A Multi-Width Parametric Bitvector Equivalence Solver

Anonymous

Anonymous

Abstract. At the core of modern electronic design automation (EDA) tools is *rewriting*: a mechanism by which local transformations are iteratively applied to circuits to make them faster and more efficient. These rewrites are crucial for producing high-quality hardware, and they often depend on extremely delicate conditions, relating, for example, to the widths of the various bitvectors involved. As such, it is both *desirable* and *difficult* to prove them correct. Prior work has studied the correctness of parametric-bitwidth rewrites in the context of software compilers and SMT solvers, but those approaches struggle to handle rewrites that have *multiple* bitwidth parameters, as are commonplace in EDA. We propose a language for expressing these multi-width parametric rewrites and provide a translation into equivalences in modular arithmetic. We then show how these equivalences can be automatically and efficiently proved using equality saturation over a set of carefully chosen axioms, and finally reconstructed automatically as theorems in Isabelle. This process is implemented in a tool called ParaBit. Using benchmarks from prior compilers work and from industrial EDA tools, we demonstrate that ParaBit can solve a class of problems that are intractable using existing techniques.

Keywords: Bitvector · Modular Arithmetic · Equality Saturation.

1 Introduction

The core purpose of logic synthesis tools is to optimize a user's hardware design [24]. A key mechanism for achieving this is *rewriting*: repeatedly transforming parts of the design into behaviorally equivalent parts, in the hope of improving the overall design's performance and resource usage. Real-world synthesis tools such as Cadence Genus, Synopsys Design Compiler [45], and Yosys [44] have hundreds of rewrites; some are simple like turning `x-x` into `0`, while some are remarkably complicated and only work under delicately crafted conditions.

It is *crucial* that these rewrites are correct. Logic synthesis occurs late in the hardware design process, after most verification activities have concluded, so unsound rewrites risk introducing circuit bugs that may only be fixed by patches to the synthesis tool itself. Since synthesis users and developers often reside in different companies, such bugs may cause costly delays in the chip development process.

```
input  wire[p-1:0] a, b, c;
       wire[q-1:0] tmp;
output wire[r-1:0] res;

// BEFORE REWRITING:          // AFTER REWRITING:
assign tmp = a + b;           assign tmp = b + c;
assign res = tmp + c;         assign res = a + tmp;
```

Fig. 1: An example of a rewrite, expressed as a transformation in Verilog

Yet, at the same time, it is *difficult* to ensure that these rewrites are correct. One reason is that many rewrites are over expressions that involve bitvectors whose widths are parametric, and we need the rewrites to be correct for all values of those parameters.

Consider a parametric Verilog example (Figure 1), adapted from Coward et al. [10]. Following the semantics of Verilog [41,17] and using e_u to abbreviate $e \bmod 2^u$, the rewrite shown is sound if the following equivalence holds:

$$((a_p + b_p)_q + c_p)_r \equiv (a_p + (b_p + c_p)_q)_r. \tag{1}$$

What are the conditions on p, q, and r under which that equivalence holds? Coward et al. gave, without proof, a condition which they claim is sufficient:

$$(q > p \wedge r > q) \vee (q \geq r). \tag{2}$$

Accordingly, a synthesis tool can safely apply this transformation upon matching a concrete instance as long as the integer values for p, q, and r satisfy (2).

We would like to prove – foundationally and automatically – that (2)⇒(1), but existing tools cannot handle examples like this. For instance, although standard satisfiability modulo theories (SMT) solvers such as Bitwuzla [31] and CVC5 [2] are able to reason automatically about bitvector equivalences like (1), they can only handle cases where the bitwidths are constants. Proprietary verification tools, such as Cadence Jasper [38] and Synopsys VCFormal [18], are similarly unable to reason about parametric bitvector problems, although these tools largely focus on proving properties of specific circuit designs.

The work that comes closest is by Berger et al. [4], who proposed an extension to SMT-Lib that allows bitvectors to have parametric widths. However, their solution does not quite work in our situation, because they attach the width parameters to *operators*. For instance, they define a $+_u$ operator for parametric u that adds two u-bit inputs and produces a u-bit output. This is suitable for the software compiler setting that they considered, but we need an operator that takes inputs and produces outputs of *different* widths. To express our $(a_p + b_p)_q$ using Berger et al.'s approach, we must pad or truncate depending on the sizes:

$$\begin{aligned}&\textbf{if } p < q \textbf{ then } \texttt{zero\_extend}(a, q) +_q \texttt{zero\_extend}(b, q)\\ &\textbf{else } \texttt{truncate}((a +_p b), q)\end{aligned}$$

The ensuing proliferation of case-splits can ruin the performance of an automatic solver (as we shall see in Section 6).

Thus, in this paper we propose a representation in which bitwidth parameters are attached to *operands*, thus allowing equivalences like (1) to be encoded directly. We call this representation `BWLang` (Section 4), and we provide a set of `BWLang` axioms (Section 5.1), which are proven in the Isabelle proof assistant [34].

Unfortunately, naïve greedy rewriting over these axioms is insufficient for constructing proofs, so we turn to equality saturation [40,42]. Equality saturation has become a popular approach for implementing transformations in the context of compilers [40] and, more recently, logic synthesis tools [10]. In our work, we employ it "the other way round": we provide a candidate equivalence between `BWLang` expressions, and use equality saturation to seek a derivation of it using our set of axioms (Section 5.2).

If a derivation *is* found, we are able to extract from the e-graph the specific axioms that were used and how they were composed, and thus produce an Isabelle script that gives a foundational proof that the candidate equivalence is correct (Section 5.3), and hence that the original rewrite is sound.

In summary, our contributions are:

- a concise representation for multi-width parametric bitvector transformations, inspired by modular arithmetic (Section 4),
- an equality saturation-based solver that automatically constructs a chain of reasoning from a set of conditional modular arithmetic axioms (Section 5),
- a formalized certificate checker in Isabelle for the proposed theory (Section 5.3), and
- a comprehensive evaluation and comparison with the state-of-the-art parametric bitvector solver (Section 6).

2 Motivational Example

To demonstrate the general approach of composing a set of modular arithmetic axioms, we will now prove (2) ⇒ (1), which we repeat here.

$$\begin{array}{rll} \text{(precondition)} & \forall p, q, r \in \mathbb{N}^+ & (q > p \wedge r > q) \vee (q \geq r) \Rightarrow \\ \text{(transformation)} & \forall a, b, c \in \mathbb{Z} & \underbrace{((a_p + b_p)_q + c_p)_r}_{\text{lhs}} \equiv \underbrace{(a_p + (b_p + c_p)_q)_r}_{\text{rhs}} \end{array}$$

In this representation the inputs are integers ($a, b, c \in \mathbb{Z}$), which are constrained to be p-bit bitvectors (a_p, b_p, c_p). The intermediate results are stored in q-bit bitvectors and the final summations are stored in r-bit bitvectors. Note that the precondition is a function only of the bitwidth parameters and is expressed in disjunctive normal form (DNF). To simplify the presentation, we shall prove the statement assuming just one clause of the precondition ($q \geq r$).

