Datapath Verification via Word-Level E-Graph Rewriting

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

***Abstract*—Formal verification of datapath circuits is challenging as they are subject to intense optimization effort in the design phase. Industrial vendors and design companies deploy equivalence checking against a golden or existing reference design to satisfy correctness concerns. State-of-the-art datapath equivalence checking tools deploy a suite of techniques, including rewriting. We propose a rewriting framework deploying bitwidth-dependent rewrites based on the e-graph data structure, providing a powerful assistant to existing tools. The e-graph allows generation of a path of rewrites between the reference and implementation designs, which can then be checked by a trusted industry tool. We demonstrate that the intermediate proofs generated by the assistant enable convergence in a state-of-the-art tool, without which the industrial tool runs for 24 hours without making progress. The intermediate proofs automatically introduced by the assistant also reduce the total proof runtime by up to 6×.**

***Index Terms*—Formal Verification, Datapath, E-Graph, Equivalence Checking.**

I. Introduction

Arithmetic datapath circuits like adders and multipliers are included in almost every electronic device. Designers of these circuits implement low-level optimizations targeting the best power, performance and area. As a result, the verification of datapath circuits is challenging since the code can be difficult to review, making it hard to identify a sufficient test suite. Typically, exhaustive simulation is infeasible due to the size of the input space. Undetected bugs lead to system failures and reputational damage [1]. Formal Verification (FV) is the only scalable option to prove the absence of bugs in hardware [2].

One of the most successful FV approaches to verify datapath circuit designs is based on Equivalence Checking (EC), where the design under test, usually called the *implementation*, is proven to be equivalent to a golden reference design, often called the *specification*. Electronic Design Automation (EDA) vendors have developed commercial tools drastically lowering the entry barrier [3], allowing semiconductor companies to fully verify many different designs [4], [5].

Commercial tools orchestrate a suite of solver technologies [3], including SAT, SMT and BDD based solvers. Yet still some simple designs can not be proven equivalent. For example, an industrial state-of-the-art tool is unable to prove the equivalence of the two designs shown in Figure 1 without requiring manual effort to apply advanced formal techniques. We enhance the capabilities of such tools by deploying word-level rewriting in combination with a data structure, known as an e(quivalence)-graph. E-graphs are found at the heart of modern SMT solvers [6], but by applying them at the abstraction level used by humans in RTL design we can tailor the rewrites to datapath verification.

```
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.

Fig. 1: A motivational example, where existing EC tools fail to prove the equivalence of these two designs.

In this work we modify an existing e-graph-based RTL optimization tool [7] to produce a powerful formal verification assistant. 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 e-graph based LLVM optimizer [8], and later modified it to perform translation validation [9]. We differ from this previous work in that we validate numerically intense optimizations at a lower abstraction level often performed by a human rather than a compiler. We also deploy modern e-graph developments allowing us to incorporate value range analysis techniques and can generate a simplified EC problem for FV engineers. The approach presented is sound, as we check each intermediate step using a trusted EC tool. The paper 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,
- a specialized and extensible bitwidth dependent rewrite set for datapath verification,
- 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.

First, we provide the necessary background on verification and e-graphs. In Section III we describe how word-level e-

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_17

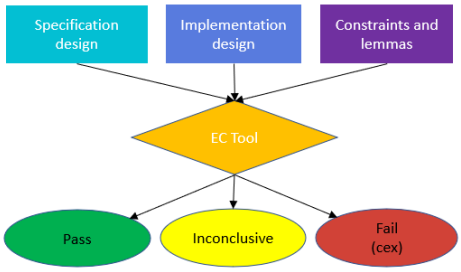


Fig. 2: The inputs of an EC tool are two designs, specification and implementation, a set of constraints to drive the possible values to tests and a set of lemmas to prove. Each lemma can pass, fail or be inconclusive. A counterexample (cex) is provided for each failing lemma.

graphs can be applied to produce a verification assistant. In Section IV we describe a case study where we outperform the industrial state of the art. Finally, in Section V we present results demonstrating overall verification runtime improvements.

II. Background

A. Datapath Verification

Classic formal property verification methods successfully used to verify state machines and communication protocols are not able to verify datapath dominated circuits. Theorem Proving [10]–[14] and Symbolic Trajectory Evaluation [15] are valuable approaches, but their common downsides are a high barrier to entry and maintenance of complex code bases.

An alternative and successful approach is to rely on 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 [4], [16]. The most popular types of EC are Boolean, Sequential and Transactional, and in this paper we focus on Transactional EC of combinational circuits, 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. See Figure 2.

