Bit-Precise Interpolation in Bitwuzla*

Aina Niemetz and Mathias Preiner

Stanford University, Stanford, USA
{niemetz, preiner}@cs.stanford.edu

Abstract. Bitwuzla is a state-of-the-art SMT solver specialized in theories relevant to bit-precise reasoning. The main bit-vector solving procedure of Bitwuzla is based on bit-blasting, a reduction of bit-vector constraints to propositional logic (SAT). Until now, Bitwuzla did not support interpolant generation, which is a key requirement for many verification applications that rely on bit-precise reasoning. We present an extension of Bitwuzla with the capability to produce interpolants and interpolation sequences for quantifier-free bit-vector formulas. Our interpolation workflow extracts bit-level interpolants from proofs produced by the back-end SAT solver, which are lifted to the word-level and post-processed to recover and simplify word-level structure. We evaluate our new bit-vector interpolation engine in the context of various interpolation-based algorithms for symbolic model checking.

1 Introduction

Many applications in computer-aided verification rely on bit-precise reasoning at the back end. One prominent example is symbolic model checking [19], both in the context of software and hardware verification, where the representation of machine integers with bit-precise semantics is a key requirement. In recent years, advances in Satisfiability Modulo Theories (SMT) research and solving enabled the lifting of classic SAT-based model checking techniques to SMT [9]. This allows for modeling of systems that require more expressive reasoning.

Interpolation-based verification techniques that utilize Craig interpolants [20] to generalize abstractions are the basis for many state-of-the-art model checking algorithms (e.g., [16, 42, 50, 51]). Applications that implement such algorithms based on bit-precise reasoning as provided by the SMT theory of fixed-size bit-vectors therefore require the solver back end to produce bit-vector interpolants. Producing interpolants, however, is a feature that is not standardized in the SMT-LIB format [8] and not widely supported by SMT solvers.

Bitwuzla [44] is a state-of-the-art SMT solver specialized in theories relevant for bit-precise reasoning. It supports the theories of fixed-size bit-vectors, arrays, floating-point arithmetic and uninterpreted functions and their combinations. The main bit-vector solving procedure in Bitwuzla is an abstraction

* This work was supported in part by the Stanford Center for Automated Reasoning, the Stanford Center for Blockchain Research, and a gift from Amazon Web Services.

refinement approach based on bit-blasting [46], which reduces bit-vector constraints to propositional logic (SAT).

In this paper, we extend Bitwuzla with the capability to produce interpolants and interpolation sequences for quantifier-free bit-vector formulas. Our interpolation workflow computes bit-level interpolants from the proofs produced by the back-end SAT solver of the bit-vector solver. These bit-level interpolants are then lifted to the word-level and post-processed to recover and simplify word-level structure. We evaluate our interpolation engine in the context of various interpolation-based engines implemented in the model checkers Pono [40] and Kind2 [14, 39], which both integrate Bitwuzla as one of their back-end solvers.

Related Work. Interpolation techniques for other theories, e.g., linear integer arithmetic, have been investigated in great detail [12]. Bit-vector interpolation, on the other hand, and especially techniques based on bit-blasting, have received relatively little attention over the years. The majority of interpolation techniques for quantifier-free bit-vector formulas rely on a reduction of bit-vector constraints to integer arithmetic [6, 28, 47]. Techniques based on such a reduction are not applicable in our context since Bitwuzla does not support integer arithmetic.

The main bit-vector interpolation technique for bit-blasting-based SMT procedures is a naive reconstruction of propositional bit-level interpolants on the word-level via bit extraction, Boolean conjunction and negation. This technique is complete but results in purely bit-level interpolants and a loss of word-level structure. Kroening et al. [37] attempted to mitigate this loss of word-level structure by lifting the propositional proof of unsatisfiability of a bit-vector interpolation problem to the word-level, prior to extracting the interpolant. This technique is, however, limited to the fragment of equality logic. The same authors later proposed an interpolation procedure based on a specialized, rewriting-based decision procedure [38], but limited to a fragment of the bit-vector theory. Our technique is also based on lifting bit-level interpolants to the word-level, augmented with additional post-processing and simplification strategies to recover more word-level structure, and not limited to a fragment of the bit-vector theory.

Few state-of-the-art SMT solvers provide support for bit-vector interpolation. MathSAT [17] implements a layered approach [28] where various incomplete interpolation techniques, including via a reduction to linear integer arithmetic, are combined with the standard bit-level technique as a fallback. Interpolants produced by the fallback technique are purely bit-level interpolants as no strategies to recover word-level structure are applied to the interpolant.

cvc5 [7] implements a generic interpolation approach where the interpolation query is translated into a Syntax-Guided Synthesis (SyGuS) query whose solutions are interpolants. SMTInterpol [15] and Princess [6] compute bit-vector interpolants via a reduction of the interpolation query to non-linear integer arithmetic. Yices2 [22] produces so-called model interpolants [27], which are derived within the MCSAT framework as explanations for why a partial model of formula B (over common symbols) cannot be extended to a full model of formula A.

Z3 [43] exposes interpolation as a reduction to solving non-recursive Constrained Horn Clauses (CHC) via its CHC solver Spacer [36]. Spacer imple-

ments a model-based projection procedure for the arithmetic fragment of the bit-vector theory (which falls back to value-based projection for operators outside of the fragment) to avoid bit-blasting-based interpolation techniques [34]. This approach iteratively approximates quantifier elimination. A layered, quantifier-elimination-based approach that also avoids bit-blasting but is limited to the linear arithmetic bit-vector fragment was proposed in [33].

2 Preliminaries

We assume the usual notions and terminology of many-sorted first-order logic with equality (see, e.g., [24, 41]). A *theory* is a pair $(\Sigma, \mathcal{I})$ where Σ is a signature consisting of sort symbols and sorted function symbols, and $\mathcal{I}$ is a class of Σ-interpretations. We assume that Σ includes equality and a designated sort Bool, values $\top$ (true) and $\bot$ (false) of sort Bool, and Boolean connectives defined as usual. We use the usual inductive definition of the satisfiability relation between Σ-interpretations and Σ-formulas. A Σ-formula is *T-satisfiable* (resp. *T-unsatisfiable*) if it is satisfied by some (resp. no) interpretation in $\mathcal{I}$; it is *T-valid* if it is satisfied by all interpretations in $\mathcal{I}$. We assume the usual definition of well-sorted terms, literals, and formulas.

We refer to 0-arity function symbols as *constants* and use $\mathcal{S}(t)$ and $\mathcal{S}(\varphi)$ for the set of uninterpreted symbols that appear in term t and formula φ, respectively. Similarly, for a formula $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n$, also denoted as $\{\varphi_1, \ldots, \varphi_n\}$, we use $\mathcal{S}(\varphi)$ for the set of uninterpreted symbols in $\mathcal{S}(\varphi_1) \cup \ldots \cup \mathcal{S}(\varphi_n)$. We write $\varphi[x_1, \ldots, x_n]$ to denote a formula φ defined over (a subset of) uninterpreted symbols $\{x_1, \ldots, x_n\}$. We further use $\varphi[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ for the formula obtained from φ by simultaneously replacing each occurrence of x_i with a_i.

