EC tool runtimes and discounting ROVERIFY's runtime. Including the runtime to generate the intermediate proofs, the total verification time is reduced by up to 6x. In most cases, the EC tool solves each of the intermediate proofs in less than 0.5 seconds as each step represents a single local modification to the design. ROVERIFY effectively selects the most optimal solver orchestration script per intermediate proof, which helps performance. This is possible because ROVERIFY understands what transformation has been applied at each stage. Such an approach avoids wasted compute resources, since there is no need to run different solvers in parallel for each of the intermediate problems.

The Box Filter is the only example where the assistant is unable to find a complete path. This verification problem may require additional rewriting iterations or entirely new rewrites to

Table 7.4: Summary of ROVERIFY resources and results across the benchmarks. We report the number of rewriting iterations, the final e-graph size in terms of node count, the number of intermediate proofs generated, and whether the e-graph found a complete path of rewrites between S and I.

Benchmark	Iterations	E-graph Nodes	Num. Proofs	Full Path
ADPCM	3	469	20	Y
VBSME-4	5	5640	26	Y
VBSME-8	5	5800	46	Y
FIR Filter	5	4700	23	Y
Box Filter	5	21400	115	N
Case Study	3	149	20	Y

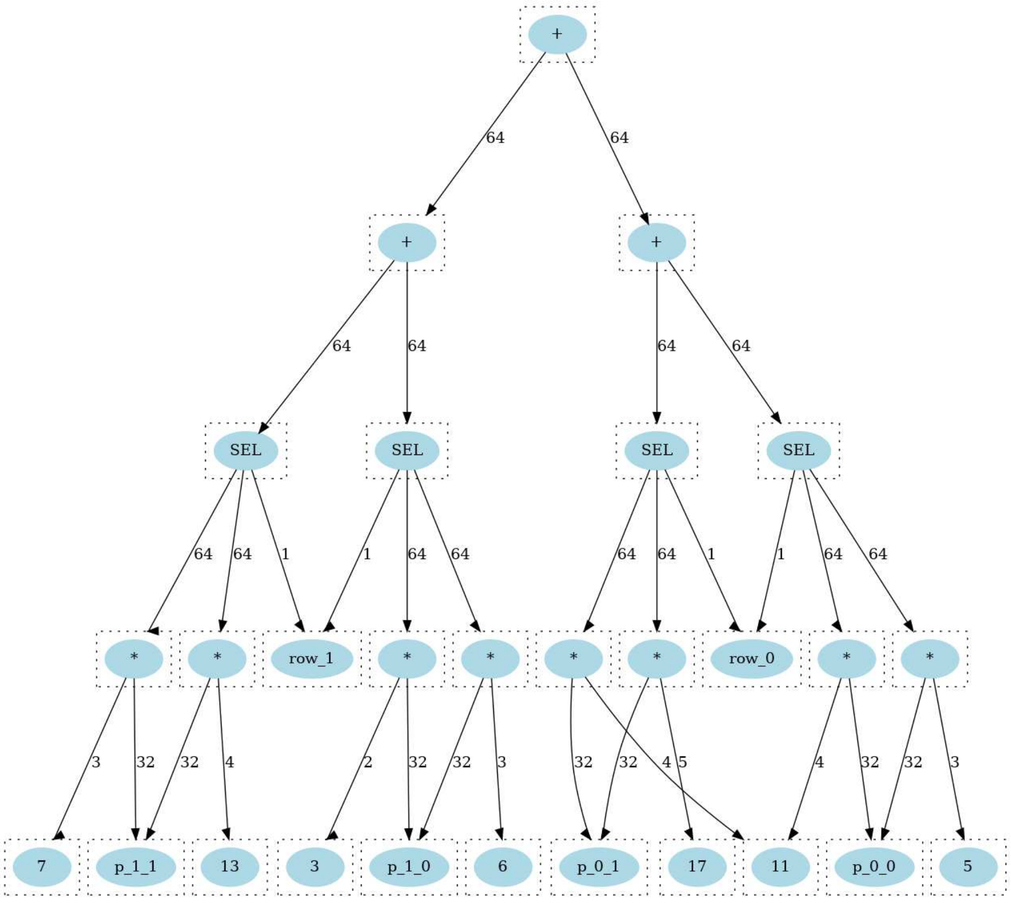


Figure 7.8: Dataflow graph of the initial Box Filter design. The `SEL` nodes represent muxes.

bridge the gap between the specification and implementation designs. To minimize ROVERIFY's runtime, the faster greedy extraction method is deployed. To solve the EC problem, $S^* \cong I^*$, one of the slower but more powerful solver orchestration scripts is used. In this case, the $S^* \cong I^*$ EC problem takes significantly longer to prove than the other sub-problems. In general, as ROVERIFY's e-graph grows through more iterations of rewriting, it is reasonable to expect to find S^* and I^* that are increasingly close. Table 7.3 reports Box Filter results based on five iterations of rewriting. If ROVERIFY is instead limited to three iterations of rewriting, the assistant's runtime is reduced from 16 seconds to 2 seconds. The intermediate proofs generated by this smaller e-graph can be proven in 1.12 seconds, reducing the total verification time to approximately 3 seconds, corresponding to a 24x speedup over the baseline. These results highlight a tradeoff between proof decomposition and the proof itself.

In addition to the commercial EC solver used above, the open-source SymbiYosys equivalence checker [87] was also tested. However, for the instances tried, SymbiYosys was not able to solve any equivalence problems since the tool is SAT/SMT based and does not handle datapath problems efficiently. A key advantage of a rewrite based approach is that performance is not affected by bitwidths, whilst SAT-based solvers will suffer from exponential slowdowns as the bitwidths are increased. This approach is promising because the full power of a SAT or SMT solver is not typically needed on the entire design, meaning that a specialized tool that does not target notions of completeness can prove valuable.