The proof uses the following axiom of modular arithmetic:

$$u \geq v \Rightarrow ((x \bmod 2^u) + y) \bmod 2^v \equiv (x + y) \bmod 2^v. \tag{3}$$

The required sequence of transformations involves applying that axiom on both the lhs and rhs to remove the intermediate mod operation, as follows:

$$\begin{array}{ll} ((a_p + b_p)_q + c_p)_r & \\ \quad \equiv & \{ \text{ using (3) with condition } q \geq r \ \} \\ ((a_p + b_p) + c_p)_r & \\ \quad \equiv & \{ \text{ using associativity of integer addition } \} \\ (a_p + (b_p + c_p))_r & \\ \quad \equiv & \{ \text{ using (3) with condition } q \geq r, \text{ in reverse } \} \\ (a_p + (b_p + c_p)_q)_r & \end{array} \tag{4}$$

Here we manually constructed the correct sequence of axiom applications. To automatically discover correct sequences using a much larger set of axioms we use equality saturation to efficiently enumerate all possible sequences of axiom applications, up to a computational limit. In Isabelle we implement a formal checker, that checks certificates generated by the equality-saturation-based solver.

3 Background

We provide a brief introduction to bitvector theories, including recent extensions to support bitvectors of parametric width, and solvers for these bitvector theories. We will then describe the e-graph data structure and equality saturation rewriting technique, which we use to automate proof construction (Section 5).

3.1 Bitvector Theories

SMT-LIB defines a theory of constant-width bitvectors T_{BV} [3]. For all constants $n \in \mathbb{N}^+$, SMT-LIB defines $\texttt{BV}_n$ as the type of bitvectors of width n. In T_{BV}, all arithmetic and logical binary operators (excluding extraction, concatenation and extension) satisfy the following signature,

$$\texttt{BV}_n \times \texttt{BV}_n \to \texttt{BV}_n. \tag{5}$$

That is, the operands and result all have the same width – essentially specifying the bitwidth at the operation level. The T_{BV} theory has been used extensively to phrase hardware and software verification challenges [22,13,43]. The most effective automated solvers for T_{BV} combine bitvector-level rewriting with eager bit-blasting to a SAT solver [31].

In 2025, Berger et al. defined T_{PBV}, which extends T_{BV} to support bitvectors of parametric width [4]. The new theory provides a single type `PBV` for bitvectors of parametric size and defines new operators that take `PBV` inputs and produce `PBV` outputs. Unlike T_{BV}, which enforces width equality in the sort system, T_{PBV} instead generates additional proof obligations that ensure operand widths match. These proof obligations can, and often do, involve parametric width variables. The T_{PBV} solver also incorporate a set of rewrites, which are carefully selected from the existing rewrites for T_{BV} then made parametric. The solver defines a

translation from T_{PBV} to integer arithmetic plus a small set of uninterpreted function symbols and adds admissibility constraints [32,4]. Once translated to integer arithmetic they then combine existing integer solvers with a procedure to iteratively refine over-approximations of the uninterpreted function symbols [4]. We will similarly consider parametric bitvector problems as integer arithmetic problems, but will define a new representation that yields more concise integer arithmetic encodings in the multi-width case (Section 4).

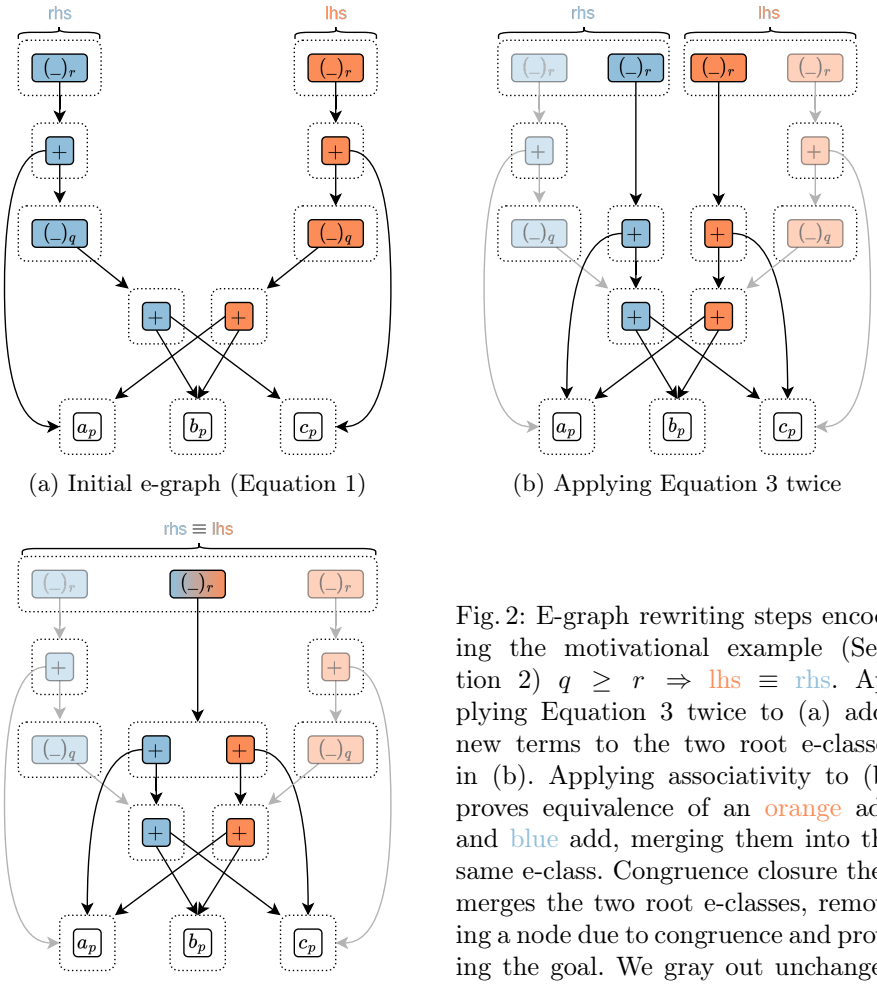
3.2 E-Graphs and Equality Saturation

The e-graph is a data structure used to represent a set of terms and an equivalence relation over that set of terms, closed under congruence [19,30]. Nodes in the graph represent either function symbols of non-zero arity or constants or variables which are zero-arity symbols. To provide a compact representation equivalent terms are grouped into equivalence classes, known as e-classes, by their root function symbol. Edges in the e-graph connect nodes to e-classes and represent operator inputs (Figure 2). To enable efficient congruence closure, each e-class is represented using a union-find data structure [39].

Equality saturation is a program rewriting technique that applies rewrites directly to the e-graph [40,42]. Traditional term rewriting finds a term matching a rewrite's left-hand side (lhs) pattern and replaces it with the rewrite's right-hand side (rhs), making the appropriate substitutions [1]. In contrast, equality saturation adds the rhs to the matched e-class, monotonically growing the set of terms represented by the e-graph. The introduction of e-classes complicates the pattern matching procedure as each e-class may contain several matching terms, leading to the development of the e-matching procedure [26]. As new terms are added to the e-graph the congruence invariant needs to be restored. This can be deferred to improve efficiency by batching rewrites and applying them in iterations that alternate between adding terms and congruence closure [42].

Given a set of rewrite rules, equality saturation terminates when further rewrite rule applications no longer add new terms to the e-graph. We say that the e-graph has saturated. In practice, and as we do in this work, it is common to terminate equality saturation early, after achieving the desired goal or after some computational limit is reached. We can encode the motivational example (Section 2) in an e-graph and use equality saturation to prove the goal with the given axioms and congruence closure (Figure 2).