We focus on the theory of fixed-size bit-vectors $T_{BV} = (\Sigma_{BV}, \mathcal{I}_{BV})$ as defined by the SMT-LIB 2 standard [8]. Signature Σ_{BV} includes a unique sort $\sigma_{[w]}$ for each bit-width w, function symbols overloaded for every $\sigma_{[w]}$, and all *bit-vector values* of sort $\sigma_{[w]}$ for each w. We denote a *bit-vector term* x of sort $\sigma_{[w]}$ as $x_{[w]}$, and omit w from the notation when it is clear from the context. We refer to the bit at index i of $x_{[w]}$ as $x[i]$ and represent a bit-vector value $v_{[w]}$ as a bit-string of 0s and 1s, with the most significant bit (MSB) as the left-most bit at index $w-1$, and the least significant bit (LSB) as the right-most bit at index 0.

3 Interpolation

We briefly review background and definitions relevant to interpolation for quantifier-free bit-vector formulas as implemented in our SMT solver Bitwuzla [44]. The main procedure for solving quantifier-free bit-vector formulas in Bitwuzla is based on bit-blasting, which reduces bit-vector constraints to SAT. Our interpolation procedure is based on extracting an interpolant for the original bit-vector problem from the resolution refutation of its reduction to SAT. It integrates both Pudlák's [49] and McMillan's [42] interpolation systems for computing propositional interpolants from clausal resolution proofs of unsatisfiability.

Definition 1 (Craig Interpolant [20]). *Given T-formula $\varphi = \{\varphi_1, \ldots, \varphi_n\}$ such that φ is T-unsatisfiable, i.e., $\bigwedge_1^n \varphi_i \models \bot$. Let (A, B) be a partitioning of φ into formulas A and B with $A \cap B = \emptyset$, i.e., $A \wedge B$ is T-unsatisfiable and $A \Rightarrow \neg B$ is T-valid. An* interpolant *of A and B is a T-formula I such that*

(i) $A \Rightarrow I$ *is T-valid,*
(ii) $I \wedge B$ *is T-unsatisfiable, and*
(iii) $\mathcal{S}(I) \subseteq \mathcal{S}(A) \cap \mathcal{S}(B)$.

Given an (A, B) partitioning of φ, we use $\mathcal{S}_G = \mathcal{S}(A) \cap \mathcal{S}(B)$ for the set of *global symbols* in φ. We further denote the sets of *A-local* and *B-local* symbols as $\mathcal{S}_A = \mathcal{S}(A) \setminus \mathcal{S}_G$ and $\mathcal{S}_B = \mathcal{S}(B) \setminus \mathcal{S}_G$, respectively.

Based on such an (A, B) partitioning, a *term* t is global or local depending on its uninterpreted symbols: if $\mathcal{S}(t) \subseteq \mathcal{S}_G$, it is *global*; if its set of local symbols $\mathcal{S}_L(t) = \mathcal{S}(t) \setminus \mathcal{S}_G$ is non-empty and $\mathcal{S}_L(t) \subseteq \mathcal{S}_A$ or $\mathcal{S}_L(t) \subseteq \mathcal{S}_B$, it is *$A$-local* or *$B$-local*; and if $\mathcal{S}_L(t) \subseteq \mathcal{S}_A \cup \mathcal{S}_B$, it is *$AB$-mixed*. We use $t \in \mathcal{S}_A$, $t \in \mathcal{S}_B$ and $t \in \mathcal{S}_G$ as shorthand for *labeling* t as A-local, B-local and global, respectively.

Definition 2 (Interpolation Sequence [31]). *Given T-formula $\varphi = \{\varphi_1, \ldots, \varphi_n\}$ such that $\bigwedge_1^n \varphi_i \models \bot$. Let $\langle (A_1, B_1), \ldots, (A_n, B_n) \rangle$ be a sequence of n partitions of φ with $A_i = \{\varphi_1, \ldots, \varphi_i\}$ and $B_i = \{\varphi_{i+1}, \ldots, \varphi_n\}$ for $1 \leq i < n$. An* interpolation sequence *$\langle I_0, \ldots, I_n \rangle$ is a sequence of T-formulas such that*

(i) $I_0 = \top$ *and* $I_n = \bot$,
(ii) I_i *is an interpolant for* (A_i, B_i), *and*
(iii) $I_i \wedge \varphi_{i+1} \Rightarrow I_{i+1}$.

Note that by definition, terms in A and B are never AB-mixed. However, solving procedures for SMT may introduce AB-mixed terms (e.g., via lemmas) that appear in a proof of unsatisfiability for $A \wedge B$. In the context of our bit-vector interpolation procedure, we never encounter AB-mixed terms. We explain in more detail why this is the case in Section 4.

3.1 SAT-Based Interpolation

A *literal* l is either a propositional variable v or its negation $\neg v$. A *clause* $C = l_1 \vee \ldots \vee l_n$, also denoted as $\{l_1, \ldots, l_n\}$, is a disjunction of literals l_i. A formula $\phi = C_1 \wedge \ldots \wedge C_m$, also denoted as $\{C_1, \ldots, C_m\}$, in Conjunctive Normal Form (CNF) is a conjunction of clauses C_j. Given two clauses $C_1 = v \vee D_1$ and $C_2 = \neg v \vee D_2$, with D_1 and D_2 clauses that do not contain complementary literals. Then $D_1 \vee D_2$ is the *resolvent* of C_1 and C_2, also denoted as $res(C_1, C_2, v)$.

Given an unsatisfiable formula ϕ, partitioned into two clause sets A and B with $A \cap B = \emptyset$, an interpolant for A and B is defined as in Definition 1. Labeling of symbols and terms as A-local, B-local and global as above naturally extends to propositional variables and literals.

Definition 3 (Clausal Resolution Proof). *Given an unsatisfiable formula ϕ in CNF, a* clausal resolution proof *Π for ϕ is a directed acyclic graph (DAG) with clauses $V_\Pi = \{C_1, \ldots, C_k, D_{k+1}, \ldots, D_n\}$ as vertices such that*

(a) *root vertices $\{C_1, \ldots, C_k\} \subseteq \phi$,*
(b) *each $D \in \{D_{k+1}, \ldots, D_n\}$ has exactly two antecedents $A_1, A_2 \in V_\Pi$,*
(c) *$D = res(A_1, A_2, v)$, and*
(d) *the single leaf vertex of Π is $\{\}$ (the empty clause).*

Our interpolation procedure integrates the interpolation system proposed by McMillan [42] as the main procedure for computing propositional interpolants from a clausal resolution proof. Given $\phi = A \wedge B$ and a proof $\Pi(\phi)$, McMillan's algorithm recursively constructs an interpolant I from partial interpolants of every resolvent in Π based on the following, *asymmetric* construction rules.

Definition 4. *Let A and B be the disjoint partitions of a CNF formula ϕ, and let C be a clause in $\Pi(\phi)$. A* ***partial interpolant*** *I_p of C is defined via McMillan's* asymmetric *construction as follows.*

$$I_p(C) = \begin{cases} \{l \mid l \in C \wedge l \in \mathcal{S}_G\} & C \in A \\ \top & C \in B \\ I_p(C_1) \vee I_p(C_2) & C = res(C_1, C_2, v), v \in \mathcal{S}_A \\ I_p(C_1) \wedge I_p(C_2) & C = res(C_1, C_2, v), v \notin \mathcal{S}_A \end{cases}$$

Optionally, our interpolation procedure supports the interpolation system based on *symmetric* construction of partial interpolants proposed by Pudlák [49]. Pudlák's construction rules are similar to McMillan's rules but differ for A-clauses and in the case when pivot v of a resolved clause is not labeled as A-local.