7.5 Summary

This chapter takes advantage of the RTL rewriting framework developed for ROVER, applying it to datapath equivalence checking. Having observed limitations in commercial EC tools, an automated formal verification assistant, ROVERIFY, was developed, that enhances the capabilities of industrial tools. By incorporating both the specification and implementation into a single e-graph, ROVERIFY simultaneously rewrites both designs to efficiently identify common equivalent sub-expressions. From the e-graph, the tool extracts a sequence of intermediate designs, decomposing the complete proof into a sequence of smaller sub-proofs which can be checked by *trusted* tools. In cases where the assistant is unable to identify a complete path between the specification and the implementation designs, the e-graph rewriting may still reduce the equivalence checking to a simpler sub-problem. This enables FV engineers to focus on the challenging core of the verification task and helps the EC tool to identify additional internal equivalence pairs automatically, reducing the complexity of the overall equivalence check. To aide in the debug process, ROVERIFIX, a tool to automatically correct bugs in broken implementations, was also developed. These diverse applications demonstrate the value of the underlying RTL rewriting framework, providing a platform upon which further extensions could be developed.

Whilst ROVERIFY has demonstrated potential, to fully realize the technology's capabilities, it must be integrated into a complete EC tool, providing access to a wider range of solvers and providing the front-end interfaces to consume designs in alternative languages, e.g. C++. The final chapter will discuss the future plans for the advances in the technology and how to widen adoption of the techniques presented.

Chapter 8

Conclusion

The work described in this thesis is inspired by the engineers working in industry, writing HDL code every day, and, as such, has addressed practical challenges that affect a large part of the semiconductor sector. Most of the research stems from an observation that manual RTL optimization and verification can be phrased as a rewriting problem, but a rewriting problem with a non-monotonic profitability model. Exploration at the word-level RTL abstraction has yielded significant PPA improvements, revisiting a subject that has not received much attention for over a decade [24, 12]. Despite being closed-source, the ROVER work now provides a baseline used (and re-implemented) by independent academics to evaluate the performance of their proposed word-level RTL optimization methodology [16]. The ROVER tool has been used to optimize a number of production hardware modules, but has also been an invaluable education tool to teach engineers how to improve their designs. Whilst the techniques have demonstrated promising results, the tools have yet to gain meaningful adoption within Intel. To significantly broaden adoption, it will be necessary to develop a clean user experience and robust front-end, increasing the scope of designs the tools are able to consume. Section 8.3 will describe the future roadmap of the tools developed throughout this thesis.

The tools developed throughout this thesis (ROVER/ROVERIFY/ROVERIFIX) have demonstrated the great potential in rewriting and, in particular, the application of equality saturation to hardware design challenges. These tools have contributed to a growing list of equality saturation based circuit design methodologies [115, 72, 145, 146], and position papers arguing for the broader adoption of equality saturation in EDA [117, 118]. Whilst equality saturation may be broadly applicable, it is important to recognize that it may not always be the best solution. For example, prior work has highlighted scalability concerns in application to gate-level circuit design [147], which others resolved by incorporating machine learning techniques [115]. Chapter 6 also highlighted a class of optimizations that are fundamentally incompatible with a local rewriting approach.

Beyond hardware design, Chapters 4 and 5 describe broadly applicable contributions to the e-graphs and equality saturation community. Indeed, the connection between abstract interpretation and e-graph rewriting inspired the `egg` developers to teach interval analysis as part of their PLDI 2022 tutorial.[1] The work described in Chapter 5 provides an approach to encoding context within the e-graph, which has become a subject of interest for several researchers [104, 105]. A comparison of these alternative methods would be a valuable contribution to better understand the strengths and weaknesses of each approach.

The remainder of the thesis will survey several areas of e-graph research with unresolved challenges, promising directions in circuit design, and discuss the future development of the tools described throughout.

8.1 E-Graph and Equality Saturation Outlook

The e-graph community of users and researchers has grown significantly over the duration of my PhD, with substantial advances in the field. This growth has brought into sharp focus two key scalability questions. In particular, scalability of both the e-graph search and extraction from the e-graph. The efficiency of e-graph search has been improved by unifying equality saturation and datalog [148], modifying the foundational algorithms and data representations. Others have modified the orchestration of e-graph search, investigating custom rewrite schedules [146] and iterative e-graph search [149]. These approaches no longer construct and explore a single e-graph

[1]https://github.com/egraphs-good/egg-tutorial-pldi-2022

as originally shown in Figure 2.5, but decompose this process into several phases of e-graph exploration. However, this re-introduces a phase-ordering problem, removing one of the main advantages of equality saturation. This observation highlights a tradeoff between exploring a large space of equivalent expressions, much of which provides no benefit, and finding the minimum cost expression. Constructing a generally applicable and scalable orchestration of e-graph search would be an invaluable contribution.

To address the second challenge, efficient common sub-expression aware extraction, there has been a more organized community effort, with the compilation of the extraction gym,[2] a set of community contributed e-graph benchmarks. In fact, several ROVER benchmarks have also been contributed. Beyond benchmarks, researchers are encouraged to contribute novel extraction methods to this repository [150, 151], which can be evaluated using a standardized set of benchmarks. As in the case of encoding context within the e-graph, each approach has strengths and weaknesses, so selecting the right extraction method is a choice made by users of equality saturation. Once again, there is still much to explore in this domain, and, in particular, the community lacks effective metrics that an equality saturation user can measure to better understand their e-graphs.

Beyond expanding the scale of problems that equality saturation can be applied to, there is also interest in expanding the scope. By definition, equality saturation explores equivalent programs, but it is natural to consider different relationships between programs. For example, in hardware, it is common to tradeoff accuracy for performance. An equality saturation based approach is unable to explore such a design space, since it is restricted to an equivalence relation. Such programs are related by a different relation, a partial order, which raises the following question. Is it possible to construct an alternative data structure capable of exploring a different underlying relation between programs?

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

8.2 Directions in Digital Circuit Design

The research directions described in Section 8.1 represent theoretical contributions, that will facilitate and enhance a range of applications. From an applications perspective, this thesis has taken steps towards bridging the gap between high-level circuit optimization techniques and low-level logic synthesis for datapath circuits. One contribution that would inspire further research in this area and facilitate effective comparison is an industrially relevant benchmark suite. Such benchmark suites have provided the foundations for consistent improvements in both SMT solvers [152] and logic synthesis tools [153]. At the conclusion of this PhD, a large collection of datapath benchmarks has been accumulated which will be extended and combined with other collections [16], to provide the datapath synthesis community with a standardized benchmark set. Building upon these datapath benchmarks, there are several future circuit design challenges, which I plan to tackle.