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. This is usually described as *C2RTL* EC. Another common option is what is called *RTL2RTL* EC, where the reference is a trusted version of the same design in RTL, usually a version from previous projects or based on a third party library like Synopsys' DesignWare [17].

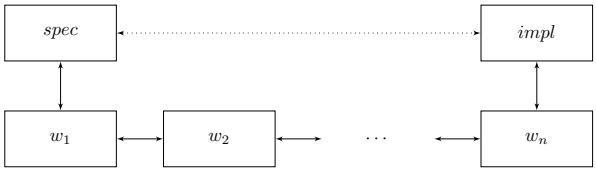


Fig. 3: Overview of the waterfall approach used by FV engineers. The dashed line between $spec$ and $impl$ represents an inconclusive verification. Full equivalence is achieved introducing n intermediate designs w_i and proving the equivalences of all the pairs $(spec, w_1)$, (w_1, w_2), ..., $(w_n, impl)$.

Inconclusive results are commonplace in real-life EC 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 achieved by introducing intermediate designs, as shown in Figure 3. If all the intermediate equivalence steps are proven, the equivalence between specification and implementation holds.

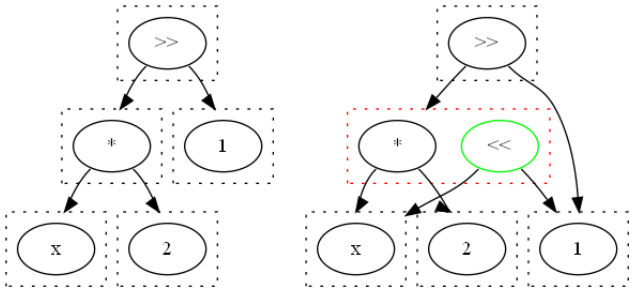
One of the key motivations for this work derives from an overview of the technology behind Synopsys' industry leading Datapath Validation (DPV) tool [3]. The tool orchestrates a suite of techniques and solvers to prove the equivalence of input designs. One of these techniques is a set of rewrite engines. In [3], the authors state that certain rewrite sets "are only applied selectively" or their application "can be counterproductive". As a result these rewrite engines are heuristic and may not explore the required space. The techniques presented in Section III describe a rewrite orchestration approach that does not suffer from these limitations.

One relevant work combined rewriting and theorem proving to verify the correctness of gate-level multiplier designs in RTL [18], [19]. In this work, the authors deploy ACL2 verified [20] rewrites to transform optimized implementations into normalized implementations. Whilst our work targets a higher abstraction level, techniques and principles applied in the multiplier verification work will be relevant here.

B. E-Graphs

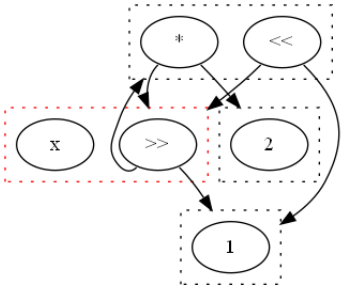
E-graphs cluster equivalent expressions into e(quivalence)-classes, enabling a compact representation of alternative but functionally identical implementations. In the e-graph, nodes represent variables, constants or operators that point to children e-classes. This captures the intuition that we may choose how to implement a given sub-expression at any point in the design. Due to these nested choices, an e-graph can represent exponentially many implementations in the number of nodes.

An e-graph is grown via constructive application of local equivalence preserving rewrites, $l \rightarrow r$, where the right-hand side of the rewrite is added to the e-class containing l, without removing l as would be done in a traditional rewrite engine. As a result, the e-graph avoids the phase-ordering problem, where the order of application impacts the results. This approach to growing an e-graph is known as equality saturation [8], [21],



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

(b) Apply $x \times 2 \rightarrow x \ll 1$



(c) Apply $(x \ll s) \gg s \rightarrow x$

Fig. 4: Simple e-graph rewriting over the integers. The dashed boxes represent e-classes of expressions, where we have highlighted the modified e-class in red at each stage. Green nodes represent newly added nodes.

[22]. A simple e-graph rewriting example is shown in Figure 4, where the dashed boxes represent e-class boundaries and the arrows connect nodes to their child e-classes.