Definition 5. *Let partitions A and B and clause C be defined as in Definition 4. If C is a resolvent $res(C_1, C_2, v)$, recall that $v \in C_1$ and $\neg v \in C_2$. A* ***partial interpolant*** *I_p of C is defined via Pudlák's* symmetric *construction as follows.*

$$I_p(C) = \begin{cases} \bot & C \in A \\ \top & C \in B \\ I_p(C_1) \vee I_p(C_2) & C = res(C_1, C_2, v), v \in \mathcal{S}_A \\ I_p(C_1) \wedge I_p(C_2) & C = res(C_1, C_2, v), v \in \mathcal{S}_B \\ (I_p(C_1) \vee v) \wedge (I_p(C_2) \vee \neg v) & C = res(C_1, C_2, v), v \in \mathcal{S}_G \end{cases}$$

4 Bit-Vector Interpolation Workflow

Our bit-vector interpolation procedure is implemented in Bitwuzla [44], an SMT solver for the (quantified and quantifier-free) theory of fixed-size bit-vectors and combinations with arrays, floating-point arithmetic and uninterpreted functions. The main solving procedure for quantifier-free bit-vector formulas in Bitwuzla

is a CEGAR-style abstraction refinement procedure based on bit-blasting [46], combined with term rewriting and preprocessing techniques to simplify input constraints prior to the actual reduction step to SAT. This procedure introduces abstractions for arithmetic operations defined over large bit-widths that are expensive for the underlying SAT solver when translated to the bit-level.

The workflow of our interpolation procedure is given in Figure 2. The main components of the procedure are the *Solver*, which determines the satisfiability of the input formula, and the *Interpolator*, which extracts a bit-level interpolant from the SAT proof, and post-processes and lifts it back to the word-level. In the following, we discuss the relevant components of our workflow in more detail.

4.1 SMT-LIB and API Interface

The SMT-LIB 2.7 language [8] does not yet standardize interactions with the solver in case of interpolant generation. Previous extensions to support interpolation implemented by cvc5 [7], MathSAT [17], OpenSMT2 [32] and SMTInterpol [15] only agree on enabling interpolant generation via `(set-option :produce-interpolants true)`. We extended Bitwuzla's SMT-LIB interface in a similar way to SMTInterpol's extension by introducing a new command `(get-interpolants` $\langle terms \rangle^+$`)`, with *terms* defined as a list of terms `(`$\langle term \rangle^+$`)` (corresponding to a conjunction of terms), and *term* defined as in SMT-LIB.

For an input formula $\varphi = A \wedge B$, we require that A and B are asserted, and that the T-unsatisfiability of φ has been determined via `check-sat`. Command `get-interpolants` must be issued after a `check-sat` command but prior to any subsequent `pop` commands. A *single* interpolant is queried with a single term list, which represents the A partition of the interpolation query. Assertions not in A make up partition B. An *interpolation sequence* is queried by specifying the list of A increments of the sequence. For example, `(get-interpolants (`a_1`) (`a_2`) (`a_3`))` computes interpolants $\langle I_1, I_2, I_3 \rangle$ for A-partitions $\{a_1\}$, $\{a_1, a_2\}$ and $\{a_1, a_2, a_3\}$ such that $I_1 \wedge a_2 \Rightarrow I_2$ and $I_2 \wedge a_3 \Rightarrow I_3$. The resulting sequence of interpolants is returned as a term list. Figure 1 shows an example in SMT-LIB format, with the output of the query shown as comments below the corresponding command. We extended the API of Bitwuzla to support single and sequence interpolation queries in a similar way.

4.2 Bit-Vector Solving in Bitwuzla

Adding support for generating bit-vector interpolants in Bitwuzla requires the extension of its bit-blasting pipeline with capabilities to produce SAT proofs. The bit-vector solver and its extensions for interpolation are shown on the left in Figure 2. In this section, we briefly review the main workflow of its bit-vector solving procedure. We discuss its extension to produce SAT proofs in Section 4.3.

Given a bit-vector formula φ, as the first step, Bitwuzla applies various *preprocessing* techniques to simplify φ. These techniques can be divided into *local* and *global* simplifications. Local simplifications, e.g., term rewriting, are independent from the current set of assertions, while global simplifications are not.

```
(set-logic QF_BV)
(set-option :produce-interpolants true)
(declare-const x1 (_ BitVec 2))
(declare-const x2 (_ BitVec 2))
(declare-const x3 (_ BitVec 2))
(assert (! (bvslt (_ bv0 4) (bvsub (concat (_ bv0 2) x1) (_ bv1 4))) :named a1))
(assert (! (= x2 x1) :named a2))
(assert (! (= x3 ((_ extract 1 0) (bvneg (concat (_ bv0 2) x2)))) :named a3))
(assert (= x3 (_ bv0 2)))
(check-sat)
; unsat
(get-interpolants (a1 a2))
; (
; (not (= x2 #b00))
; )
(get-interpolants (a1) (a2) (a3))
; (
; (= #b0 ((_ extract 3 3) (bvadd (concat #b00 x1) #b1111)))
; (not (= x2 #b00))
; (not (= x3 #b00))
; )
```

Fig. 1: Interpolation example.

As a consequence, when applied *across* partitions, global simplifications may "pollute" the A-partition with B-local symbols and vice versa. Safely applying simplifications across partitions therefore requires tracking of such transformations during solving, and reconstructing the resulting SAT proof with respect to these simplifications when extracting the interpolant. Vizel et al. [52] address a similar problem in the context of interpolant generation for SAT-based bounded model checking. Limiting global simplifications to within a partition, on the other hand, requires that the partition is known at the time of preprocessing. Since the partitioning is defined by interpolation queries after the satisfiability check and queries for interpolation sequences require a dynamic AB-partitioning we cannot apply global preprocessing techniques within a partition. Thus, when interpolant generation is enabled, we currently only apply preprocessing passes that perform local simplifications. Extending our interpolation procedure to allow for application of global simplifications is non-trivial and left to future work.

As mentioned above, Bitwuzla implements a counterexample-guided abstraction refinement (CEGAR) [18] procedure for bit-vector arithmetic based on bit-blasting [46]. Thus, in the next step, the abstraction module replaces abstracted terms in the preprocessed formula φ' with fresh (uninterpreted) constants, yielding bit-vector formula φ'', which is an over-approximation of φ'. This over-approximation is iteratively refined with lemmas L until its reduction to SAT is *unsat*, or its satisfying assignment is consistent for all abstracted terms.

The bit-blasting pipeline of Bitwuzla consists of two stages. In the first stage, an And-Inverter Graph (AIG) representation of $\varphi'' \wedge L$ is constructed while applying AIG-level rewriting techniques [13]. In the second stage, this AIG circuit is converted to CNF via Tseitin transformation and sent to the SAT solver.

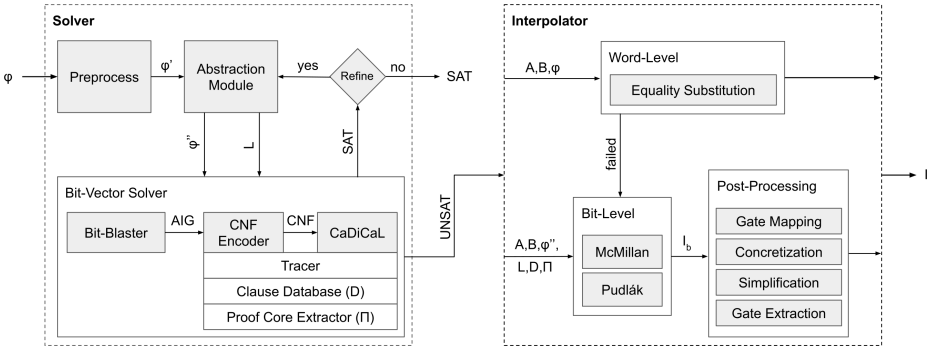


Fig. 2: The bit-vector interpolation workflow in Bitwuzla.