E-graphs can be found at the heart of modern SMT solvers [27,2], although without typically using equality saturation to directly rewrite the e-graph. Equality saturation has gained popularity since the release of the `egg` library that introduced a number of innovations to improve performance and limit the computational overhead of equality saturation [42]. The `egg` library later gained the ability to produce a proof, essentially a sequence of rewrite applications, to justify why two terms were merged into the same e-class [14]. Researchers have developed numerous applications using `egg`, including tools to improve the numerical stability of programs [35], hardware design automation [7], and lemma discov-



(a) Initial e-graph (Equation 1)

(b) Applying Equation 3 twice

(c) By associativity and congruence

Fig. 2: E-graph rewriting steps encoding the motivational example (Section 2) $q \geq r \Rightarrow$ lhs $\equiv$ rhs. Applying Equation 3 twice to (a) adds new terms to the two root e-classes in (b). Applying associativity to (b) proves equivalence of an orange add and blue add, merging them into the same e-class. Congruence closure then merges the two root e-classes, removing a node due to congruence and proving the goal. We gray out unchanged nodes to highlight changes.

ery [20]. We build a solver using `egg` (Section 5.2) and use its proof production capabilities to generate a certificate that we formalize in Isabelle (Section 5.3).

4 BW-Lang

We introduce a new language, `BWLang` (Figure 3), to express bitvector equivalences containing *multiple* bitwidth variables. In this language operators can be applied to bitvectors of different widths, and the result of the operation can itself assume a different width. To achieve this we introduce the $(\_)\_ : \mathbb{N}^+ \to \mathbb{Z} \to \mathbb{N}$ operator, which, given a positive non-zero natural number representing a width and an integer, returns the least positive residue of the integer modulo two to

$$
\begin{array}{lll}
\mathbb{Z} & \ni n & \\
\mathbb{N}^+ & \ni w & \\
\mathsf{Var} & \ni a, p & \\
\mathsf{BVExpr} & \ni B \quad ::= a \mid n \mid (B)_W \mid \texttt{signed}(W, B) & \\
 & \quad \mid B \ll B \mid B \gg B \mid B \boxtimes B & \boxtimes \in \{+, -, \div, \times, \&, \oplus, |\} \\
 & \quad \mid (ite\, B, B, B) \mid \boxminus B & \boxminus \in \{-, \sim\} \\
\mathsf{WidthExpr} & \ni W \quad ::= W \boxplus W \mid \mathrm{pow}_2(W) \mid p \mid w & \boxplus \in \{+, -, \times\} \\
\mathsf{Precond} & \ni P \quad ::= P \wedge P \mid W \diamond W & \diamond \in \{>, \geq, <, \leq\}
\end{array}
$$

Fig. 3: `BWLang` grammar. P describes the representable preconditions.

the power of the width. The result is an arbitrary bitvector if the width is non-positive. Bitvector operations are performed by interpreting the bitvectors as natural numbers (akin to SMT-LIB bv2nat), performing the integer operation and then recasting the result as a bitvector (akin to SMT-LIB nat2bv). Division by zero and logical shifts by negative integers are interpreted arbitrarily.

ParaBit is capable of solving a subset of the language admitted by the `BWLang` grammar, which we call well-formed.

Definition 1 (Well-Formed BVExpr). *A given $B \in \mathsf{BVExpr}$, is well-formed if every constant, variable and operation result in B is cast to a bitvector, meaning it is wrapped in a $(\_)_p$ operation for some $p \in \mathsf{WidthExpr}$.*

For example, $(a_p{+}1_1)_q$ is well-formed but $(a_p{+}1)$ is not. Intuitively, this restricts ParaBit to the domain where every intermediate result has a specified, although potentially parametric, width. ParaBit will reject input BVExprs that are not well-formed. We do not enforce this restriction in the `BWLang` grammar, as the axioms (Section 5.1) will remove bitvector casts when provably sound according to `BWLang` semantics (Section 4.1).

The restricted `BWLang` subset is able to capture bitvector expressions such as those defined in T_{PBV} [4], by encoding the widths of each operand and destination to be identical. Further, we are able to reason about parametric bitvector problems similar to those expressible in Verilog, where each operand can take on a different width and it is truncated or zero-extended accordingly. Indeed, zero-extension and truncation amount to the same operation in `BWLang`, i.e., changing the width of some expression to a new width, by nesting bitvector casts $((\_)_p)_q$. The difference lies in the fact that a truncation assumes the outer width is smaller than the inner, whereas extension assumes the opposite. Therefore, based on the provided conditions, the operator will be interpreted as either of the operations, but if no condition is provided it captures both.

4.1 Semantics

The semantics of `BWLang` are defined by the following denotation functions:

$$
\begin{array}{lll}
\mathcal{B}[\![\_]\!]_{\pi,\rho} & : \mathsf{BVExpr} & \to ((\mathsf{WEnv} \times \mathsf{Env}) \to \mathbb{Z}) \\
\mathcal{W}[\![\_]\!]_{\pi} & : \mathsf{WidthExpr} & \to (\mathsf{WEnv} \to \mathbb{Z}) \\
\mathcal{P}[\![\_]\!]_{\pi} & : \mathsf{Precond} & \to (\mathsf{WEnv} \to \mathsf{Bool})
\end{array}
$$

where the width environment $\mathsf{WEnv} : \mathsf{Var} \to \mathbb{N}^+$ captures the mapping of width parameters to positive non-zero integers and the environment $\mathsf{Env} : \mathsf{Var} \to \mathbb{Z}$ captures the mapping of the remaining variables in the BVExpr to integers. Preconditions are interpreted ($\mathcal{P}$) following the standard rules of Boolean algebra and integer comparisons.

The interpretation of a WidthExpr ($\mathcal{W}$) is standard integer arithmetic:

$$\begin{aligned}
\mathcal{W}[\![p]\!]_\pi &= \pi(p) \\
\mathcal{W}[\![w]\!]_\pi &= w \\
\mathcal{W}[\![\mathrm{pow}_2(w)]\!]_\pi &= \mathrm{pow}_2(\mathcal{W}[\![w]\!]_\pi) \\
\mathcal{W}[\![a \boxplus b]\!]_\pi &= \mathcal{W}[\![a]\!]_\pi \boxplus \mathcal{W}[\![b]\!]_\pi.
\end{aligned}$$

The interpretation of a BVExpr ($\mathcal{B}$) is defined via modular arithmetic.