The e-graph data structure has been used in the formal methods community for many years [23] and can be found in modern SMT solvers such as Z3 [6]. One particularly relevant work used e-graphs to perform translation validation of an LLVM compiler [9]. The recently released `egg` library [21] has enabled researchers to quickly develop a wide range of e-graph applications, ranging from hardware design [7] to rewrite rule synthesis [24]. By using e-graphs to represent datapath circuit designs in RTL we can take advantage of the e-graphs' ability to explore equivalent designs efficiently.

III. Proving Equivalence via E-Graph Rewriting

The problem we tackle is, given specification and implementation RTL designs, prove them equivalent or reduce the original EC problem to a simpler one to solve. Given this objective, we will now describe how e-graph rewriting can provide an efficient solution. Figure 5 illustrates the overall flow of the assistant. In this work we used a particular commercial tool throughout, but any RTL2RTL EC tool could be substituted in its place.

A. E-Graph Initialization

Using the framework developed in [7], we produce an e-graph representation of RTL, encoding all signal bitwidth and signage definitions. We use an intermediate language made up of nested S-expressions, as in Common Lisp [25]:

```
term::=(operator [term] [term]...[term])
```

For example, the following System Verilog:

```
logic [7:0] a, b;
logic [8:0] c;
always_comb c = a[7:0] + b[7:0];
```

corresponding to an unsigned addition of two primary 8-bit inputs `a` and `b`, stored in a 9-bit result is expressed as:

```
(+ 9 unsigned 8 unsigned a 8 unsigned b).
```

The + operator takes eight arguments, describing the output and operand signals.

This intermediate language is sufficient to correctly represent the functional behaviour of combinational RTL. Verilog operator definitions are context dependent meaning knowledge of all bitwidth and signage definitions is essential [26]. In this work we target word-level RTL written in System Verilog. Using the open-source Slang parser [27], we implemented an automated flow converting System Verilog into this intermediate language. We parse both RTL designs and generate expressions, S and I, in the intermediate language, for the specification and implementation respectively.

In most e-graph applications built using `egg`, the e-graph is initialized with a single expression representing the design to be optimized. However, in our work we initialize the e-graph with both S and I, such that the e-graph has two roots. The nodes common to both designs are automatically shared by `egg`. In this paper we describe RTL generating a single output, but using constructs from [7] it is trivial to generalize to multiple outputs from each design.

In Figure 6, we represent the two designs shown in Figure 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. In the following sections we will discuss how as the e-graph is grown, common intermediate signals can be discovered. Initialising the e-graph with both designs means that we can simultaneously rewrite both designs in order to find a common equivalent.

B. Bitwidth Dependent Rewriting

The rewrites define the space of equivalent designs that can be reached as the e-graph grows. We build upon a subset of the bitwidth dependent rewrites described in [7], which was originally designed for optimization and was learnt from industrial RTL engineers. The optimization rewrite set deployed specific rewrites to improve correlation with the downstream logic synthesis tool. These rewrites are not deployed in the verification rewrite set. It is natural that the verification rewrite set should include many of the optimization capabilities but 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.

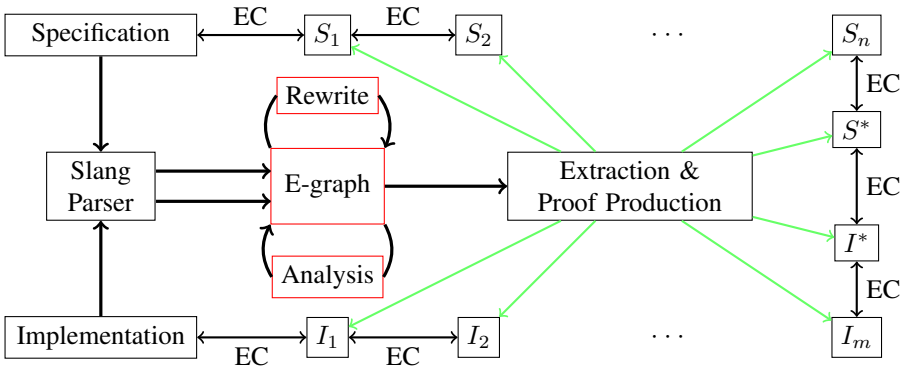


Fig. 5: Flow diagram for the verification assistant, taking a specification and implementation circuit design in System Verilog. The designs are parsed and an e-graph is constructed. From the rewritten e-graph, extract two designs S^* and I^* along with intermediate designs forming a verification waterfall.