4.3 SAT Proofs via Tracer Interface of CaDiCaL

Bitwuzla supports multiple off-the-shelf SAT solvers as SAT back end and uses CaDiCaL [10] as its main SAT Solver. CaDiCaL provides an interface for *tracing* proof-related events such as clause addition and deletion via notifications and callbacks. This allows to define proof tracers for specific purposes and is utilized by CaDiCaL to provide user-facing printers and checkers for different proof formats, including the LRAT format [21]. For this format, CaDiCaL is required to provide justification derivation (resolution) chains for each derived clause, which was shown to only add a 5% overhead on average to solving time [48].

In its full expressive power, the LRAT format not only tracks resolution chains of clauses derived via reverse unit propagation (RUP) [25], but additional hints if the clause addition relied on the stronger resolution asymmetric tautology (RAT) property. CaDiCaL does not implement any reasoning techniques that require the latter, and thus effectively does not produce LRAT but *LRUP* proofs, i.e., RUP proofs augmented with clause ids and resolution chains [48].

As shown in Figure 2, we instrument the bit-blasting pipeline of Bitwuzla to facilitate recording proofs produced by CaDiCaL with three components: the proof tracer, the clause database, and the proof core extraction.

Tracer. The proof tracer is responsible for tracking and recording original and derived clauses and is implemented utilizing CaDiCaL's `Tracer` API. Each clause is assigned a unique identifier and stored in a clause database. Derived clauses are mapped to their antecedents, which are recorded in order of derivation. Additionally, each clause also maps to the AIG it corresponds to. Together with the bit-blaster's mapping from AIGs to T_{BV}-terms in $\varphi'' \wedge L$, this facilitates a dynamic back-mapping from clause to A/B-partition, which is required for generating sequence interpolants. We discuss partitioning of assertions and clauses, and labeling of symbols in more detail in Section 4.4.

Clause Database. Since the satisfiability query preceding an interpolation query may be non-trivial, for interpolation sequences it is desirable to not require indi-

ID	Label	Clause	Antecedents	ID	Label	Clause	Antecedents
3	A	{ 7, 2, 3 }		47	B	{ -27 }	
11	A	{ -7 }		49	B	{ -22 }	
18	A	{ 13, -2, 10 }		50		{ -21 }	{ 49, 45, 35 }
24	A	{ 16, -3, 11 }		51	B	{ -23 }	
26	A	{ -13 }		52		{ -11 }	{ 51, 47, 41 }
28	A	{ -16 }		53		{ -3 }	{ 52, 28, 24 }
31	B	{ 21, -10, 11 }		54		{ -10 }	{ 52, 50, 31 }
35	B	{ 24, -21, 22 }		55		{ 2 }	{ 53, 11, 3 }
41	B	{ 27, -11, 23 }		56		{ }	{ 55, 54, 26, 18 }
45	B	{ -24 }					

Fig. 3: Proof core of the first interpolation query in Figure 1.

vidual satisfiability queries for each interpolant. This, however, requires dynamic A/B-partitioning, decoupled from solving and proof generation. To achieve this, the bit-level interpolant is computed from the proof core, depending on the A/B-partitioning for each individual interpolation query. Since the proof core is extracted from the full proof, until CaDiCaL concludes *unsat*, we keep all recorded clauses in memory and ignore notifications about clause deletions.

Proof Core Extraction. The clause database represents the recorded proof as a DAG as in Definition 3, but with $n \geq 2$ instead of $n = 2$ antecedents for derived clauses. When CaDiCaL concludes unsat, a proof core is extracted by traversing the proof in reverse topological order, starting from the empty clause. The proof core of the first example interpolation query in Figure 1, augmented with the partition labeling for the given query, is given in Figure 3.

Discussion. The integration of CaDiCaL in Bitwuzla utilizes incremental SAT solving via solving under assumptions [23]. Recently, Khouri et al. [35] presented a CaDiCaL proof tracer implementation for producing interpolants from DRUP proofs that incorporates the proof core minimization technique presented in [29]. The authors argue that especially in the incremental case, this can lead to better interpolants for model checking applications. We leave the investigation of the impact of proof core minimization on our interpolation workflow to future work.

CaDiCaL provides a proof tracer implementation for extracting CNF interpolants [10], which supports both McMillan's and Pudlák's interpolation systems. This tracer, however, requires that the AB-partitioning is known at solve time, and is thus not suitable for our use case.

4.4 Interpolant Generation

After concluding that bit-vector formula φ is unsatisfiable and proof core Π is extracted, given a partitioning of φ into A and B, the Interpolator processes the proof core to compute interpolant I as shown in Figure 2 on the right. The Interpolator consists of three components: the *word-level* interpolator, the *bit-level* interpolator, and a *post-processor* for bit-level interpolants. Interpolant I is

extracted either via word-level interpolation or our bit-level interpolation workflow. Word-level interpolation is optional and implements a substitution-based technique that is not always applicable. If enabled and not applicable, we fall back to bit-level interpolation.

A/B-Partitioning. Word-level interpolation operates on the original input formula φ partitioned into A and B as given by the interpolation query and natively constructs a word-level interpolant. Bit-level interpolation, on the other hand, first computes a bit-level interpolant from the SAT proof core, which is then lifted to the word-level. That is, the recorded proof core is a proof for $\varphi'' \wedge L$. Thus, prior to computing the bit-level interpolant, the A/B-partitioning of φ must first be mapped to $\varphi'' \wedge L$ before mapping it to the bit-level.

Partitioning the set of clauses sent to the SAT solver into A and B partitions is straight forward: original assertions in φ are mapped to preprocessed assertions in φ'', which are then mapped to SAT clauses via the bit-blaster's mapping from T_{BV}-terms to AIGs and the clause database's mapping from AIGs to clauses.

Labeling SAT variables as A-local, B-local and global is more involved, as each stage of the bit-blasting pipeline introduces auxiliary constructs (AIGs when bit-blasting to AIG circuits, and Tseitin variables when translating these AIGs to CNF). We first partition $\mathcal{S}(\varphi)$ into $\mathcal{S}_A(\varphi)$, $\mathcal{S}_B(\varphi)$ and $\mathcal{S}_G(\varphi)$, and label all terms in φ'' based on this partitioning as described in Section 3. Term abstractions, which are fresh uninterpreted bit-vector constants from the point of view of the bit-blaster, are labeled according to the label of the corresponding abstracted term. The labeling of uninterpreted constants is mapped to the bits of their AIG representation, and each AND-gate in these AIGs is labeled based on its inputs. The AIG-labeling is then mapped to the SAT variables of its corresponding CNF translation. Note that our labeling workflow ensures that auxiliary constructs are only labeled as global if they appear in both A and B.

As a final step, we assign each lemma to partition A or B, depending on the labeling of its symbols, and label their SAT variables as above. Note that in the context of our bit-vector solving procedure, lemmas always appear as either A-local, B-local, or global, and cannot be AB-mixed. This is due to the fact that we only introduce abstractions for arithmetic operations $x \diamond s$ with $\diamond \in \{\cdot, \div, \mathrm{mod}\}$, while preserving the original label of the abstracted term for its abstraction t. Refinement lemmas for these abstractions are defined over only x, s, and t (for details, see [46]), and are therefore, by construction, never AB-mixed.