$$\begin{aligned}
\mathcal{B}[\![a]\!]_{\pi,\rho} &= \rho(a) \\
\mathcal{B}[\![n]\!]_{\pi,\rho} &= n \\
\mathcal{B}[\![(B)_W]\!]_{\pi,\rho} &= \mathcal{B}[\![B]\!]_{\pi,\rho} \bmod \mathrm{pow}_2(\mathcal{W}[\![W]\!]_\pi) \\
\mathcal{B}[\![\mathtt{signed}(W,B)]\!]_{\pi,\rho} &= 2 \times \mathcal{B}[\![(B)_{\mathcal{W}[\![W-1]\!]_\pi}]\!]_{\pi,\rho} - \mathcal{B}[\![(B)_{\mathcal{W}[\![W]\!]_\pi}]\!]_{\pi,\rho} \\
\mathcal{B}[\![B \gg C]\!]_{\pi,\rho} &= \mathcal{B}[\![B]\!]_{\pi,\rho} \div \mathrm{pow}_2(\mathcal{B}[\![C]\!]_{\pi,\rho}) \\
\mathcal{B}[\![B \ll C]\!]_{\pi,\rho} &= \mathcal{B}[\![B]\!]_{\pi,\rho} \times \mathrm{pow}_2(\mathcal{B}[\![C]\!]_{\pi,\rho}) \\
\mathcal{B}[\![B \boxtimes C]\!]_{\pi,\rho} &= \mathcal{B}[\![B]\!]_{\pi,\rho} \boxtimes \mathcal{B}[\![C]\!]_{\pi,\rho} \\
\mathcal{B}[\![(ite\, B, C, D)]\!]_{\pi,\rho} &= \mathcal{B}[\![(B)_1]\!]_{\pi,\rho} \times \mathcal{B}[\![C]\!]_{\pi,\rho} + \mathcal{B}[\![(\sim (B)_1)_1]\!]_{\pi,\rho} \times \mathcal{B}[\![D]\!]_{\pi,\rho} \\
\mathcal{B}[\![\boxminus B]\!]_{\pi,\rho} &= \boxminus \mathcal{B}[\![B]\!]_{\pi,\rho}
\end{aligned}$$

The $\mathtt{signed}(W, B)$ operation treats B as a two's complement encoded bitvector of length W. Its interpretation is the two's complement conversion to an integer. Letting $a = \mathcal{B}[\![a']\!]_{\pi,\rho}$, $p = \mathcal{W}[\![p']\!]_\pi$, we can rearrange the natural definition to recover the interpretation given, where $(a \gg (p-1))_1$ extracts the sign-bit.

$$\begin{aligned}
\mathcal{B}[\![\mathtt{signed}(p', a')]\!]_{\pi,\rho} &= a_{p-1} - \left(2^{p-1} \times (a \gg (p-1))_1\right) \\
&= 2 \times a_{p-1} - a_{p-1} - \left(2^{p-1} \times (a \gg (p-1))_1\right) \\
&= 2 \times a_{p-1} - a_p
\end{aligned}$$

Bitwise operations are defined recursively over the integers following the Isabelle `Bit_Operation` definitions [15]. For example, & is defined as:

$$a \,\&\, b = isodd(a) \wedge isodd(b) + 2 \times ((a \div 2) \,\&\, (b \div 2)).$$

4.2 Problem Definition

We now define a BW problem that can be solved by ParaBit, as a tuple of type ($\mathsf{Precond} \times \mathsf{BVExpr} \times \mathsf{BVExpr}$), containing a precondition and two well-formed

BVExpr terms. Given an instance of a BW problem, (P, B, C) ParaBit attempts to prove $P \Rightarrow B \equiv C$ which is shorthand for:

$$\forall \pi \in \mathsf{WEnv}.\, \mathcal{P}[\![P]\!]_\pi \implies \forall \rho \in \mathsf{Env}.\, \mathcal{B}[\![B]\!]_{\pi,\rho} = \mathcal{B}[\![C]\!]_{\pi,\rho} \tag{6}$$

The formulation of Equation 6 means that if the statement is found to be true, this is equivalent to finding the UNSAT result for $\neg(6)$, where $\neg(6)$ is the conventional SMT formulation for proving bitvector rewrite rules [4]. The parametric case differs from bitvector problems with concrete widths (i.e., T_{BV}) in the further existential quantification of the width variables ($\exists\pi$).

$$\neg(6) = \exists \pi.\, \mathcal{P}[\![P]\!]_\pi \wedge (\exists \rho.\, \mathcal{B}[\![B]\!]_{\pi,\rho} \neq \mathcal{B}[\![C]\!]_{\pi,\rho}) \tag{7}$$

5 A solver for `BWLang`

Having defined the `BWLang` representation, we now describe how our ParaBit solver automatically finds a proof for BW problems. We generate proofs by combining `BWLang` axioms (Section 5.1) using equality saturation to efficiently explore all possible paths (Section 5.2). To reduce the trusted compute base, we formalize most of the axioms in Isabelle and generate a proof certificate that is also checked using Isabelle.

5.1 Axioms

The BW problems admitted by ParaBit are restricted to the well-formed subset of `BWLang` (Section 4.2). By contrast, ParaBit's axioms are triples of type $(\mathsf{Precond} \times \mathsf{BVExpr} \times \mathsf{BVExpr})$, comprised of a precondition and two (not necessarily well-formed) BVExpr terms, along with a direction of application. For each axiom we define, $(P, \mathit{pat}, \mathit{repl})$, we prove using Isabelle (Section 5.3) that:

$$\forall \pi \in \mathsf{WEnv}.\, \mathcal{P}[\![P]\!]_\pi \implies \forall \rho \in \mathsf{Env}.\, \mathcal{B}[\![\mathit{pat}]\!]_{\pi,\rho} = \mathcal{B}[\![\mathit{repl}]\!]_{\pi,\rho}. \tag{8}$$

Whilst essentially the same as Equation 6, the axioms we define are far simpler to prove than the general class of BW problems.

As ParaBit primarily reasons about integer arithmetic, we define a set of axioms that convert certain bitvector operations to integer arithmetic (Table 1 - top). We combine these definitional axioms with a second set of conditionally applied axioms that simplify `BWLang` expressions involving $(\_)_p$ operations (Table 1 - bottom). It is these axioms that depend heavily upon the preconditions that accompany each BW problem. Namely, given a particular axiom, say "add.rm.l.prec", when we find a match we get an assignment to each of the free inputs (p, q, a, b) in the "Pattern". After replacing each free input by its matched assignment, we then check whether the "Precondition" is implied by the preconditions given in the BW problem. Equality saturation will make matching axiom patterns more complicated but also make checking the "Precondition" simpler (Section 5.2). Although we convert certain bitvector operations (e.g., $\gg$) to

Table 1: The first-half are axioms converting bitvector operations to arithmetic, whilst the second-half are axioms removing or simplifying bitvector casts $(\_)_p$. The $\leftrightarrow$ indicates that the rule is applied in both directions. The Isa. column indicates whether the axiom has been proven in Isabelle.