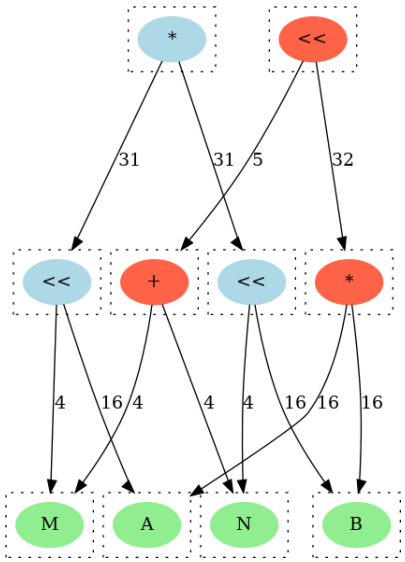


Fig. 6: Initial e-graph representing two designs shown in Figure 1, a specification (blue) and implementation (red). Shared nodes are colored green. Edge labels denote bitwidths. All e-classes (dashed boxes) initially contain a single node.

Table I 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. We selected rewrites that were 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. To achieve this we parameterize the pattern matching left-hand side and apply the rewrites conditionally. We construct necessary and sufficient conditions that are functions of the rewrite parameters. Letting $l(\cdot)$ and $r(\cdot)$ denote functions mapping a vector of parameters $\vec{p}$, encoding operand bitwidth and signage, to expressions in the intermediate language. Given a parameterized rewrite, $l(\vec{p}) \rightarrow r(\vec{p})$, we construct a condition, ϕ, such that $l(\vec{p}) \cong r(\vec{p}) \iff \phi(\vec{p})$. The sufficiency ensures that only valid, equivalence preserving, rewrites are applied. The necessity guarantees that no rewriting opportunities are missed for this rewrite. Missed opportunities can be the difference between a proven equivalence check and an inconclusive result. We will see this in Section IV.

TABLE I: 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	$a \times 2$	$a + a$

A challenge for RTL level verification is that functional behaviour 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. We use the interval analysis and bitwidth reduction rewrites described in [28], deploying `egg`'s built-in e-class analysis feature. 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 [3], but program analysis on e-graphs is able to provide more precise abstractions [29].

Having defined a set of rewrites, we use equality saturation to apply them to the e-graph initialized as described in Section III-A. Rewrites are applied to both the specification and implementation designs simultaneously with the objective being to discover equivalent sub-expressions across the two designs. As rewrites are applied, new nodes are added to the e-graph and the e-classes grow, as we see in Figure 8. We vary the number of e-graph rewriting iterations to control the e-graph growth

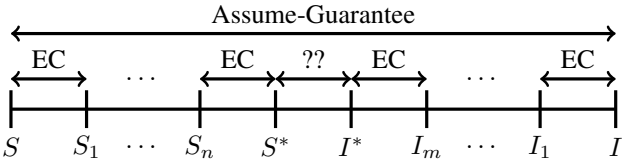


Fig. 7: E-graph extracted waterfall, generated automatically by the assistant. We use EC tools to prove the equivalence of the intermediate steps. The central equivalence check between S^* and I^*, which may not be true, may not be provable using the EC tool, but represents a simplified problem.

throughout this work. Constructive rewrite application adds the overhead of maintaining many equivalent representations of the two designs in the e-graph, but greatly simplifies the problem of determining a correct rewrite application order.

C. Extraction

Once the e-graph has saturated or reached a timeout, the e-graph represents two sets of equivalent designs, one for the specification and one for the implementation. From the e-graph we now seek to extract 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 identical. If they are found in different e-classes, we extract distinct S^* and I^* sharing as many of the common sub-expressions as is feasible from the e-graph. Figure 5 shows the flow.

To extract S^* and I^* we first identify which e-classes in the e-graph are associated with each design. Let C denote the set of all e-classes. Given a root e-class, r, we recursively construct an associated $C_r \subseteq C$. Starting from $C_r = \emptyset$, we iterate through each node in r, adding its children e-classes to C_r. We continue, 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.

Letting root(e), be a function returning the root e-class for a given expression e in the intermediate language, we use the algorithm described above to construct $C_{\text{spec}} \subseteq C$ starting from root(S) and $C_{\text{impl}} \subseteq C$ starting from root(I). We construct the shared e-class set, $C_{\text{shared}} = C_{\text{spec}} \cap C_{\text{impl}}$, which is used to identify the S^* and I^* that share the most common nodes. We update the spec and impl sets, $C'_{\text{spec}} = C_{\text{spec}} \setminus C_{\text{shared}}$ and $C'_{\text{impl}} = C_{\text{impl}} \setminus C_{\text{shared}}$. In Figures 6 and 8, we highlight C'_{spec} in blue, C'_{impl} in red and C_{shared} in green.