Word-Level Interpolation. Our bit-vector interpolation workflow optionally supports a simple word-level interpolation technique, referred to as *equality substitution* in [28], which constructs a word-level interpolant via term substitution of local symbols. Given $A = \{a = t, A'\}$ with $a \in \mathcal{S}_A$, then $A'[a \mapsto t]$ is an interpolant for A and B if it does not contain any A-local symbols. Similarly, if $B = \{b = t, B'\}$ and $b \in \mathcal{S}_B$, then $\neg B'[b \mapsto t]$ is an interpolant for A and B if it does not contain any B-local symbols. This technique exploits a corner case of the more general fact that existential quantifier elimination of all A-local symbols in A (or all B-local symbols in $\neg B$) yields an interpolant. In Bitwuzla,

equality substitution is implemented by utilizing its substitution preprocessing pass [44], which applies syntactic substitutions based on existing equalities and new equalities inferred based on a set of normalization techniques. In practice, we apply equality substitution to the A/B-partitions of φ' (rather than φ).

Bit-Level Interpolation. Given a SAT proof core Π and the A/B-partitioning of the set of clauses corresponding to formula φ as described above, our workflow extracts a bit-level interpolant I_b using either McMillan's (default) or Pudlák's interpolation algorithm as defined in Section 3.1. Recall that Π is represented as a DAG with $n \geq 2$ antecedents for derived clauses. These antecedents, recorded in the clause database in order of derivation, form a chain derivation [26]. Partial interpolant computation as defined in Section 3 is naturally lifted to such chains of length $n > 2$ without the need for fully expanding the recorded LRUP proof into a clausal resolution proof. Partial interpolants and the resulting bit-level interpolant I_b are constructed as AIGs. We utilize structural hashing when constructing AIGs, across the bit-blasted AIG representation of the bit-vector abstraction and the partial interpolants generated during interpolant extraction.

Post-Processing. Bit-level interpolant I_b is represented as an AIG, which must be lifted back to the word-level in the final stage of our interpolation workflow. Intuitively, the easiest way to lift back I_b is the naive approach where its AIG structure is reconstructed on the T_{BV}-level from the input bits, which correspond to bits of uninterpreted T_{BV}-constants. This would, however, result in a complete loss of word-level structure, which may not be desirable for techniques that exploit the structure of the interpolant. An example of such a technique is IC3+IA [16], which extracts and filters predicates from the interpolant to be used as an abstraction refinement. And even for techniques that use the interpolant as-is, subsequent reasoning in the back-end SMT solver may exploit structure.

In the post-processing stage of our workflow, we therefore do not naively lift I_b back to the word-level, but try to recover as much word-level structure as possible. We describe our post-processing pipeline in detail in Section 4.5.

4.5 From Bit-Level to Word-Level Interpolants

Given a bit-level interpolant I_b, in the post-processing stage of our interpolation workflow we lift I_b back to the word-level while recovering and simplifying word-level structure. Our post-processing pipeline consists of several stages, some optional, which are processed in order as follows.

Gate Mapping. In the first stage, we construct a T_{BV}-representation of I_b, starting from the input bits, while mapping AND-gates back to bits in corresponding T_{BV}-terms when possible. This back-mapping is the main difference to naively reconstructing the AIG structure on the node level by means of only Boolean conjunction and negation on top of the input bits, as outlined above.

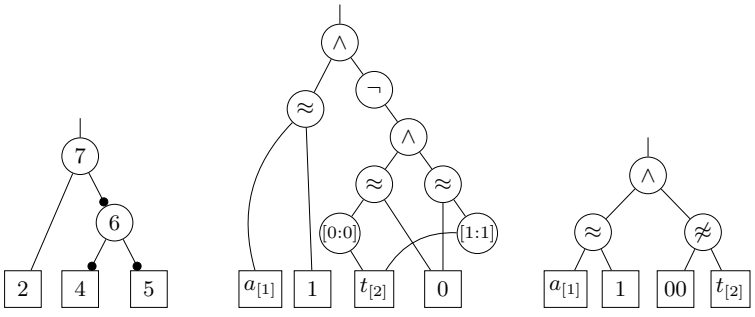
Gate mapping already recovers some word-level structure, albeit potentially still sliced into individual bits. Note that mapping AND-gates back to terms is

only possible for gates that represent "output" bits of the AIG circuit representation of a T_{BV}-term, with the additional constraint that the term only involves global symbols. AND-gates that map to bits of terms that involve local symbols may occur in I_b as a consequence of simplifications due to AIG-level rewriting when bit-blasting. Gates that cannot be mapped to bits of terms with only global symbols are reconstructed via Boolean conjunction and negation as expected.

As an example, consider bit-level interpolant I_b as given in Figure 4a. For this example, the original interpolation query is not relevant, it is only important to note that $\mathcal{S}(\varphi) = \{x, a, t\}$ with local symbol x (Boolean) and global bit-vector symbols $a_{[1]}$ and $t_{[2]}$. Interpolant I_b is an AIG with AND-gates $\{6, 7\}$ and constants $\{2, 4, 5\}$, labeled with the identifiers of the SAT variables in the CNF translation, which represent a, $t[1]$ and $t[0]$, respectively. Assume that AND-gate 6 maps to $t \approx 00$, and AND-gate 7 maps to the *msb* of term $ite(t \approx 00, ite(x, 00, 01), ite(a \approx 1, 10, 00))$. Figure 4b shows the word-level interpolant constructed from I_b via the naive construction, without gate mapping. When constructing the interpolant with gate mapping, gate 7 maps back to a term that involves local symbol x, thus we only map gate 6 to term $t \approx 00$. The resulting interpolant is given in Figure 4c.

Concretization. Recall that Bitwuzla's abstraction module may have replaced bit-vector arithmetic operations that occur in the original formula with fresh uninterpreted constants as abstractions. Each occurrence of such an abstraction in the interpolant must be concretized, i.e., replaced with the term it abstracts.

Simplification. In the simplification stage, we utilize Bitwuzla's preprocessor to simplify the concretized interpolant I_c. For this purpose, we create a fresh preprocessor instance and apply a pipeline of preprocessing passes to I_c in a predefined order until fixed-point. The passes configured in this pipeline are *term rewriting*, *and-flattening*, *term substitution*, *Boolean skeleton preprocessing*, *embedded constraints* and *arithmetic normalization* as described in [44]. The preprocessor may transform I_c into a set of simplified formulas $\{F_1, \ldots, F_n\}$,



(a) Interpolant I_b (AIG). (b) Naive lifting. (c) Gate mapping.

Fig. 4: Gate mapping example.

which yields the resulting simplified interpolant $\bigwedge_1^n F_i$. If I_c is of the form $\neg F$, we first preprocess F, and construct $\neg(\bigwedge_1^n F_i)$ as the simplified interpolant.

Gate Extraction. Bit-level interpolants extracted from the SAT proof as described above usually contain multiple bit-level equalities over consecutive bits of a bit-vector term. If such equalities occur within an n-ary AND-gate, they can be lifted to word-level equalities over bit ranges. The goal of the gate extraction stage is to recover as many word-level equalities over bit ranges as possible.