Name	Precondition	Pattern	Replacement	Isa.
lshift.def	True	$a_p \ll b_q$	$\rightarrow a_p * 2^{b_q}$	✓
rshift.def	True	$a_p \gg b_q$	$\rightarrow a_p \div 2^{b_q}$	✓
ite.def	True	$(ite\ c_1\ a_p\ b_p)$	$\rightarrow c_1 * a_p + (\sim c_1)_1 * b_p$	✓
xor.def	True	$a_p \oplus b_p$	$\leftrightarrow (a_p \| b_p) - (a_p \& b_p)$	✓
signed.def	True	$\texttt{signed}(p, a)$	$\rightarrow 2 * a_{p-1} - a_p$	✓
bw.1	True	1_p	$\rightarrow 1$	✓
bw.0	True	0_p	$\rightarrow 0$	✓
add.rm.l.prec	$q \geq p$	$(a_q + b)_p$	$\rightarrow (a + b)_p$	✓
add.rm.r.prec	$q \geq p$	$(a + b_q)_p$	$\rightarrow (a + b)_p$	✓
add.full.prec	$(q < p) \wedge (r < p)$	$(a_q + b_r)_p$	$\rightarrow a_q + b_r$	✓
diff.rm.l.prec	$q \geq p$	$(a_q - b)_p$	$\rightarrow (a - b)_p$	✓
diff.rm.r.prec	$q \geq p$	$(a - b_q)_p$	$\rightarrow (a - b)_p$	✓
mul.full.prec	$r \geq (p + q)$	$(a_q * b_p)_r$	$\rightarrow a_q * b_p$	✓
mul.rm.prec	$p \geq q$	$(a_p * b)_q$	$\rightarrow (a * b)_q$	✓
mul.bit	$q \geq p$	$(a_p * b_1)_q$	$\rightarrow a_p * b_1$	✓
div.gte	$p \geq q$	$(a_q \div b)_p$	$\rightarrow a_q \div b$	✓
div.floor	$b \geq 2^p$	$a_p \div b$	$\rightarrow 0$	✓
mod.inner	$q \geq p$	$(a_p)_q$	$\rightarrow a_p$	✓
mod.outer	$p \geq q$	$(a_p)_q$	$\rightarrow a_q$	✓
mul.pow2	$r \geq p + 2^q - 1$	$(a_p * 2^{b_q})_r$	$\rightarrow a_p * 2^{b_q}$	✓
xor.one	True	$a_p \oplus 1$	$\rightarrow 2 * (a_p \div 2) + (\sim a_1)_1$	✗
rshift.mod	$p - q \geq b$	$(a_p \gg b)_q$	$\rightarrow (a \gg b)_q$	✗
rshift.const	$b \geq 0$	$(a_p \gg b)$	$\rightarrow (a \div 2^b)$	✗
rm.signed	True	$\texttt{signed}(p, a_p)_p$	$\rightarrow a_p$	✗
zext.signed	$q > p$	$\texttt{signed}(q, a_p)$	$\rightarrow a_p$	✗
neg.signed	$q > p$	$\texttt{signed}(q, (-a_p)_q)$	$\rightarrow -a_p$	✗
diff.signed	$r > p \wedge r > q$	$\texttt{signed}(r, (a_p - b_q)_r)$	$\rightarrow a_p - b_q$	✗

arithmetic, thanks to equality saturation we are still able to reason about these operations (e.g., "shift.mod") as they are not removed from the e-graph. Axioms marked with a ✓ (Table 1) are mechanized in Isabelle. Example Isabelle proofs and the yet to be mechanized (✗) axioms will be discussed later (Section 5.3).

The axioms described so far attempt to reduce the equivalence check to reasoning over Boolean and integer arithmetic. To tackle this remaining problem, ParaBit incorporates standard axioms of Boolean and integer arithmetic (associativity, commutativity, distributivity etc). Whilst associativity and commutativity are traditionally considered a shortcoming of equality saturation, the rewrite verification problem setting means that we rarely encounter long chains of associative operators, and hence rarely suffer the e-graph explosion problem.

```
input  wire[p  :0] a;
       wire[p  :0] tmp;
output wire[p-1:0] res;

// OPTION 1:                          // OPTION 2:
                                      assign tmp = a + 1'b1;
assign res = a[p-1:1] + a[0];         assign res = tmp[p:1];
```

Fig. 4: Unsigned round-to-nearest implementations in Verilog.

One additional axiom, "half.adder" unrolls the definition of addition.

$$a_p + b_q \rightarrow (a_p \oplus b_q) + 2 * (a_p \,\&\, b_q) \tag{9}$$

The rule applies a half-adder then sums the results, but can be recursively applied to the "Replacement" creating a source of exponential e-graph growth (Section 5.2). Whilst costly, "half.adder" proves extremely helpful in proving BW problems that mix Boolean and arithmetic operations, as is common in circuit optimization. Consider two equivalent unsigned round-to-nearest implementations (Figure 4). Applying "half.adder" to `OPTION 2` we can then prove it equivalent to `OPTION 1` by simplifying Boolean operations with a constant argument (`1'b1`) and applying standard arithmetic axioms.

5.2 Equality Saturation

We define `BWLang` using an S-expression syntax (not presented here), allowing us to develop ParaBit using the `egg` equality saturation library. Given a BW problem, (P, lhs, rhs), we initialize the e-graph with the *lhs* and *rhs* BVExprs. ParaBit's goal is to unify the root e-classes of each expression, L and R, respectively. We then create a *special* e-class $C_{\texttt{True}}$ and add each P_i expression to it, where $P = \bigwedge_i P_i$, converting the precondition P to an assumption. We will use this e-graph to encode a contextual relation. Namely, given an e-class c, if $x, y \in c$ then $P \Rightarrow x \equiv y$. Recent work in contextual equality saturation has identified the challenge of handling multiple contexts within a single e-graph [9,16,36]. For example, if $P_1 \Rightarrow x \equiv y$ and $P_2 \Rightarrow x \equiv z$, $P_1 \vee P_2 \not\Rightarrow y \equiv z$, yet a standard e-graph only encodes a single equivalence relation, so reasoning under such a disjunction of preconditions can lead to incorrect derivations. Instead of turning to complex solutions for encoding multiple equivalence relations in an e-graph [36], we resolve this by restricting BW problem preconditions to conjunctions of preconditions (Precond). This means that the original motivational problem (Section 2) corresponds to two BW problems, with a separate e-graph for each clause of the DNF. Whilst this means that we must rediscover common equalities, e.g. those for which $P_1 \vee P_2 \Rightarrow y \equiv z$, in practice we did not encounter scalability or runtime issues to motivate a more complex solution.

Having initialized the e-graph, we now run iterations of equality saturation applying all the axioms (Section 5.1) at every iteration. Each time a conditional

```
theorem motivational_example:
  fixes p q r :: nat
  fixes a b c :: int
  assumes "p > 0" and "r > 0" and "q > 0"
  assumes "q >= r"
  shows "(bw r ((bw q ((bw p a) + (bw p b))) + (bw p c)))
        = (bw r ((bw p a) + (bw q ((bw p b) + (bw p c)))))"
        (is "?lhs = ?rhs")
proof -
  have "?lhs = (bw r (((bw p a) + (bw p b)) + (bw p c)))"
    using add_rm_l_prec assms by simp
  moreover have "... = (bw r ((bw p a) + ((bw p b) + (bw p c))))"
    using add.assoc assms by metis
  moreover have "... = (bw r ((bw p a) + (bw q ((bw p b) + (bw p c)))))"
    using add_rm_r_prec assms by simp
  ultimately show ?thesis by argo
qed
```

Fig. 5: Isabelle script for proving the motivational example (Section 2). In Isabelle `(bw p a)` is used to denote the bitvector cast operation a_p.

axiom is matched via e-matching, ParaBit performs the substitutions to generate a Precond comprised of variables declared in the input BW problem. ParaBit then checks whether that expression is contained within the $C_{\texttt{True}}$ e-class. If it is, the "Replacement" expression is then added to the matched e-class. We also implement a basic constant folding to handle trivial comparisons, such as $2 > 0$.

After each iteration of axiom application, ParaBit terminates immediately if L and R have been merged. If ParaBit terminates having achieved its goal, then it extracts a sequence of axiom applications that maps the *lhs* expression to the *rhs* expression using `egg`'s proof production feature [14].

5.3 Isabelle Formalization

We leverage `egg`'s proof-producing capabilities to obtain an equational explanation proving the equivalence of BVExprs *lhs* and *rhs*. The explanation is a sequence of equivalent expressions, where consecutive expressions differ by the application of a single axiom. From this, we generate an Isabelle proof script that transforms L into R through a chain of justified equational steps.