8.2.1 Pipelining and Retiming

This thesis has primarily focused on combinational RTL, yet most production RTL is pipelined, inserting registers that separate a computation across several clock cycles. Often, architectural design and scheduling are separate tasks. For example, in HLS tools datapath optimizations are performed before passing the optimized designs onto a scheduling engine. Phrasing datapath design and scheduling as a co-optimization problem has the potential to greatly improve the quality of pipelined circuits. Optimal register placement in an implementation designed without considering a pipeline structure will yield a sub-optimal circuit.

Building on Chapter 6, it is trivial to phrase register retiming as an e-graph rewriting problem, where alternative register placements can be evaluated once their consequences are known. In combination with datapath transformations, such an approach can co-optimize the datapath and schedule. Tying this co-optimization to a circuit power model will further enhance the interaction between register placement and power optimization. Deploying the rewrite scheduling approach described in Section 8.1, register movement can be limited, as recommended by the ABC developers [154].

8.2.2 Approximate Computing

All of the work described in this thesis has maintained bit-accurate equivalence of the circuit outputs, but as suggested in Section 8.1, hardware designers are often not subject to such strict accuracy constraints. Error tolerance is particularly prominent in graphics hardware, since the directX specifications [155] and standard compliance tests permit some freedom in the result. A more substantial contribution will extend the e-graph framework to explore approximately equivalent circuits. Namely, to incorporate transformations that do not preserve equivalence, but produce results within some error threshold. Such transformations are relevant in many aspects of hardware design [156, 64], for example a faithfully rounded circuit may produce one of two valid outputs for a given input [157]. The key challenges in using an e-graph representation were identified above, but an ability to explore and quantify the accuracy performance tradeoffs would be invaluable in hardware design.

8.3 Software Roadmap

Chapter 1 described a collection of software libraries and applications that have been developed throughout the PhD. A simplified hierarchy is repeated in Figure 8.1. An ambitious goal, is for all Intel Graphics RTL to be run through ROVER, allowing us to quantify the technique's impact on a diverse code base. To achieve this, ROVER must first become more robust with a greatly improved front-end and stable analyses. Once ROVER can consume more code, it will be possible to perform a broad evaluation as ROVER will also be able to capture wider context, such as late arriving module inputs. Groups that have re-implemented e-graph based RTL optimization in combination with open-source logic synthesis have shown that the technique yields promising results across an extensive benchmark suite [16].

A secondary objective is to build a developer base within Intel. In particular, `rtl2egg` is

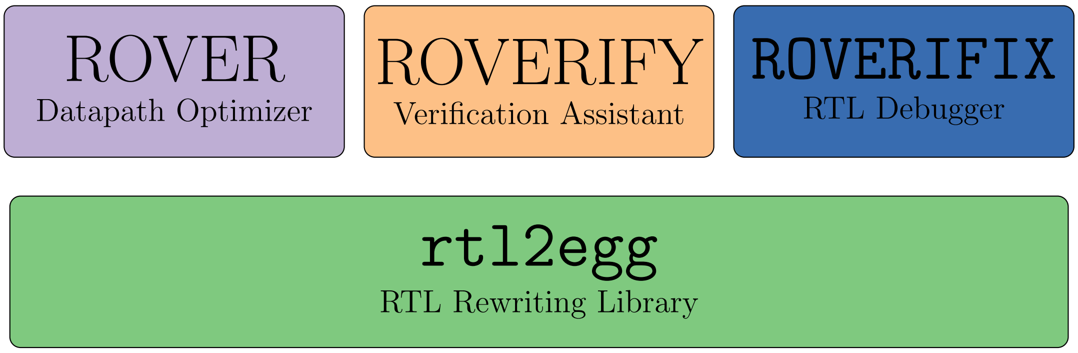


Figure 8.1: A diagrammatic representation of the library and tool suite developed.

designed to be an extensible RTL rewriting library, providing an `egg` like interface on which further applications can be built. An extensible library is only adopted when paired with mature documentation and tutorials. Both these items are in development currently. A broader user base will certainly raise new challenges and highlight areas to improve within the library.

Finally, it is clear that ROVERIFY and ROVERIFIX would benefit substantially from integration into a broader verification suite [87]. In particular, this would allow us to deploy e-graph rewriting directly on the verification tool's internal representation, avoiding the need to communicate between tools via Verilog. This would also greatly enhance the power of ROVERIFY and ROVERIFIX as existing suites orchestrate a collection of solvers that could be combined with e-graph rewriting.

8.4 Final Remarks

To conclude the thesis, I wish to highlight a broader challenge that will gate the wider adoption of high-level digital circuit design flows. Working top-down, the programming languages community continues to develop new languages and HLS frameworks [158, 159], yet the vast majority still generate Verilog to be consumed by a logic synthesis tool. Working bottom-up, there has been extensive work on low-level logic minimization, implemented in tools such as ABC [160] and libraries such as those developed at EPFL [161]. Between these two abstractions, tools such as Yosys [9] have made progress on higher-level optimizations, yet still the gap is substantial. Commercial HLS vendors have identified this issue and include call-outs to their proprietary logic synthesis tools [6]. Meanwhile, in one leading HLS research group, they have started to observe significant advantages by guiding elements of low-level design optimizations via high-level re-structuring [162].

This discussion leads to two observations. First, Verilog was not designed to be an intermediate representation for a high-level design flow. Second, given their configuration complexity, logic synthesis tools are not well-suited for use in a tool-chain. These observations point towards one ambitious solution. An integrated high-level design flow, that bridges the gap and constructs a clean path from high-level languages down to a netlist representation. There has been some early explorations in this direction [163, 164], but building a mature flow will require a more sustained effort. The key benefit would come from an ability to maintain high-level design information throughout the abstraction lowering, improving the correlation between high- and low-level design decisions whilst also simplifying the verification challenge.

References

[1] European Commission, "European Chips Act," 9 2023. [Online]. Available: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age/european-chips-act_en

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

[3] D. R. Coelho, *The VHDL Handbook*, 1st ed. Kluwer Academic Publishers, 1989.

[4] M. C. Mcfarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[5] 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.