In previous work we have deployed hardware specific cost functions, for example circuit area or delay, that we seek to minimize in the extraction phase [7], [28]. In this instance, we use a simpler objective function of e-graph nodes n:

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

where class(n) returns the e-class containing the node n and K is the total number of e-classes in the e-graph. We maximize this objective function to ensure that we share the maximum number of nodes possible, where the negative scoring of unshared nodes ensures that amongst designs sharing the same number of nodes, we extract the simplest one. We formulate the problem as an integer linear programming problem (ILP) [30]. We define N to be the set of nodes and $E \subseteq N \times C$ the set of edges. We also introduce N_c to denote the set of nodes in a given e-class c and P_c to denote the set of parent nodes of c. For each node $n \in N$ we associate an objective, obj(n), and a binary variable $x_n \in \{0, 1\}$, which indicates whether n is implemented in either of the extracted RTL designs. Lastly we introduce $R = \text{root}(I) \cup \text{root}(S)$. With these definitions the problem formulation is the following:

$$\text{maximize} \quad \sum_{n \in N} \text{obj}(n) \cdot x_n \tag{2}$$

$$\text{subject to} \quad \forall (n, c) \in E \quad : \; x_n \leq \sum_{\hat{n} \in N_c} x_{\hat{n}} \tag{3}$$

$$\forall c \in R \quad : \sum_{n \in N_c} x_n = 1 \tag{4}$$

$$\forall c \in C \quad : \sum_{n \in N_c} x_n \leq 1 \tag{5}$$

$$\forall c \in C \text{ s.t. } P_c \neq \emptyset : \sum_{n \in N_c} x_n \leq \sum_{\hat{n} \in P_c} x_{\hat{n}}. \tag{6}$$

In the ILP problem, (3) guarantees that for every node n, we implement a node from each of its child e-classes, extracting only valid designs. (4) then ensures that the outputs from both specification and implementation designs are produced by the extracted design. Lastly, (5) allows at most one node in each e-class to be implemented and (6) ensures that only e-classes with implemented parents are selected, namely there are no unused signals in the generated RTL. We deploy topological sorting variables to handle cycles in the e-graph [7], [30]. We use the Coin-Or CBC solver to solve the ILP problem.

For improved performance, we also use a comparable objective function that computes a greedy extraction based on `egg`'s built-in method. 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. We would recommend the ILP approach for solving EC problems that will require manual intervention.

The extracted solution corresponds to two expression in the intermediate representation, S^*, equivalent to the specification and I^*, equivalent to the implementation, from which the tool automatically generates RTL. Using the recently added proof production feature in `egg` [31], two sequences of intermediate designs 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 remove the need to trust the correctness of the rewrites, the assistant generates System Verilog implementations for each of the intermediate designs and deploys the EC tool to formally verify the equivalence at each step as shown in Figure 7. If the EC tool can prove each step including $S^* \cong I^*$, we have proven the equivalence of S and I. Each intermediate proof is independent and can thus be proven in

parallel. We can also specialize the solver configuration for each intermediate proof, since we are able to map rewrites 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 a user. We encoded a mapping from rewrites to the most efficient solve script in the assistant. With limited effort the assistant can be extended to target additional solvers.

To ensure soundness of the generated waterfall, 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. If the tool is unable to prove $S^* \cong I^*$ then human intervention is required. However the EC problem is simplified, as these designs share more common signals than the original S and I.

IV. Case Study

We present a case study of a real world problem where this technique proves beneficial. In all the following results we 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 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 usually reduced to a standard non-denormal multiplication by shifting 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 $lzc(\cdot)$ is the leading zero counter function.

In three iterations of rewriting the e-graph 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 8. After two iterations of rewriting the first shared signal is detected, see the green left-shift in Figure 8a, where we have highlighted the initial specification and designs sharing the green node with brighter arrows. The e-graph shown in Figure 8b, after three iterations of rewriting, contains only green nodes, since the tool 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 $C_{\text{shared}} = C$.

From the final e-graph, Figure 8b, the tool then extracts identical S^* and I^* along with the sequence of rewrites that were applied to reach it. We summarise the rewrites applied below, omitting bitwidth alteration and commutativity steps.