The reconstruction proceeds by introducing intermediate subgoals, one for each step in the explanation, where each subgoal asserts the equality between consecutive expressions. Each transformation is then justified by applying the specific axiom responsible for that rewriting step. To discharge these subgoals, we employ a hierarchy of tactics with increasing automation: we first attempt `simp only` to resolve the goal using only the relevant axiom, then fall back to more powerful methods such as `auto` and `metis` when contextual reasoning or assumption handling is required.

```
lemma add_rm_l_prec:
  "(bw p ((bw q a) + b)) = (bw p (a + b))"
  if "q >= p"
by (metis bw_def le_imp_power_dvd mod_add_cong mod_mod_cancel that)

lemma add_rm_r_prec:
  "(bw p (a + (bw q b))) = (bw p (a + b))"
  if "q >= p"
using add_rm_l_prec by (simp add: add.commute that)
```

Fig. 6: Isabelle scripts proving the `add.rm.l.prec` and `add.rm.r.prec` axioms.

Each problem instance is formalized as an Isabelle theorem, with the preconditions encoded as assumptions in the theorem statement. We present a readable version of an Isabelle script (Figure 5) that ParaBit generates to prove the correctness of the motivational example (Section 2). We provide proofs for two of the axioms required to construct the proof (Figure 6). Note that the `add.assoc` and `add.commute` axioms are native to Isabelle.

Mechanizing proofs for all axioms in Isabelle is an ongoing effort. We do not anticipate any fundamental difficulties in mechanizing the remaining axioms.

6 Evaluation

We have implemented ParaBit in approximately 2000 lines of Rust code, building on the `egg` library to perform equality saturation using a total of 99 axioms, of which 92 are encoded in Isabelle. The remaining seven axioms are supported by pen-and-paper proofs, which we are continuing to mechanize.

We also developed an automatic translator between `BWLang` and PBV to allow for comparison against prior work [4]. As noted earlier (Section 1), each multi-width `BWLang` problem corresponds to potentially many PBV problems due to the ensuing proliferation of case-splits. We enumerate all valid case-splits and generate a separate PBV problem for each branch. We consider one BW problem solved if all the corresponding PBV problems can be solved.

6.1 Benchmarks

We evaluate ParaBit using benchmark sets arising from both hardware and software domains. Given ParaBit's particularly focus on multi-width problems, we separate each benchmark set into those containing multiple parametric widths (i.e., p, q) and those containing a single width (i.e., just p).

Alive consists of peephole optimizations used by LLVM-based compilers that were verified via enumeration by Alive [23]. The evaluation of Berger et al. [4] included 200 such peephole optimizations. 62 can be represented using `BWLang`, which we generate by converting from the PBV representation.

Hydra consists of software compiler peephole optimizations that were automatically generalized to support parametric width expressions using program synthesis techniques [28]. From a total set of 198 peephole optimizations, `BWLang` can express 70 whilst PBV can express 149. We generate `BWLang` and PBV directly from the Souper representation used by Hydra.

Rover consists of circuit design optimizations used by the Rover optimization tool [8,10], that targeted arithmetic designs written in Verilog. Only the 2022 work contains both the transformations and the necessary preconditions [8]. `BWLang` can express 21 out of 27 transformations described. Encoding in `BWLang` produces 28 BW problems, and only 27 of these can be converted to PBV, due to the presence of a power of two in the precondition. We convert the Rover rules directly to `BWLang`, from which we automatically generate 3283 PBV problems.

Industry consists of representative circuit design optimizations employed by Cadence's Genus logic synthesis tool. Cadence was willing to share 7 such optimizations, of which both PBV and `BWLang` can express 5. As for Rover, we convert the Cadence representation directly to `BWLang` and use it to generate 49 PBV problems.

Although not fundamentally difficult, at present `BWLang` is able to encode fewer problems than PBV due to unsupported operators (bitvector comparisons) and an inability to encode data constraints (e.g., a matched variable is actually a constant that is a power of two). More fundamentally, an equality saturation-based solver makes it challenging to encode existential quantifiers. We only include the Alive set from the evaluation of Berger et al. as the remaining sets either include existential quantifiers or did not contain any problems with multiple parametric widths, which is the focus of this work.

6.2 Experimental Setup

We compared ParaBit against the state-of-the-art parametric bitvector solver from Berger et al. [4], which we call PBV when reporting results. Since PBV can represent a larger class of problems, we present results for PBV on both the full set of benchmarks that can be represented by their language and results running PBV on the restricted subset that ParaBit can admit.

We ran the experiments single-threaded on an Intel i7-7700k CPU running at 4.5GHz. For each solver and benchmark instance we set a 60 second CPU time limit and a memory limit of 8GB. Similar to the results observed by Berger et al. [4], increasing the time limit had little impact on the results; that is, if either solver was able to construct a proof it usually did so in well under the time limit. Whilst PBV never breached the memory limit of the system, the memory demands of equality saturation meant that ParaBit did breach the memory limit for 15 benchmarks.

Table 2: Number of solved benchmarks for each configuration. VBS represents the virtual best solver over PBV and ParaBit. The unique rows indicate how many benchmarks are *only* solved by that solver on the restricted subset. We bold the best results on the restricted subset representable in `BWLang`.

Width	Benchmark Set	PBV [4]		ParaBit	VBS
		Full	Restricted	(our work)	
Single	Alive	104/196	31/61	**53/61**	129/196
	Hydra	55/74	**26/30**	24/30	56/74
	Rover	3/3	**3/3**	2/3	3/3
	Industry	2/4	2/4	**3/4**	4/4
	unique	-	7	28	-
	solved (%)	60%	64%	85%	71%
Multiple	Alive	3/4	0/1	**1/1**	4/4
	Hydra	33/75	9/40	**32/40**	56/75
	Rover	7/24	7/24	**25/25**	25/25
	Industry	0/1	0/1	**1/1**	1/1
	unique	-	3	42	-
	solved (%)	41%	24%	88%	72%

6.3 Results

We summarize the results (Table 2) for the different benchmark sets (Section 6.1), splitting them into their single and multiple width subsets. We also include the virtual best solver (VBS) across both ParaBit and PBV.

From these results we can see that on the subset of the benchmarks that ParaBit supports, we are able to solve a much higher proportion than PBV. Most noticeably, for benchmarks involving multiple parametric widths we see that ParaBit dramatically outperforms PBV. Whilst not surprising given the benchmarks used to evaluate PBV contained fewer instances involving multiple parametric widths, these results show that our approach works particularly well for the kind of problems commonly found in EDA (e.g., Rover, Industry). The Alive benchmarks provide the most meaningful comparison point, as they were part of the original PBV evaluation. In the single-width case we again see that ParaBit solves a much larger proportion of the problems it can support, suggesting that the equality saturation-based approach offers a major advantage.

PBV does outperform ParaBit on both the Hydra and Rover single width subsets. Interestingly, the Rover benchmark that is uniquely solved by PBV is the relatively simple transformation, $(ite\, b_1, a_p, a_p)_p \to a_p$. This proves challenging for Rover due to the way in which *ite* is lowered to arithmetic, creating a non-trivial algebraic rewriting challenge. For two Industry benchmarks, it was necessary to add additional trivial-to-prove preconditions to the BW problem, e.g., $(w{+}1){-}w \geq 1$, to overcome limitations in ParaBit's precondition reasoning.