[6] Cadence, "Stratus HLS," 2023. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[7] Xilinx, "Vitis HLS," 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

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

[9] C. Wolf and J. Glaser, "Yosys-A Free Verilog Synthesis Suite," in *Proceedings of Austrochip*, 2013. [Online]. Available: https://yosyshq.net/yosys/files/yosys-austrochip2013.pdf

[10] Cadence Design Systems, "Jasper Formal Verification Platform," 2024. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html

[11] D. W. Knapp, *Behavioral synthesis: digital system design using the synopsys behavioral compiler*. USA: Prentice-Hall, Inc., 1996.

[12] A. K. Verma, P. Brisk, and P. Ienne, "Challenges in automatic optimization of arithmetic circuits," in *19th IEEE Symposium on Computer Arithmetic*, 2009, pp. 213–218.

[13] ——, "Variable latency speculative addition: a new paradigm for arithmetic circuit design," in *Proceedings of the Conference on Design, Automation and Test in Europe*. Munich: Association for Computing Machinery, 2008, pp. 1250–1255.

[14] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, vol. 44, no. 1. Association for Computing Machinery, 2009, pp. 264–276.

[15] 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. Association for Computing Machinery, 2021.

[16] X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu, "RTLRewriter: Methodologies for Large Models aided RTL Code Optimization," 2024. [Online]. Available: https://arxiv.org/abs/2409.11414

[17] K. W. Ho, S. T. Chung, T. F. Chen, Y. W. Fan, C. Cheng, C. H. Liu, and J. H. R. Jiang, "WolFEx: Word-Level Function Extraction and Simplification from Gate-Level Arithmetic Circuits," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE/ACM, 2023, pp. 1–9.

[18] G. D. Micheli, *Synthesis and optimization of digital circuits.* McGraw-Hill Higher Education, 1994.

[19] Cadence, "Genus Synthesis Solution," 2024. [Online]. Available: https://www.cadence.com/en_US/home/resources/datasheets/genus-synthesis-solution-ds.html

[20] T. Ajayi, D. Blaauw, T.-B. Chan, C.-K. Cheng, V. A. Chhabria, K. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Li, Z. Liang, U. Mallappa, P. Penzes, G. Pradipta, S. Reda, A. Rovinski, K. Samadi, S. S. Sapatnekar, L. Saul, C. Sechen, V. Srinivas, W. Swartz, D. Sylvester, D. Urquhart, L. Wang, M. Woo, and B. Xu, "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain," *Government Microcircuit Applications and Critical Technology Conference*, 2019.

[21] R. Zimmermann, "Datapath synthesis for standard-cell design," in *19th IEEE Symposium on Computer Arithmetic*, 2009, pp. 207–211.

[22] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. AK Peters/CRC Press, 2001.

[23] Synopsys, "Coding Guidelines for Datapath Synthesis," Synopsys, Mountain View, Tech. Rep., 12 2019.

[24] 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.

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

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

[27] 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.

[28] R. I. Hartley, “Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 677–688, 1996.

[29] M. D. Ercegovac and T. Lang, *Digital arithmetic*, 1st ed. Morgan Kaufmann, 2003.

[30] F. De Dinechin, S. I. Filip, M. Kumm, and A. Volkova, “Towards Arithmetic-Centered Filter Design,” in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, vol. 2021-June, 2021, pp. 115–118.

[31] M. Kumm, “Optimal constant multiplication using integer linear programming,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, pp. 567–571, 2018.

[32] 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, pp. 1976–1988, 2023.

[33] 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.

[34] J. E. Volder, “The CORDIC Trigonometric Computing Technique,” *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959.

[35] B. Orloski, S. Coward, and T. Drane, “Automatic Generation of Complete Polynomial Interpolation Design Space for Hardware Architectures,” in *Proceedings of the 28th Asia and South Pacific Design Automation Conference.* Tokyo: Association for Computing Machinery, 2023, pp. 573–578.

[36] J. Detrey and F. De Dinechin, “Table-based polynomials for fast hardware function evaluation,” in *International Conference on Application-Specific Systems, Architectures and Processors.* IEEE, 2005, pp. 328–333.

[37] R. Zimmermann and D. Q. Tran, “Optimized synthesis of sum-of-products,” in *37th Asilomar Conference on Signals, Systems & Computers*, vol. 1, 2003, pp. 867–872.

[38] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic.* Cham: Springer International Publishing, 2024.

[39] 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 *58th ACM/IEEE Design Automation Conference (DAC)*, vol. 2021-December. San Francisco, CA: IEEE, 2021, pp. 853–858.

[40] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, “BetterV: Controlled Verilog Generation with Discriminative Guidance,” in *Proceedings of the 41st International Conference on Machine Learning.* Vienna: JMLR.org, 2 2024.

[41] A. Beaumont-Smith and C.-C. Lim, “Parallel prefix adder design,” in *Proceedings 15th IEEE Symposium on Computer Arithmetic*, 2001, pp. 218–225.

[42] L. Dadda, “Some schemes for parallel multipliers,” *Alta frequenza*, vol. 34, pp. 349–356, 1965.

[43] C. S. Wallace, “A Suggestion for a Fast Multiplier,” *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.

[44] J. Miao and S. Li, “A design for high speed leading-zero counter,” in *Proceedings of the International Symposium on Consumer Electronics, ISCE*, 2017, pp. 22–23.

[45] Synopsys, “DesignWare Library,” 2024. [Online]. Available: https://www.synopsys.com/designware-ip/soc-infrastructure-ip/designware-library.html

[46] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim, “Reduced latency IEEE floating-point standard adder architectures,” *Proceedings 14th IEEE Symposium on Computer Arithmetic*, pp. 35–42, 1999.

[47] J. Sohn and E. E. Swartzlander, “Improved architectures for a fused floating-point add-subtract unit,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 10, pp. 2285–2291, 2012.

[48] P. M. Seidel and G. Even, "Delay-Optimized Implementation of IEEE Floating-Point Addition," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 97–113, 2004.

[49] D. R. Lutz, A. Saini, M. Kroes, T. Elmer, and H. Valsaraju, "Fused FP8 4-Way Dot Product With Scaling and FP32 Accumulation," in *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*, 2024, pp. 40–47.

[50] H. H. Saleh and E. E. Swartzlander, "A floating-point fused Dot-product unit," in *26th IEEE International Conference on Computer Design, ICCD*, 2008, pp. 427–431.