In a first step, we flatten binary AND-gates to n-ary AND-gates if flattening does not destroy sharing of common subexpressions. For example, $(x \,\&\, (y \,\&\, z))$ is flattened to $(x \,\&\, y \,\&\, z)$ only if $(y \,\&\, z)$ does not occur in other subexpressions of the interpolant. Next, since XNOR-gates correspond to bit-level equalities, for every n-ary AND-gate G in the interpolant I, we extract XOR-gates and XNOR-gates from the bit-level structure. Note that when we reach this stage, the AIG representation of the bit-level interpolant has already been lifted to the T_{BV}-level and simplified. Thus, to allow for a uniform gate extraction process, in this stage, we treat Booleans as bit-vector of size 1.

As an example, consider $(\sim(x[0] \oplus y[0]) \,\&\, \sim(x[1] \oplus y[1]) \,\&\, \sim(x[2] \oplus y[2]))$ over single bits of bit-vector constants x and y. This can be lifted to the bitwise comparison $\texttt{cmp}(x[2\!:\!0], y[2\!:\!0])$. Similarly, $(\sim x[0] \,\&\, \sim x[1] \,\&\, x[2] \,\&\, x[3])$ is lifted to $\texttt{cmp}(x[3\!:\!0], 1100)$. Note that bitwise comparison (as defined in SMT-LIB) is similar to bitwise equality, but defined over bit-vector sorts $\sigma_{[w]} \times \sigma_{[w]} \to \sigma_{[1]}$.

We extract bitwise comparison operations by applying the rewrite rules shown in Table 1 until fixed point. Rules (1)-(3) are the main rules for extracting XOR and XNOR gates, while rules (4-6) are responsible for extracting bit-level comparisons. Rule (7) combines bit-level comparisons to comparisons over ranges of consecutive bits via the concatenation operator $\circ$, and is only applied if it preserves sharing of common subexpressions. When no more rules can be applied, we create a new n-ary AND-gate G' from the (sorted) rewritten conjuncts $\mathcal{C}'$ and substitute G with G' in I. Note that Table 1 omits symmetric rewrite rules.

As a last step, we lift operations over bit-vectors of size 1 back to Boolean and translate bitwise comparison operations to word-level equalities, if applicable.

5 Evaluation

We evaluate our bit-vector interpolator as implemented in Bitwuzla in the context of the interpolation-based engines implemented in the SMT-based model checkers Pono [40] and Kind2 [14, 39]. Pono primarily targets hardware verification problems and implements four interpolation-based algorithms as engines *interp* [42], *ismc* [50], *ic3ia* [16], and *dar* [51]. We evaluate these engines on the benchmarks used in the bit-vector track of the hardware model checking competitions 2019 [1], 2020 [2], and 2024 [11]. After removing duplicates, this combined set contains 839 benchmarks in BTOR2 [45] format. Kind2 is a model checker for finite-state and infinite-state synchronous reactive systems, which also features an *ic3ia* engine. We evaluate Kind2 on a set of 786 Lustre models [30] for verifying integer arithmetic properties represented as bit-vectors of size 32.

	Condition	Extracted
(1)	$\sim(\sim x \,\&\, \sim y) \in \mathcal{C} \wedge \sim(x \,\&\, y) \in \mathcal{C}$	$x \oplus y$
(2)	$\sim(\sim x \,\&\, y) \in \mathcal{C} \wedge \sim(x \,\&\, \sim y) \in \mathcal{C}$	$\sim(x \oplus y)$
(3)	$\sim x \oplus y \in \mathcal{C}$	$\sim(x \oplus y)$
(4)	$x[i] \in \mathcal{C}$	$\texttt{cmp}(x[i], 1)$
(5)	$\sim x[i] \in \mathcal{C}$	$\texttt{cmp}(x[i], 0)$
(6)	$\sim(x[i] \oplus t) \in \mathcal{C}$	$\texttt{cmp}(x[i], t)$
(7)	$\texttt{cmp}(x[i\!:\!j], t_1) \in \mathcal{C} \wedge \texttt{cmp}(x[k\!:\!l], t_2) \in \mathcal{C} \wedge j = k + 1$	$\texttt{cmp}(x[i\!:\!l], t_1 \circ t_2)$

Table 1: Gate extraction rewrite rules applied on n-ary AND-gate with conjuncts $\mathcal{C}$. A rewrite rule applies if the conjuncts given in column Condition appear in $\mathcal{C}$. When applied, these conjuncts are replaced with the extracted expression.

We ran our experiments on a cluster of 48 compute nodes with AMD Ryzen 9 7950X CPUs and 128GB of RAM. For each model checker and benchmark pair, we allocated one CPU core and 16GB of memory with a time limit of 600 seconds. We implemented our approach on top of Bitwuzla commit 28c4d80 [3], as integrated in Pono on top of version v2.0.0-beta.1 [5] and in Kind2 on top of commit cce530b [4]. We compare our interpolation engine against MathSAT [17] version 5.6.12, which is the default interpolation engine for both tools.

We use a combination of three options to configure our interpolation engine and use + and − to indicate that an option is enabled and disabled, respectively. Option *map* configures the gate mapping stage of our bit-level interpolation pipeline, option *post* the simplification and gate extraction stage, and option *subst* the word-level equality substitution interpolation.

Preliminary experiments have shown that gate mapping has in general a positive impact on the evaluated interpolation-based algorithms. We thus only disable gate mapping for configuration *-map-post-subst*, which corresponds to producing *purely bit-level* interpolants. Additional experiments with Pono across all four interpolation engines also showed that on average, the size of interpolants produced by Pudlák's interpolation system is 2.9x on the bit-level and 3.7x after post-processing over using McMillan's system. In general, configurations using McMillan's system significantly outperform configurations using Pudlak's system. We thus use McMillan's system in the following and by default.

In our first experiment, we evaluate the performance of our interpolation engine against MathSAT in the context of Pono as shown in Figure 5. Overall, Pono with Bitwuzla as the interpolator significantly outperforms Pono with MathSAT as the interpolator in all configurations for all engines except the *ic3ia* engine. For the *ic3ia* engine, *Bzla:Bzla+post+subst* outperforms *Bzla:MSat*, but *MSat:MSat* performs best and better than any other *ic3ia* configuration. This could be due to Pono using a custom configuration of MathSAT as the solver specifically for *ic3ia* which was tuned for combination with MathSAT as the interpolator, whereas Bitwuzla is used in default configuration.

In the context of Kind2's *ic3ia* engine, using MathSAT as the interpolator clearly outperforms Bitwuzla as the interpolation engine, as shown in Figure 6.