Table 3: Average (mean) statistics for solved ParaBit benchmarks.

		ParaBit		PBV
Benchmark Set	E-graph Nodes	Proof Steps	Time (s)	Time (s)
Alive	17,000	5200	0.84	0.47
Hydra	580	26	0.02	0.08
Rover	1300	25	0.05	0.16
Industry	14,000	53	0.43	0.61

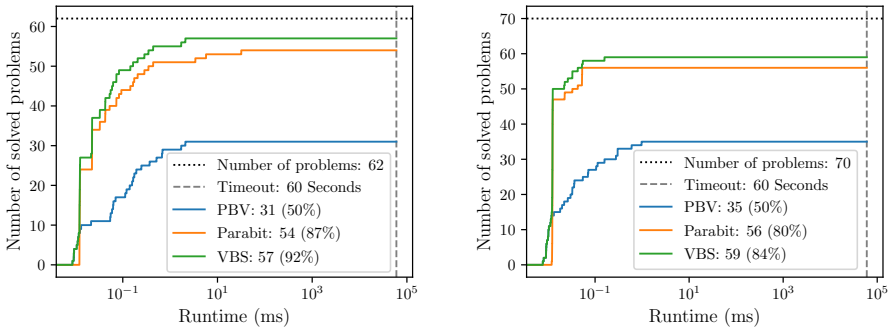
When evaluating PBV on the Rover benchmark, the solver either solved all the instances of a given `BWLang` problem or none, with a notable exception in the case of the associativity of multiplication, where it can solve 67% of the generated PBV queries (1254).

For benchmarks in the Hydra [28], Rover [8] and Industry sets these results provide the first automated correctness proofs quantified over all possible bitwidths. Prior to ParaBit, each of these transformations had only been verified by proving the correctness for different combinations of concrete widths.

Unlike PBV, each ParaBit proof is accompanied by a proof certificate (Section 5.3), that can be checked by Isabelle to increase confidence in the generated proof. Of the 131 problems solved by ParaBit, Isabelle was able to check 116 of the generated certificates. We were unable to verify eight Hydra and three Industry benchmarks as their proofs used axioms not yet mechanized in Isabelle (✗-Table 1). Three of the Alive benchmark certificates could not be verified as either there was an error in our translation to Isabelle (one case), or the number of steps involved was too large for Isabelle to verify within a 5 minute timeout (two cases). The longest proof contains over 200,000 steps, however this appears to be an issue with `egg`'s proof production feature as the proof contains many identical steps. Specifically, the proof contains loops where no progress towards the goal is actually being made, causing an explosion in the size of the proof.

We also provide solver statistics (Table 3) averaged across benchmarks that the solvers were able to solve. Comparing PBV and ParaBit average time to solution, it is difficult to draw strong conclusions, but we see that both typically find a proof in under a second. We can see that across the benchmark sets the size of e-graphs varied significantly and was reasonably correlated with average solve time. The average number of steps in the proof was less variable, with the exception of the Alive benchmarks. Removing the large uncheckable outliers from the Alive set, the average was reduced to 200 steps. We attribute this to the conversion from PBV to `BWLang` which produces a more verbose encoding of the problem.

We present survival plots comparing the performance of our work with PBV for Hydra and Alive (Figure 7), where we consider only the subset representable in `BWLang`. We omit the Industry benchmark due to its small size, and the ROVER benchmarks because each problem corresponds to multiple PBV instances, making direct comparison difficult. In both benchmarks we observe the



(a) Alive (b) Hydra

Fig. 7: Survival plots comparing Parabit, PBV and the virtual best solver (VBS) on the Alive and Hydra benchmarks. The comparison is on the restricted subset representable in `BWLang`.

improved performance of Parabit, in terms of the number of problems solved and the speed at which they are solved. However, in the Alive case (Figure 7a), it is interesting to note how the problems last solved by Parabit *do not* contribute to the virtual best solution, meaning they have already been solved by PBV.

7 Related Work

SMT Across a series of works the CVC5 SMT solver [2] has developed theory engines for parametric bitvector theories. In their 2021 work on the subject [33], Niemitz et al. propose a translation from parametric bitvector formulas to non-linear integer arithmetic with uninterpreted functions. In 2025 they proposed the `PBV` theory that we introduced earlier (Section 3.1) and developed a solver which we compared against (Section 6). Unlike `BWLang` their representation attached bitvector widths to operations rather than operands. Additionally, their solver relied on existing SMT theory solvers combined with a set of simplification rewrites, whilst we turned to equality saturation.

Rewrite Verification Given their central role in compilers, solvers and many other applications, there has been plenty of research into the verification of rewrite rules. In the compilers domain, the Alive series of works [23,22] provided SMT-LIB semantics for the LLVM intermediate representation and a domain specific language for expressing rewrites over concrete bitwidths. An SMT backend verifies the correctness of the proposed transformation. The interactive theorem prover community has also developed several automated decision procedures for proving compiler peephole rewrites [5,29], one of which they later extended to support parametric bitvectors with a single parametric width [6]. In the SMT

community, the IsaRare work [21] formalized rewrites used during proof certificate checking in Isabelle. Whilst we also employ Isabelle for formalization, we aim to provide push-button verification of the rewrites, for a different class of multi-width parametric bitvectors problems arising from the hardware domain.

Equality Saturation One of the earliest applications of equality saturation performed translation validation for Java compilers [40,37]. In the hardware domain, researchers extended the ROVER work's foundations [8] to develop a basic equivalence checker that used equality saturation to construct an equivalence proof for specific circuits [11]. Whilst the ROVER representation is visually similar to `BWLang`, it actually encodes operand and output widths in the operator definition, something `BWLang` specifically avoids. These works target concrete program instances, but if combined with the approach we presented here, they could, for example, prove the equivalence of bitwidth-parameterized circuit components.

8 Conclusion

In this paper, we have developed automatic techniques to prove the correctness of parametric bitvector equivalences, particularly focusing on the circuit transformations at the heart of EDA tools. We introduced the new `BWLang` representation which, unlike previous work on this problem, attaches width parameters to *operands* rather than *operators*. This allows for concise representation of multi-width bitvector problems, which are common in the hardware domain. We constructed a set of axioms, mostly formalized using Isabelle, that we combined to construct proofs using equality saturation. The techniques were implemented in an automated solver, ParaBit, that also produces proof certificates that can be checked using Isabelle. Evaluation on a diverse set of benchmarks demonstrated ParaBit's advantage over existing methods, solving 86% of the problems that could be encoded in `BWLang`.

Future work will explore the use of symbolic interval analyses to provide a more general reasoning engine that can capture data constraints as well as constraints on width parameters. More practically, we will explore whether the recently introduced grind tactic in Lean can provide a more integrated solver, as grind also conducts an e-graph-based search [25]. We will also extend `BWLang` to cover more operators and plan to integrate ParaBit with the CIRCT framework [12] to allow for Alive-style verification of circuit design transformations.