[51] A. Roldao Lopes and G. A. Constantinides, "A fused hybrid floating-point and fixed-point dot-product for FPGAs," in *Reconfigurable Computing: Architectures, Tools and Applications*, vol. 5992 LNCS. Springer Berlin Heidelberg, 2010, pp. 157–168.

[52] J. Sohn and E. E. Swartzlander, "A Fused Floating-Point Four-Term Dot Product Unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 3, 2016.

[53] T. Yao, D. Gao, X. Fan, and X. Ren, "Three-operand floating-point adder," in *IEEE 12th International Conference on Computer and Information Technology*, 2012, pp. 192–196.

[54] 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, pp. 624–631.

[55] 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, pp. 1051–1060, 1998.

[56] A. P. Hurst, "Automatic synthesis of clock gating logic with controlled netlist perturbation," in *Proceedings of the 45th Annual Design Automation Conference*. Anaheim, CA: Association for Computing Machinery, 2008, pp. 654–657.

[57] M. Donno, A. Ivaldi, L. Benini, and E. Macii, "Clock-tree power optimization based on RTL clock-gating," in *Proceedings of the 40th Annual Design Automation Conference*. Anaheim, CA: Association for Computing Machinery, 2003, pp. 622–627.

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

[59] 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

[60] Siemens Digital Industries Software, “Automatic sequential clock gating with PowerPro,” 2021. [Online]. Available: https://resources.sw.siemens.com/en-US/white-paper-overview-of-automatic-sequential-clock-gating/

[61] T. T. Hoang and P. Larsson-Edefors, “Data-width-driven power gating of integer arithmetic circuits,” in *Proceedings of the 2012 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2012, pp. 237–242.

[62] 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, pp. 213–229, 2006.

[63] H. Y. Liu and L. P. Carloni, “On Learning-based methods for design-space exploration with high-level synthesis,” in *Proceedings of the 50th Annual Design Automation Conference*. Austin: Association for Computing Machinery, 2013.

[64] L. Ferretti, J. Kwon, G. Ansaloni, G. D. Guglielmo, L. P. Carloni, and L. Pozzi, “Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3736–3747, 2020.

[65] A. Canis, S. D. Brown, and J. H. Anderson, “Modulo SDC scheduling with recurrence minimization in high-level synthesis,” in *24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[66] Z. Zhang and B. Liu, “SDC-based modulo scheduling for pipeline synthesis,” in *IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, 2013, pp. 211–218.

[67] Y. Hara-Azumi, T. Matsuba, H. Tomiyama, S. Honda, and H. Takada, "Selective resource sharing with RT-level retiming for clock enhancement in high-level synthesis," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems.* IEEE, 2012, pp. 1534–1540.

[68] P. N. Krishnapriya and B. Bala Tripura Sundari, "High level synthesis for retiming stochastic VLSI signal processing architectures," in *8th International Conference on Advances in Computing & Communications (ICACC-2018)*, vol. 143, 2018, pp. 10–19.

[69] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo, "Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications," in *58th ACM/IEEE Design Automation Conference (DAC)*, vol. 2021-December. San Francisco, CA: IEEE, 2021, pp. 1327–1330.

[70] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Automated Accelerator Optimization Aided by Graph Neural Networks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference.* San Francisco, California: Association for Computing Machinery, 2022, pp. 55–60.

[71] Y. Liao, T. Adegbija, and R. Lysecky, "Efficient System-Level Design Space Exploration for High-level Synthesis using Pareto-Optimal Subspace Pruning," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference.* Tokyo: Association for Computing Machinery, 2023, pp. 567–572.

[72] 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.

[73] J. Cheng, S. Coward, L. Chelini, R. Barbalho, and T. Drane, "SEER: Super-Optimization Explorer for HLS using E-graph Rewriting with MLIR," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Op-*

erating Systems. La Jolla, CA: Association for Computing Machinery, 8 2024, pp. 1029–1044.

[74] 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.

[75] J. S. Moore, T. W. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5K86™ floating-Point division program," *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 913–926, 1998.

[76] D. M. Russinoff, " A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD-K7 ™ Processor ," *LMS Journal of Computation and Mathematics*, vol. 1, pp. 148–200, 1998.

[77] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, vol. 1690. Springer-Verlag, 1999, pp. 113–130.

[78] ——, *Handbook of practical logic and automated reasoning*, 1st ed. Cambridge University Press, 2009, no. 06.

[79] C. J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, 1995.

[80] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[81] M. Temel and W. A. Hunt, "Sound and Automated Verification of Real-World RTL Multipliers," in *Proceedings of the 21st Formal Methods in Computer-Aided Design, FMCAD*, 2021, pp. 53–62.

[82] M. Temel, A. Slobodova, and W. A. Hunt, "Automated and Scalable Verification of Integer Multipliers," in *Computer Aided Verification*, vol. 12224 LNCS. Springer, 2020, pp. 485–507.

[83] M. Kaufmann and J. S. Moore, "ACL2: An industrial strength version of Nqthm," in *Proceedings of 11th Annual Conference on Computer Assurance*, 1996, pp. 23–34.

[84] T. Drane and H. Jain, "Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks," in *Synopsys User Group (SNUG) Conference*, 2011.

[85] B. Xue, P. Chatterjee, and S. K. Shukla, "Simplification of C-RTL equivalent checking for fused multiply add unit using intermediate models," in *18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013, pp. 723–728.

[86] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to RTL equivalence checking," in *Proceedings of the Conference on Design, Automation and Test in Europe.* Nice: European Design and Automation Association, 2009, pp. 196–201.

[87] YosysHQ GmbH, "SymbiYosys." [Online]. Available: https://symbiyosys.readthedocs.io/en/latest/index.html

[88] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *46th ACM/IEEE Design Automation Conference*, 2009, pp. 460–465.

[89] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A Versatile and Industrial-Strength SMT Solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 415–442, 2022.

[90] L. De Moura and N. Bjørner, "Z3: An efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963 LNCS. Budapest: Springer-Verlag, 2008, pp. 337–340.

[91] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, CGO.* IEEE, 2004, pp. 75–86.

[92] K. D. Cooper and L. Torczon, *Engineering a compiler*, 2nd ed. Elsevier, 2011.

[93] P. Cousot, *Principles of Abstract Interpretation*, 1st ed. MIT Press, 9 2021.