$$(A \times B) \ll (M + N) \rightarrow \tag{7}$$

Unmerge Left-Shift $$((A \times B) \ll N) \ll M \rightarrow \tag{8}$$

Left-Shift Mult $$(A \times (B \ll N)) \ll M \rightarrow \tag{9}$$

Left-Shift Mult $$(A \ll M) \times (B \ll N) \tag{10}$$

The e-graph assistant runs in 0.14 seconds, growing an e-graph comprised of 77 nodes. The EC tool is unable to prove the "Left-Shift Mult" and "Mult Left-Shift" transformations when non-uniform bitwidths are used. We resolve this by automatically inserting an additional intermediate step with standardized bitwidths. We hypothesize that this is due to a rewrite rule only being applied under certain parameterizations in the EC engine.

Including all commutativity and bitwidth alteration rewrites, the assistant generated a total of 20 intermediate equivalence checks (including the "Assume-Guarantee" lemma) for the EC tool to prove. All intermediate proofs and the final completeness lemma are proven in 0.1 seconds by the EC tool. In contrast, when passed the original EC problem, $S \cong I$ with no assistance, the tool did not return a result within 24 hours.

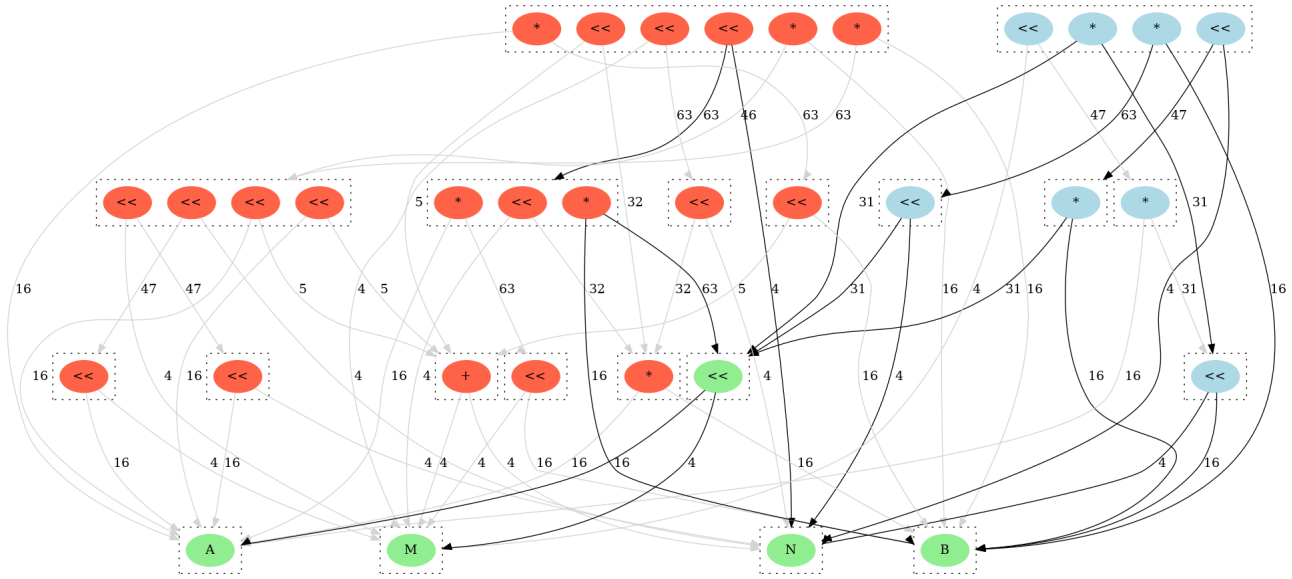
V. Results

Having demonstrated how the verification assistant can provide the intermediate steps transforming a previously inconclusive proof into one solved in under one second, we will now demonstrate how the assistant can improve overall verification runtimes across a datapath optimization benchmark set. We take benchmarks from [32] and implement original and optimized RTL for those designs that are fully described in this paper. The ADPCM decoder is an approximate multiplication implementation. We include 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, $\sum_i |a_i - b_i|$. The FIR Filter is a typical finite impulse response filter of depth eight. The case study and box filter are Intel provided benchmarks. The box filter is a reconfigurable square filter, sampling four pixels at a time. The dataflow graph for this design is shown in Figure 9. The optimized design deploys constant factorization and mux rewriting which is relatively challenging for the EC tool to prove.