References

1. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge university press (1998)
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength smt solver. Tools and Algorithms for the Construction and Analysis of Systems pp. 415–442 (2022). `https://doi.org/10.1007/978-3-030-99524-9_24`

3. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (smt-lib) (2016), https://smt-lib.org/
4. Berger, Z., Zohar, Y., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-precise reasoning with parametric bit-vectors. In: 28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025). pp. 4–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2025)
5. Bhat, S., Keizer, A., Hughes, C., Goens, A., Grosser, T.: Verifying Peephole Rewriting in SSA Compiler IRs. In: Bertot, Y., Kutsia, T., Norrish, M. (eds.) 15th International Conference on Interactive Theorem Proving (ITP 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 309, pp. 9:1–9:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.ITP.2024.9
6. Bhat, S., Stefanesco, L., Hughes, C., Grosser, T.: Certified decision procedures for width-independent bitvector predicates. Proc. ACM Program. Lang. **9**(OOPSLA2) (Oct 2025). https://doi.org/10.1145/3763148
7. Cheng, J., Coward, S., Chelini, L., Barbalho, R., Drane, T.: 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 Operating Systems. pp. 1029–1044. Association for Computing Machinery (7 2024). https://doi.org/10.1145/3620665.3640392
8. Coward, S., Constantinides, G.A., Drane, T.: Automatic datapath optimization using e-graphs. In: 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH). pp. 43–50 (2022). https://doi.org/10.1109/ARITH54963.2022.00016
9. Coward, S., Drane, T., Constantinides, G.A.: Constraint-aware e-graph rewriting for hardware performance optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems pp. 1–14 (2024). https://doi.org/10.1109/TCAD.2024.3483096
10. Coward, S., Drane, T., Constantinides, G.A.: Rover: Rtl optimization via verified e-graph rewriting. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **43**, 4687–4700 (12 2024). https://doi.org/10.1109/TCAD.2024.3410154
11. Coward, S., Morini, E., Tan, B., Drane, T., Constantinides, G.: Datapath verification via word-level e-graph rewriting. In: Formal Methods in Computer-Aided Design (FMCAD) (10 2023). https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_17
12. Eldridge, S., Barua, P., Chapyzhenka, A., Izraelevitz, A., Koenig, J., Lattner, C., Lenharth, A., Leontiev, G., Schuiki, F., Sunder, R., Young, A., Xia, R.: MLIR as Hardware Compiler Infrastructure. In: WOSET '21: Workshop on Open Source EDA Technology (2021)
13. Emmer, M., Khasidashvili, Z., Korovin, K., Voronkov, A.: Encoding industrial hardware verification problems into effectively propositional logic. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. p. 137–144. FMCAD '10, FMCAD Inc, Austin, Texas (2010)
14. Flatt, O., Coward, S., Willsey, M., Tatlock, Z., Panchekha, P.: Small proofs from congruence closure. In: Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design, FMCAD 2022. p. 9. TU Wien Academic Press (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2-13
15. Haftmann, F.: Theory bit operations (2025), https://isabelle.in.tum.de/library/HOL/HOL/Bit_Operations.html#Bit_Operations.semiring_bit_operations_class

16. Hou, T., Laddad, S., Hellerstein, J.M.: Towards relational contextual equality saturation (2024), https://tylerhou.com/contextual-eqsat.pdf
17. IEEE: IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. Tech. Rep. IEEE Std 1800-2023, IEEE (2023). https://doi.org/10.1109/IEEESTD.2024.10458102
18. Koelbl, A., Jacoby, R., Jain, H., Pixley, C.: Solver technology for system-level to rtl equivalence checking. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 196–201. European Design and Automation Association (2009). https://doi.org/10.1109/date.2009.5090657
19. Kozen, D.: Complexity of finitely presented algebras. In: Proceedings of the ninth annual ACM symposium on Theory of computing. pp. 164–177 (1977)
20. Kurashige, C., Ji, R., Giridharan, A., Barbone, M., Noor, D., Itzhaky, S., Jhala, R., Polikarpova, N.: Cclemma: E-graph guided lemma discovery for inductive equational proofs. Proceedings of the ACM on Programming Languages **8**, 818–844 (8 2024). https://doi.org/10.1145/3674653
21. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: Isarare: Automatic verification of smt rewrites in isabelle/hol. In: Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I. p. 311–330. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-57246-3_17
22. Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: bounded translation validation for llvm. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 65–79 (2021)
23. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. SIGPLAN Not. **50**(6), 22–32 (Jun 2015). https://doi.org/10.1145/2813885.2737965, https://doi.org/10.1145/2813885.2737965
24. Micheli, G.D.: Synthesis and optimization of digital circuits. McGraw-Hill Higher Education (1994)
25. de Moura, L.: Lean grind tactic (2025), https://lean-lang.org/doc/reference/latest/The--grind--tactic/
26. Moura, L.D., Bjørner, N.: Efficient e-matching for smt solvers. In: Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction. vol. 4603 LNAI, pp. 183–198. Springer-Verlag (2007). https://doi.org/10.1007/978-3-540-73595-3_13
27. Moura, L.D., Bjørner, N.: 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, pp. 337–340. Springer-Verlag (2008). https://doi.org/10.1007/978-3-540-78800-3_24
28. Mukherjee, M., Regehr, J.: Hydra: Generalizing peephole optimizations with program synthesis. Proc. ACM Program. Lang. **8**(OOPSLA1) (Apr 2024). https://doi.org/10.1145/3649837
29. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for compcert. SIGPLAN Not. **51**(6), 448–461 (Jun 2016). https://doi.org/10.1145/2980983.2908109, https://doi.org/10.1145/2980983.2908109
30. Nelson, C.G.: Techniques for program verification. Ph.D. thesis, Stanford University (1980)
31. Niemetz, A., Preiner, M.: Bitwuzla. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture

Notes in Bioinformatics). vol. 13965 LNCS (2023). https://doi.org/10.1007/978-3-031-37703-7_1
32. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: 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). https://doi.org/10.1007/978-3-030-29436-6_22
33. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards satisfiability modulo parametric bit-vectors. Journal of Automated Reasoning **65**(7), 1001–1025 (2021)
34. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
35. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. vol. 50, pp. 1–11. Association for Computing Machinery (2015)
36. Singher, E., Itzhaky, S.: Easter egg: Equality reasoning based on e-graphs with multiple assumptions. In: Formal Methods in Computer-Aided Design. p. 70 (2024)
37. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for llvm. In: Proceedings of the 23rd International Conference on Computer Aided Verification. vol. 6806 LNCS, pp. 737–742. Springer-Verlag (2011). https://doi.org/10.1007/978-3-642-22110-1_59
38. Systems, C.D.: Jasper formal verification platform (2024), https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html
39. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM (JACM) **22** (1975). https://doi.org/10.1145/321879.321884
40. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. vol. 44, pp. 264–276. Association for Computing Machinery (2009). https://doi.org/10.1145/1480881.1480915
41. Thomas, D., Moorby, P.: The Verilog hardware description language. Springer Science & Business Media (2008)
42. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. In: Proceedings of the ACM on Principles of Programming Languages. vol. 5. Association for Computing Machinery (2021). https://doi.org/10.1145/3434304
43. Wolf, C.: Symbiyosys (sby)-front-end for yosys-based formal verification flows (2017)
44. Wolf, C., Glaser, J.: Yosys-a free verilog synthesis suite. In: Proceedings of Austrochip (2013), https://yosyshq.net/yosys/files/yosys-austrochip2013.pdf
45. Zimmermann, R.: Datapath synthesis for standard-cell design. In: 19th IEEE Symposium on Computer Arithmetic. pp. 207–211 (2009). https://doi.org/10.1109/ARITH.2009.28