[94] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, vol. Part F130756. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252.

[95] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, 2010.

[96] D. Lesbre and M. Lemerre, "Compiling with Abstract Interpretation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, p. 26, 6 2024.

[97] P. Talbot, D. Cachera, E. Monfroy, and C. Truchet, "Combining constraint languages via abstract interpretation," in *IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, vol. 2019-November. IEEE, 2019, pp. 50–58.

[98] P. Ferrara, L. Negrini, V. Arceri, and A. Cortesi, "Static analysis for dummies: Experiencing LiSA," in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis.* Association for Computing Machinery, 2021, pp. 1–6.

[99] G. Parthasarathy, M. K. Iyer, K. T. Cheng, and F. Brewer, "RTL SAT simplification by Boolean and interval arithmetic reasoning," in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, vol. 2005, 2005, pp. 297–302.

[100] J. Xu, E. Murphy, J. Cortadella, and L. Josipovic, "Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* Monterey, CA: Association for Computing Machinery, 2023, pp. 27–37.

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

[102] 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).* Association for Computing Machinery, 2002, pp. 304–314.

[103] L. De Moura and N. Bjørner, "Efficient E-matching for SMT solvers," in *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, vol. 4603 LNAI. Bremen: Springer-Verlag, 2007, pp. 183–198.

[104] E. Singher and S. Itzhaky, "Easter Egg: Equality Reasoning Based on E-Graphs with Multiple Assumptions," in *Formal Methods in Computer-Aided Design*, 2024, p. 70.

[105] T. Hou, S. Laddad, and J. M. Hellerstein, "Towards Relational Contextual Equality Saturation," 2024. [Online]. Available: https://tylerhou.com/contextual-eqsat.pdf

[106] B. Y. E. Chang and K. R. M. Leino, "Abstract interpretation with alien expressions and heap structures," in *Verification, Model Checking, and Abstract Interpretation*, vol. 3385. Springer, 2005, pp. 147–163.

[107] O. Flatt, S. Coward, M. Willsey, Z. Tatlock, and P. Panchekha, "Small Proofs from Congruence Closure," in *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design, FMCAD 2022.* TU Wien Academic Press, 2022, p. 9.

[108] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 50, no. 6. Association for Computing Machinery, 2015, pp. 1–11.

[109] 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.* Association for Computing Machinery, 2021, pp. 21–31.

[110] A. Vanhattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, “Vectorization for digital signal processors via equality saturation,” in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. Association for Computing Machinery, 2021, pp. 874–886.

[111] 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, pp. 1919–1932, 2020.

[112] H. Xu and F. Kjolstad, “Copy-and-patch compilation: A fast compilation algorithm for high-level languages and bytecode,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, 2021.

[113] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, “Rewrite rule inference using equality saturation,” in *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA. Association for Computing Machinery, 2021.

[114] A. Pal, B. Saiki, R. Tjoa, C. Richey, A. Zhu, O. Flatt, M. Willsey, Z. Tatlock, and C. Nandi, “Equality Saturation Theory Exploration à la Carte,” *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA, pp. 1034–1062, 2023.

[115] C. Chen, G. Hu, D. Zuo, C. Yu, Y. Ma, and H. Zhang, “E-Syn: E-Graph Rewriting with Technology-Aware Cost Functions for Logic Synthesis,” in *61st ACM/IEEE Design Automation Conference*. San Francisco, CA: Association for Computing Machinery, 3 2024, pp. 1–6.

[116] A. T. Sheikh, A. Shoker, S. A. Fahmy, and P. Esteves-Verissimo, “ResiLogic: Leveraging Composability and Diversity to Design Fault and Intrusion Resilient Chips,” 9 2024. [Online]. Available: https://arxiv.org/abs/2409.02553

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

[118] G. H. Smith, Z. D. Sisco, T. Techaumnuaiwit, J. Xia, V. Canumalla, A. Cheung, Z. Tatlock, C. Nandi, and J. Balkind, "There and Back Again: A Netlist's Tale with Much Egraphin'," 3 2024. [Online]. Available: https://arxiv.org/abs/2404.00786

[119] 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.

[120] S. Coward, T. Drane, and G. A. Constantinides, "ROVER: RTL Optimization via Verified E-Graph Rewriting," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, pp. 4687–4700, 2024.

[121] G. Steele, *Common LISP: the language*, 2nd ed. Elsevier, 1990.

[122] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. Barrett, and C. Tinelli, "Towards bit-width-independent proofs in SMT solvers," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11716 LNAI, 2019.

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

[124] A. Solar-Lezama, "Program Synthesis by Sketching," Ph.D. dissertation, University of California at Berkeley, 2009.

[125] C. M. Bishop, *Pattern recognition and machine learning*, 1st ed. Berlin: Springer-Verlag, 2006, no. 4.

[126] R. Lougee-Heimer, "The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community," *IBM Journal of Research and Development*, vol. 47, no. 1, pp. 57–66, 2003.

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

[128] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," 2 2020.

[129] 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.

[130] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," in *Numerical Software Verification*, vol. 10152 LNCS. Springer, 2017, pp. 63–77.

[131] S. Coward, G. A. Constantinides, and T. Drane, "Combining E-Graphs with Abstract Interpretation," in *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis.* Orlando, FL: Association for Computing Machinery, 2023, pp. 1–7.

[132] P. Cousot, R. Cousot, and L. Mauborgne, "The Reduced Product of Abstract Domains and the Combination of Decision Procedures," in *Foundations of Software Science and Computational Structures*, vol. 6604 LNCS. Springer, 2011, pp. 456–472.

[133] G. Birkhoff, "Von Neumann and Lattice Theory," *Bulletin of the American Mathematical Society*, vol. 64, no. 3, 1958.

[134] P. Cousot and R. Cousot, "Comparing the galois connection and widening/narrowing approaches to abstract interpretation," in *Programming Language Implementation and Logic Programming*, vol. 631 LNCS. Springer, 1992, pp. 269–295.

[135] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis.* Society for Industrial and Applied Mathematics, 2009.

[136] U. Kulisch, *Computer Arithmetic in Theory and Practice.* Academic press, 1981.

[137] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic Taylor ex-

pansions," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 1, 2018.