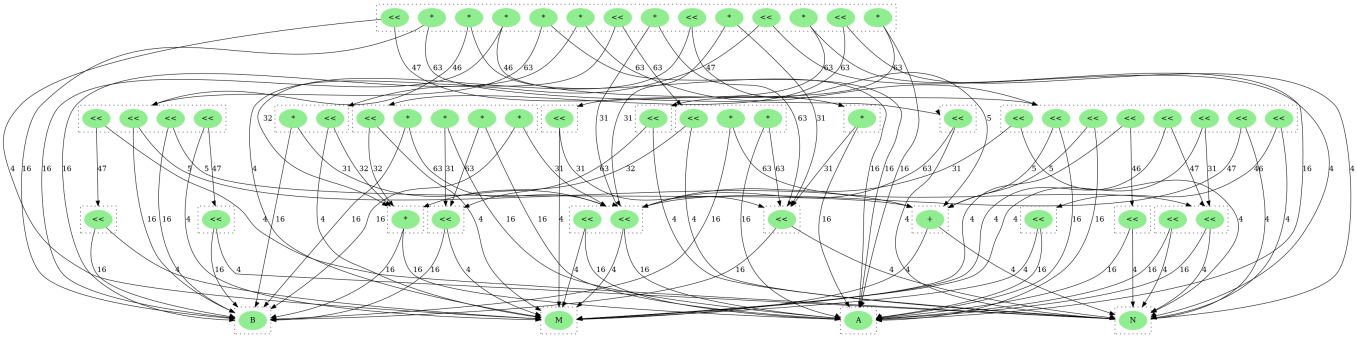
The benchmarks include a range of arithmetic and logical operators, representative of typical RTL optimizations that may be performed by hand or by a specialized datapath optimization tool. For each benchmark, we run the assistant until either, it discovers a complete path between specification and implementation or it deploys five iterations of rewriting, whichever comes first. The e-graph applies all rewrites in parallel at each iteration, meaning that 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. We report the runtime from when the solvers start running. For the baseline, we deploy all the EC tool's solvers in parallel and take the fastest proof. When the verification assistant has generated a sequence of intermediate proofs, we report the maximum time taken to solve a single sub-problem, 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 licence checks and interactions with the workload management software.

Table II presents the performance impact of the assistant on the total verification time. In the first example, ADPCM



(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.

Fig. 8: Stages of e-graph growth starting from the initial e-graph in Figure 6.

Decoder, the EC tool efficiently proves the correctness of the two designs, meaning that the overhead of the assistant is detrimental, increasing total runtime. It is worth noting that the intermediate proofs do help reduce the solve time.

In the remaining benchmarks, the baseline EC tool takes longer to prove equivalence. The introduction of intermediate proofs reduces the EC solve time by up to 465x, when we just compare the EC tool runtimes and discount the assistant'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. The assistant can effectively select the most optimal solver orchestration script per intermediate proof, which greatly helps performance. This is possible because the assistant 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 an Intel provided benchmark and 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 reach the implementation design. To minimize runtime, we deploy the faster greedy extraction method. To solve the EC problem, $S^* \cong I^*$, we default to one of the slower but more powerful solver orchestration scripts. In this case, the $S^* \cong I^*$ EC problem takes significantly longer to prove than the other sub-problems. In general, as the assistant is able to deploy longer sequences of dependent rewrites, corresponding to more iterations of rewriting, we expect to find S^* and I^* that are increasingly close. In Table II, we reported box filter results based on five iterations of rewriting. If we instead limit the e-graph 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.

The box filter results highlight a tradeoff between resource investment into generating intermediate proofs and into solv-

TABLE II: Industrial EC tool performance with and without intermediate proofs generated by the assistant. We report the baseline EC tool performance when solving the original EC problem. We also report the runtime of the e-graph assistant and the runtime of the EC tool when solving the problem with the intermediate proofs. The sum provides a total verification time for the assisted proof. The last column shows the speedup ratio achieved using the assistant. Runtimes are in seconds.

Benchmark	EC without assistance	Assistant	EC with assistance	Assisted Total	Speedup (without/with)
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**	-

TABLE III: Summary of e-graph assistant properties across the benchmarks. We report the number of rewriting iterations, the 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	Num. Iter.	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

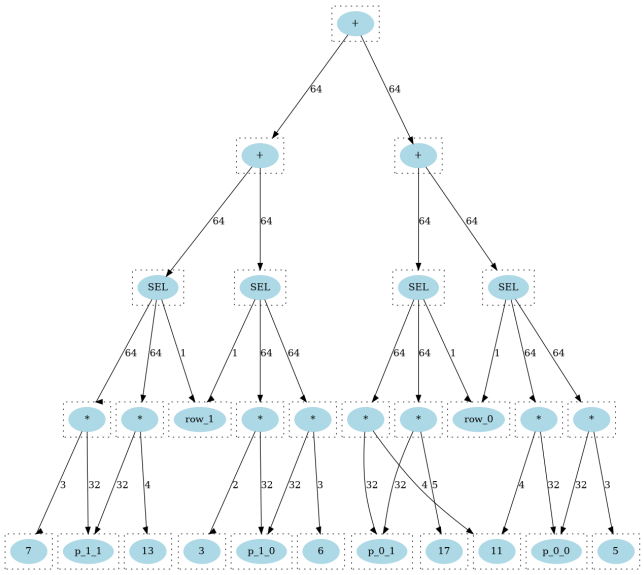


Fig. 9: Dataflow graph of the initial box filter design. The `SEL` nodes represent muxes.