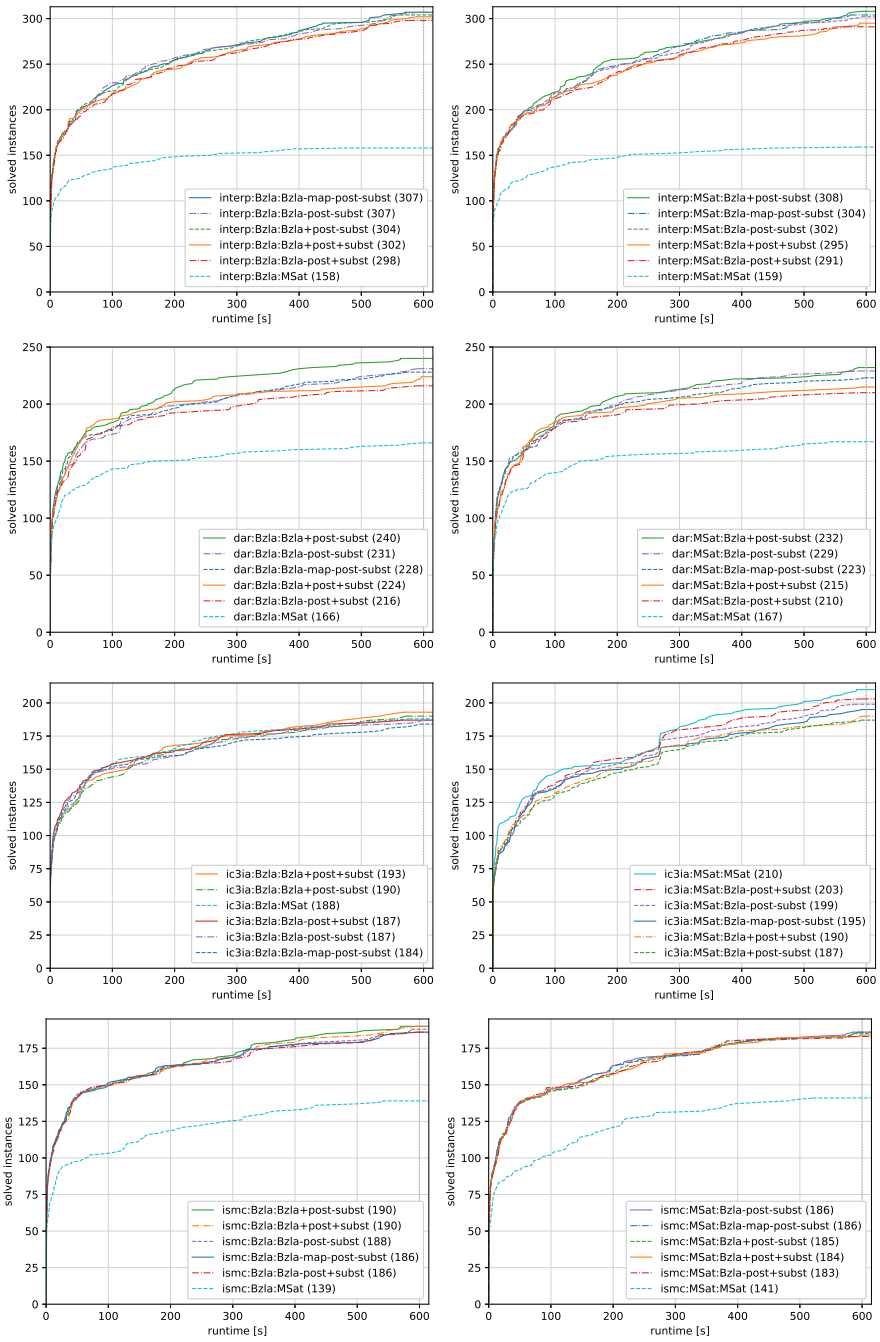


Fig. 5: Pono engines (E) with Bitwuzla (Bzla) and MathSAT (MSat) as solver (S) and interpolator (I) back ends. Configurations are given as E:S:I with Bitwuzla interpolator options gate mapping (map), post-processing (post), word-level substitution (subst) enabled (+) or disabled (-). Number of solved instances given in parentheses.

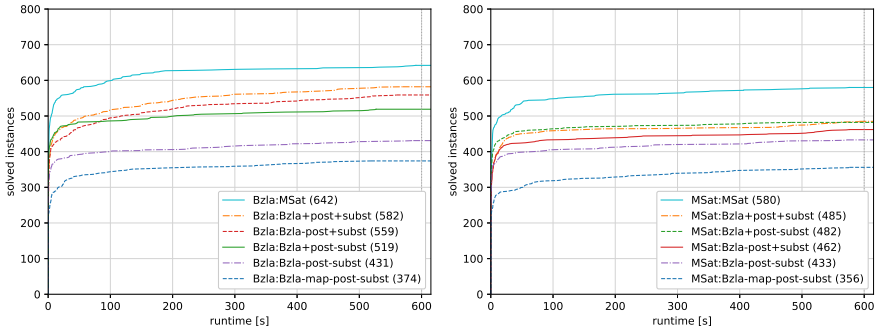


Fig. 6: Kind2 IC3IA engine with Bitwuzla (Bzla) and MathSAT (MSat) as solver (S) and interpolator (I) backends. Configurations given as S:I, with Bitwuzla interpolator options as in Figure 5. Number of solved instances given in parentheses.

The *ic3ia* algorithm heavily relies on extracting predicates from interpolants and can get overwhelmed if too many bit-level predicates are present. The Lustre benchmarks are bit-vector encodings of problems that were originally encoded in integer arithmetic and are thus arithmetic-heavy, with little bitwise reasoning. Hence, it is not too surprising that for *ic3ia*, the structure of these benchmarks seems to favor the arithmetic-based interpolation technique of MathSAT's layered interpolation approach, which produces predicates with potentially more word-level structure, especially in terms of bit-vector arithmetic. Further, MathSAT's technique also constructs bit-vector inequalities, which our approach does not recover if they cannot be mapped back in the gate mapping stage.

In general, we observe that different configurations of our interpolator, corresponding to varying degrees of how much word-level structure is reconstructed, perform best across different interpolation-based word-level model checking techniques. Term substitution-based interpolation, if successful, preserves the word-level structure of the original problem and is especially beneficial for the *ic3ia* engines. Interpolator configurations that extract more word-level structure generally perform better for *ic3ia* engines than configurations that produce interpolants with less structure, while bit-level interpolant configurations perform worse. Our post-processing pipeline is especially beneficial for engine *dar*, while for engine *interp*, less word-level structure seems to lead to better performance.

In our second experiment, we measure the impact of our simplification and gate extraction stages in our post-processing pipeline on the size of the produced interpolants. We measure the size of an interpolant in terms of the number of AND-gates. For this purpose, we analyzed the 371,592 interpolation queries issued by the interpolation engines of Pono in configuration *Bzla:Bzla+post-subst* on all 839 HWMCC benchmarks within the given time and memory limits. For each interpolation query, we computed the size of the initial bit-level interpolant, the size after the simplification stage, and the size after the gate extraction stage. After each stage, we compute the number of AND-gates in the bit-blasted AIG representation of the interpolant. Figure 7 (left) compares the interpolant size

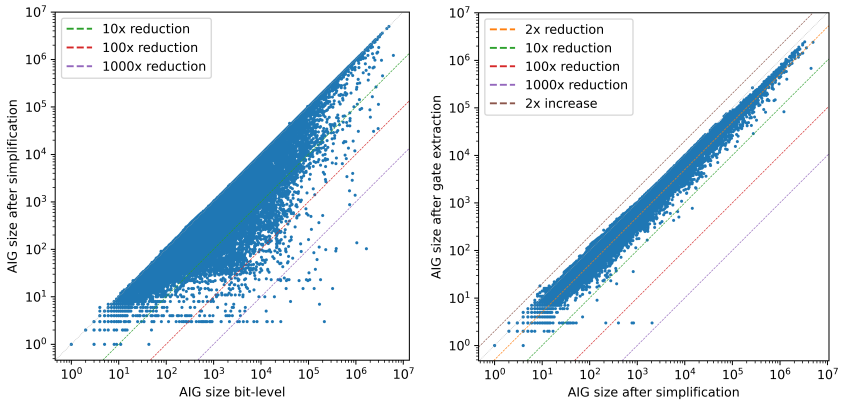


Fig. 7: Interpolant size comparison for our post-processing pipeline.

of the initial interpolant with the size of the interpolant after the simplification stage, and Figure 7 (right) compares the size of the interpolants after the simplification stage and after the gate extraction stage.