[138] S. Coward, G. Constantinides, and T. Drane, "Automating Constraint-Aware Datapath Optimization using E-Graphs," in *60th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, 2023, pp. 1–6.

[139] S. Coward, T. Drane, and G. A. Constantinides, "Constraint-Aware E-Graph Rewriting for Hardware Performance Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–14, 2024.

[140] S. Coward, T. Drane, E. Morini, and G. A. Constantinides, "Combining Power and Arithmetic Optimization via Datapath Rewriting," in *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*. IEEE, 6 2024, pp. 24–31.

[141] 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, pp. 121–216, 5 2012.

[142] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016. [Online]. Available: https://smt-lib.org/

[143] M. Stepp, R. Tate, and S. Lerner, "Equality-based translation validator for LLVM," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, vol. 6806 LNCS. Snowbird, UT: Springer-Verlag, 2011, pp. 737–742.

[144] S. Coward, E. Morini, B. Tan, T. Drane, and G. Constantinides, "Datapath Verification via Word-Level E-Graph Rewriting," in *23rd Conference on Formal Methods in Computer-Aided Design – FMCAD*. TU Wien Academic Press, 10 2023, pp. 92–100.

[145] J. Woodruff, T. Koehler, A. Brauckmann, C. Cummins, S. Ainsworth, and M. F. P. O'Boyle, "Rewriting History: Repurposing Domain-Specific CGRAs," 9 2023. [Online]. Available: https://arxiv.org/abs/2309.09112

[146] S. Thomas and J. Bornholt, “Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 19–34. [Online]. Available: https://doi.org/10.1145/3617232.3624873

[147] A. Wanna, S. Coward, T. Drane, G. A. Constantinides, and M. D. Ercegovac, “Multiplier Optimization via E-Graph Rewriting,” in *57th Asilomar Conference on Signals, Systems, and Computers*, 12 2023, pp. 1528–1533.

[148] Y. Zhang, Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, Z. Tatlock, and M. Willsey, “Better Together: Unifying Datalog and Equality Saturation,” *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 468–492, 2023.

[149] T. Kœhler, A. Goens, S. Bhat, T. Grosser, P. Trinder, and M. Steuwer, “Guided Equality Saturation,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 1727–1758, 1 2024.

[150] G. Sun, Y. Zhang, and H. Ni, “E-Graphs as Circuits, and Optimal Extraction via Treewidth,” 2024. [Online]. Available: https://arxiv.org/abs/2408.17042

[151] A. K. Goharshady, C. K. Lam, and L. Parreaux, “Fast and Optimal Extraction for Sparse Equality Graphs,” in *Proceedings of the ACM on Programming Languages*. Association for Computing Machinery, 10 2024, pp. 2551–2577.

[152] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump, “6 Years of SMT-COMP,” *Journal of Automated Reasoning*, vol. 50, no. 3, pp. 243–277, 2013.

[153] A. Mishenko, “IWLS Logic Synthesis Competition,” 2024. [Online]. Available: https://github.com/alanminko/iwls2024-ls-contest

[154] A. Mishenko, R. Brayton, and M. Fujita, “Mapping and Retiming Revisited,” in *International Workshop on Logic Synthesis*, 2023. [Online]. Available: https://people.eecs.berkeley.edu/~alanmi/publications/2023/iwls23_m&r.pdf

[155] Microsoft, “DirectX-Specs,” Microsoft, Tech. Rep., 2022. [Online]. Available: https://microsoft.github.io/DirectX-Specs/

[156] M. Rezaalipour, M. Biasion, I. Scarabottolo, G. A. Constantinides, and L. Pozzi, “A Parametrizable Template for Approximate Logic Synthesis,” in *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2023, pp. 175–178.

[157] T. Drane, S. Coward, M. Temel, and J. Leslie-Hurd, “On the Systematic Creation of Faithfully Rounded Commutative Truncated Booth Multipliers,” in *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*. IEEE, 6 2024, pp. 108–115.

[158] H. Chen, N. Zhang, S. Xiang, Z. Zeng, M. Dai, and Z. Zhang, “Allo: A Programming Model for Composable Accelerator Design,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 593–620, 6 2024.

[159] B. Pelton, A. Sapek, K. Eguro, D. Lo, A. Forin, M. Humphrey, J. Xi, D. Cox, R. Karandikar, J. de Fine Licht, E. Babin, A. Caulfield, and D. Burger, “Wavefront Threading Enables Effective High-Level Synthesis,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1066–1090, 6 2024.

[160] R. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” in *Computer Aided Verification*, vol. 6174 LNCS. Springer Berlin Heidelberg, 2010, pp. 24–40.

[161] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage, and G. D. Micheli, “The EPFL Logic Synthesis Libraries,” 2018. [Online]. Available: https://arxiv.org/abs/1805.05121

[162] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang, Z. Zhang, and J. Cong, “TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 4, pp. 1–31, 12 2023.

[163] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “LLHD: a multi-level intermediate representation for hardware description languages,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY, USA: ACM, 6 2020, pp. 258–271.

[164] A. Guerrieri, S. Guha, L. Josipovic, and P. Ienne, “DynaRapid: From C to FPGA in a Few Seconds,” in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* New York, NY, USA: ACM, 4 2024, p. 40.

Appendices

Appendix A

Benchmarks

Source	Benchmark	Ch. 3 Area	Ch. 5 Delay	Ch. 6 Power	Ch. 7 Verif
Intel	Media Kernel	✓	*✓		
	Weight Calculation	✓		✓	
	Unorm8 to FP32	5	✓		
	FP16 to Unorm11	5	✓		
	Normalization	5	✓		
	Comb. Mux Add Tree	6		✓	
	Address Generation	6		✓	
Open	FIR Filter Kernel [24]	✓			✓
	ADPCM Decoder [24]	✓			✓
	Shift FMA (pg. 57)	✓			
	Shift Mult (pg. 137)	✓			✓
	MCM (pg. 58)	✓			
	FP16 Sub (pg. 108)	5	✓	*✓	
	FP8 Sub (pg. 108)	5	✓		
	Max Tree (pg. 111)	5	✓		
	Pipe Mux Add Tree [54]	6		✓	
	Dual-Op ALU [55]	6		✓	
	Sequential Reg [60]	6		✓	
	H-264 VBSME [24]				✓
	Box Filter (pg. 152)				✓

Table A.1: All Intel closed-source and open-source RTL benchmarks used in the thesis. Each benchmark consists of a single RTL module. A tick indicates that the benchmark was used in that chapter. A number indicates that the evaluation of the approach for that benchmark was presented in a different chapter. For example, the results of applying techniques from Chapter 3 to FP16 Sub are presented in Chapter 5. Entries marked with a star indicate the benchmark was modified for use in that chapter. Chapter 4 did not use any RTL benchmarks.