ing these proofs. Understanding the turning point would allow the assistant to automatically identify which intermediates will be most beneficial, a task beyond the current tool.

In addition to the solvers described so far, we also investigated the open-source SymbiYosys equivalence checker [33]. However, for the instances we tried, we were 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 we do not typically need the full power of a SAT or SMT solver on the entire design, meaning that a specialized tool that does not target notions of completeness can prove valuable.

VI. Conclusion

This paper applies recent advances in e-graph rewriting to datapath equivalence checking to develop an automated formal verification assistant that enhances the capabilities of industrial tools. By incorporating both the specification and implementation into a single data structure, the assistant simultaneously rewrites both designs to efficiently identify common equivalent sub-expressions. From the e-graph, the tool extracts a sequence of intermediate designs, breaking the complete equivalence check into a sequence of smaller sub-proofs which can be proven 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.

The assistant developed through this work is able to find a complete sequence of intermediate designs, enabling a commercial EC tool to prove equivalence in under a second on a problem that was previously beyond its capabilities. We also demonstrated test cases where the verification assistant was able to reduce verification runtimes by up to 6x.

Future work will primarily investigate integration of the techniques presented in this paper into complete solvers to improve the rewrite engines in such tools. We will also explore different front-ends to enable C to RTL equivalence checking and will incorporate registers to facilitate equivalence checking across multiple cycles. Exploring alternative applications, such as the gate-level multiplier verification challenge discussed in the background, would highlight the generality of the approach. Lastly, there are many performance optimizations that we will make to the assistant. For example, having discovered shared classes in the e-graph, we could freeze these subgraphs to limit e-graph growth. Such optimizations and better orchestration would allow us to extend the evaluation to larger inconclusive problems requiring deeper e-graph exploration.

REFERENCES

[1] V. Pratt, "Anatomy of the Pentium bug," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 915, 1995.

[2] A. Darbari, "Smart Formal for Scalable Verification," in *Design and Verification Conference and Exhibition (DVCon) United States*, 2019.

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

[4] B. Xue, P. Chatterjee, and S. K. Shukla, "Simplification of C-RTL equivalent checking for fused multiply add unit using intermediate models," in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, 2013.

[5] E. Morini and S. Elliott, "Formal verification of integrated circuit hardware designs to implement integer division," 2019.

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

[7] 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.

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

[9] M. Stepp, R. Tate, and S. Lerner, "Equality-based translation validator for LLVM," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6806 LNCS, 2011.

[10] 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.

[11] 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, 1998.

[12] 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, 1998.

[13] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1690, 1999.

[14] ——, "Handbook of practical logic and automated reasoning," *Choice Reviews Online*, vol. 47, no. 06, 2010.

[15] 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, 1995.

[16] T. Drane and H. Jain, "Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks," in *SNUG*, 2011.

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

[18] 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*, 2021.

[19] M. Temel, A. Slobodova, and W. A. Hunt, "Automated and Scalable Verification of Integer Multipliers," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12224 LNCS, 2020.

[20] M. Kaufmann and J. S. Moore, "ACL2: An industrial strength version of Nqthm," in *COMPASS - Proceedings of the Annual Conference on Computer Assurance*, 1996.

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

[22] 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.

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

[24] 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, 2021.

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

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

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

[28] S. Coward, G. A. Constantinides, and T. Drane, "Automating Constraint-Aware Datapath Optimization using E-Graphs," 2023. [Online]. Available: https://arxiv.org/abs/2303.01839

[29] ——, "Abstract Interpretation on E-Graphs," 3 2022. [Online]. Available: https://arxiv.org/abs/2203.09191

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

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

[32] 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.

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