In the simplification stage, the preprocessor of Bitwuzla is able to consistently reduce the size of the interpolant, sometimes more than three orders of magnitude. The gate extraction stage further reduces the size of the simplified interpolant for 59.8% of the queries, up to two orders of magnitude.

We also measured the time spent in interpolant generation, not including the time for the satisfiability check that must precede the interpolation query. On average, 7.4% (6.3% total) of the total runtime of Pono is attributed to interpolant generation, including 2.0% (2.3% total) spent in our post-processing pipeline. For the 44,100 (out of 371,592) queries that were generated for solved benchmarks, interpolant generation is on average 7.8% (4.4% total), and our post-processing 0.9% (0.6% total) of the total runtime.

6 Conclusion

We have presented an extension of Bitwuzla with a new bit-vector interpolation engine, based on extracting interpolants from proofs produced by CaDiCaL as the back-end SAT engine of the bit-blasting bit-vector solver. Our interpolation engine supports the quantifier-free fragment of the theory of fixed-size bit-vectors. Interpolation for other theories supported by Bitwuzla, including arrays and uninterpreted functions, is currently limited to queries without occurrences of AB-mixed lemmas in the proof core. We leave extending our procedure to lift this limitation as future work.

Acknowledgement. We thank Po-Chun Chien and Áron Ricardo Perez-Lopez for their work on integrating Bitwuzla as an interpolator in Pono. We also thank Daniel Larraz for integrating Bitwuzla as interpolator in Kind2.

Data Availability Statement. The artifact accompanying this paper is archived and available in the Zenodo repository at https://zenodo.org/record/17427360.

References

1. Hardware Model Checking Competition 2019. https://fmv.jku.at/hwmcc19/ (2019)
2. Hardware Model Checking Competition 2020. https://hwmcc.github.io/2020/ (2020)
3. Bitwuzla on GitHub. https://github.com/bitwuzla/bitwuzla/ (2026)
4. Kind2 on GitHub. https://github.com/kind2-mc/kind2/ (2026)
5. Pono on GitHub. https://github.com/stanford-centaur/pono/ (2026)
6. Backeman, P., Rümmer, P., Zeljic, A.: Interpolating bit-vector formulas using uninterpreted predicates and presburger arithmetic. Formal Methods Syst. Des. **57**(2), 121–156 (2021). https://doi.org/10.1007/S10703-021-00372-6
7. Barbosa, H., Barrett, C.W., 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. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
8. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.7. Tech. rep., Department of Computer Science, The University of Iowa (2025), available at http://smt-lib.org
9. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
10. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: Cadical 2.0. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14681, pp. 133–152. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_7
11. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2024. In: Narodytska, N., Rümmer, P. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024. p. 1. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_6
12. Bonacina, M.P., Johansson, M.: Interpolation systems for ground proofs in automated deduction: a survey. J. Autom. Reason. **54**(4), 353–390 (2015). https://doi.org/10.1007/S10817-015-9325-5
13. Brummayer, R., Biere, A.: Local Two-Level And-Inverter Graph Minimization without Blowup. In: 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), Mikulov, Czechia, October 2006, Proceedings (2006)
14. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part

II. Lecture Notes in Computer Science, vol. 9780, pp. 510–517. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_29
15. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A.F., Parker, D. (eds.) Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7385, pp. 248–254. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19
16. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 46–61. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_4
17. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
18. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000). https://doi.org/10.1007/10722167_15
19. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
20. Craig, W.: Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957). https://doi.org/10.2307/2963594
21. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_14
22. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
23. Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
24. Enderton, H.B.: A mathematical introduction to logic. Academic Press (1972)
25. Gelder, A.V.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Math-

ematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008 (2008), http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf

26. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany. pp. 10886–10891. IEEE Computer Society (2003). https://doi.org/10.1109/DATE.2003.10008
27. Graham-Lengrand, S., Jovanovic, D., Dutertre, B.: Solving bitvectors with MCSAT: explanations from bits and pieces. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 103–121. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_7
28. Griggio, A.: Effective word-level interpolation for software verification. In: Bjesse, P., Slobodová, A. (eds.) International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011. pp. 28–36. FMCAD Inc. (2011), http://dl.acm.org/citation.cfm?id=2157662
29. Gurfinkel, A., Vizel, Y.: Druping for interpolates. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014. pp. 99–106. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987601
30. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with smt-based techniques. In: Cimatti, A., Jones, R.B. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008. pp. 1–9. IEEE (2008). https://doi.org/10.1109/FMCAD.2008.ECP.19
31. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
32. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: Opensmt2: An SMT solver for multi-core and cloud computing. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 547–553. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_35
33. John, A.K., Chakraborty, S.: A layered algorithm for quantifier elimination from linear modular constraints. Formal Methods Syst. Des. **49**(3), 272–323 (2016). https://doi.org/10.1007/S10703-016-0260-9
34. K., H.G.V., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020. pp. 107:1–107:9. IEEE (2020). https://doi.org/10.1145/3400302.3415708
35. Khouri, B., Vizel, Y.: Revisiting drup-based interpolants with cadical 2.0. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II. Lecture Notes in Computer Science, vol. 15697, pp. 88–107. Springer (2025). https://doi.org/10.1007/978-3-031-90653-4_5

36. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 17–34. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_2
37. Kroening, D., Weissenbacher, G.: Lifting propositional interpolants to the word-level. In: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings. pp. 85–89. IEEE Computer Society (2007). https://doi.org/10.1109/FAMCAD.2007.13
38. Kroening, D., Weissenbacher, G.: An interpolating decision procedure for transitive relations with uninterpreted functions. In: Namjoshi, K.S., Zeller, A., Ziv, A. (eds.) Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference, HVC 2009, Haifa, Israel, October 19-22, 2009, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6405, pp. 150–168. Springer (2009). https://doi.org/10.1007/978-3-642-19237-1_15
39. Larraz, D., Viswanathan, A., Tinelli, C., Laurent, M.: Beyond model checking of idealized lustre in Kind 2. Ada Lett. **42**(2), 40–44 (Apr 2023). https://doi.org/10.1145/3591335.3591338
40. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.W.: Pono: A flexible and extensible smt-based model checker. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 461–474. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_22
41. Manzano, M.: Introduction to many-sorted logic. In: Many-sorted logic and its applications, pp. 3–86. John Wiley & Sons, Inc., New York, NY, USA (1993)
42. McMillan, K.L.: Interpolation and sat-based model checking. In: Jr., W.A.H., Somenzi, F. (eds.) Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
43. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
44. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 3–17. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_1
45. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32

46. Niemetz, A., Preiner, M., Zohar, Y.: Scalable bit-blasting with abstractions. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14681, pp. 178–200. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_9
47. Okudono, T., King, A.: Mind the gap: Bit-vector interpolation recast over linear integer arithmetic. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 79–96. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_5
48. Pollitt, F., Fleury, M., Biere, A.: Faster LRAT checking than solving with cadical. In: Mahajan, M., Slivovsky, F. (eds.) 26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy. LIPIcs, vol. 271, pp. 21:1–21:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.SAT.2023.21
49. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. **62**(3), 981–998 (1997). https://doi.org/10.2307/2275583
50. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA. pp. 1–8. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351148
51. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 308–323. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_22
52. Vizel, Y., Gurfinkel, A., Malik, S.: Fast interpolating BMC. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 641–657. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_43