Appendix B

Associativity of Addition Condition

The fully general associativity of addition rewrite contains no type restrictions, namely all possible combinations of signage and bitwidth type parameters can be matched. For this rewrite definition, a condition is synthesized that is a function of the bitwidth and signage parameters using the approach described in Section 3.2.2. It can be specified as a triple, (ϕ, lhs, rhs), such that (3.9) holds, where,

$$lhs = (+\ w_e\ w_d\ s_d\ (+\ w_d\ s_d\ w_a\ s_a\ \mathbf{a}\ w_b\ s_b\ \mathbf{b})\ w_c\ s_c\ \mathbf{c})$$

$$rhs = (+\ w_e\ w_a\ s_a\ \mathbf{a}\ w_d\ s_d\ (+\ w_d\ s_d\ w_b\ s_b\ \mathbf{b}\ w_c\ s_c\ \mathbf{c}))$$

$$
\begin{aligned}
\phi = \quad & (!(w_e < w_d + 1) \wedge ((!(w_d < w_a + w_b)\wedge \\
& ((!(w_d < w_b + w_c) \wedge ((!(s_c \wedge s_d) \wedge ((!(s_a \wedge s_b)\wedge \\
& ((!(s_a \wedge s_d) \wedge ((!(s_b \wedge s_c))))))))\vee \\
& (s_c \wedge s_d \wedge (((s_a \wedge s_b))))))))))\vee \\
& (w_e < w_d + 1 \wedge ((!(w_e < w_b + 1) \wedge ((!(s_b)\wedge \\
& ((!(s_c \wedge s_d) \wedge ((!(s_a \wedge s_d))\vee \\
& (s_a \wedge s_d \wedge ((!(w_a < w_e))))))\vee \\
& (s_c \wedge s_d \wedge ((!(w_c < w_e - 1)\wedge \\
& ((!(w_e < w_a + 1) \wedge ((!(s_a \wedge s_c))))\vee \\
& ((w_e < w_a + 1))))))))\vee \\
& (s_b \wedge ((!(s_a \wedge s_c)\wedge \\
& ((!(s_a \wedge s_b) \wedge ((!(s_b \wedge s_c))))))\vee \\
& (s_a \wedge s_c \wedge ((!(s_b \wedge s_d)\wedge \\
& ((w_e < w_c + 1 \wedge ((!(w_a < w_e))))))\vee \\
& ((s_b \wedge s_d))))))))\vee \\
& (w_e < w_b + 1 \wedge ((!(w_a < w_e) \wedge ((!(w_c < w_e))\vee \\
& (w_c < w_e \wedge ((!(s_c))\vee \\
& (s_c \wedge ((!(s_b \wedge s_d)\wedge \\
& ((!(s_b \wedge s_c) \wedge ((!(s_c \wedge s_d))))))\vee \\
& ((s_b \wedge s_d))))))))\vee \\
& (w_a < w_e \wedge ((!(s_a) \wedge ((!(w_c < w_e - 1))\vee \\
& (w_c < w_e - 1 \wedge ((!(s_c))\vee \\
& (s_c \wedge ((!(s_b \wedge s_d) \wedge ((!(s_d)\wedge \\
& ((!(s_b \wedge s_c))))))\vee \\
& ((s_b \wedge s_d))))))))\vee \\
& (s_a \wedge ((!(s_b \wedge s_d) \wedge ((!(s_a \wedge s_b)\wedge \\
& ((!(s_d)))))) \vee ((s_b \wedge s_d)))))))))).
\end{aligned}
$$

Appendix C

SMT Encoding of Motivational Example

The code below describes a complete encoding of the motivational equivalence checking problem we summarised in Figure 7.1c.

```
(set-logic QF_BV)
; Inputs
(declare-fun A () (_ BitVec 16))
(declare-fun B () (_ BitVec 16))
(declare-fun M () (_ BitVec 4))
(declare-fun N () (_ BitVec 4))
; Outputs
(declare-fun spec () (_ BitVec 63))
(declare-fun impl () (_ BitVec 63))
; Signals
(declare-fun C () (_ BitVec 32))
(declare-fun D () (_ BitVec 31))
(declare-fun E () (_ BitVec 31))
(declare-fun P () (_ BitVec  5))
```

```
; Zeros
(declare-fun zeros_1  () (_ BitVec  1))
(declare-fun zeros_12 () (_ BitVec 12))
(declare-fun zeros_15 () (_ BitVec 15))
(declare-fun zeros_16 () (_ BitVec 16))
(declare-fun zeros_27 () (_ BitVec 27))
(declare-fun zeros_31 () (_ BitVec 31))
(declare-fun zeros_32 () (_ BitVec 32))
(declare-fun zeros_58 () (_ BitVec 58))
(assert (= zeros_1  #b0))
(assert (= zeros_12 #b000000000000))
(assert (= zeros_15 #b000000000000000))
(assert (= zeros_16 #b0000000000000000))
(assert (= zeros_27 #b000000000000000000000000000))
(assert (= zeros_31 #b0000000000000000000000000000000))
(assert (= zeros_32 #b00000000000000000000000000000000))
(assert (= zeros_58 (concat zeros_27 zeros_31)))

(assert (= D (bvshl (concat zeros_15 A) (concat zeros_27 M))))
(assert (= E (bvshl (concat zeros_15 B) (concat zeros_27 N))))
(assert (= spec (bvmul (concat zeros_32 D) (concat zeros_32 E))))

(assert (= C (bvmul (concat zeros_16 A) (concat zeros_16 B))))
(assert (= P (bvadd (concat zeros_1 M) (concat zeros_1 N))))
(assert (= impl (bvshl (concat zeros_31 C) (concat zeros_58 P))))

(assert (not (= spec impl)))
(check-sat)
```