TBUDDY: A Proof-Generating BDD Package

Randal E. Bryant
Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Email: Randy.Bryant@cs.cmu.edu

***Abstract*—The TBUDDY library enables the construction and manipulation of reduced, ordered binary decision diagrams (BDDs). It extends the capabilities of the BUDDY BDD package to support *trusted* BDDs, where the generated BDDs are accompanied by proofs of their logical properties. These proofs are expressed in a standard clausal framework, for which a variety of proof checkers are available. Building on TBUDDY via its application-program interface (API) enables developers to implement automated reasoning tools that generate correctness proofs for their outcomes. In some cases, BDDs serve as the core reasoning mechanism for the tool, while in other cases they provide a bridge from the core reasoner to proof generation. A Boolean satisfiability (SAT) solver based on TBUDDY achieves polynomial scaling when generating unsatisfiability proofs for a number of problems that yield exponentially-sized proofs with standard solvers. It performs particularly well for formulas containing parity constraints, where it can employ Gaussian elimination to systematically simplify the constraints.**

I. INTRODUCTION

Proof generation has become a core requirement for Boolean satisfiability (SAT) solvers when they encounter an unsatisfiable problem. The SAT solver generates a detailed proof in a standard proof format. An independent proof checker can then affirm that the problem is indeed unsatisfiable, ruling out any false negative results due to a bug in the SAT solver's algorithms or implementation. Most modern solvers are based on conflict-driven clause-learning (CDCL) algorithms, and these can readily be extended to generate proofs in the *Deletion Resolution Asymmetric Tautology* (DRAT) proof framework [1], [2]. Like resolution proofs [3], a DRAT proof is a *clausal proof* consisting of a sequence of clauses, each of which preserves the satisfiability of the preceding clauses. An unsatisfiability proof starts with the clauses of the input formula and ends with an empty clause, indicating logical falsehood. The fact that this clause can be derived from the original formula proves that the original formula cannot be satisfied.

Although a number of SAT solvers based on Binary Decision Diagrams (BDDs) have been implemented over the years [4]–[8], most of these predated the era when proof generation became a priority. In 2006, Biere, Jussila, and Sinz demonstrated that the underlying logic behind standard BDD algorithms can be encoded as steps in an extended resolution framework [9], [10]. Extended resolution [11], [12] augments standard resolution by allowing proofs to introduce *extension variables*, serving as abbreviations for Boolean formulas over the input and other extension variables. This can yield proofs that are exponentially more compact than standard resolution proofs [13]. Biere, Jussila, and Sinz use this capability by introducing an extension variable for each BDD node generated. The logic for each recursive step of standard BDD operations, based on the Apply algorithm [14], can then be expressed with a short sequence of proof steps. TBUDDY builds on this work.

The DRAT framework also supports extension variables. Our solver PGBDD [15], [16] (for "proof-generating BDD") demonstrated that a BDD-based SAT solver can generate DRAT proofs of unsatisfiability by integrating proof generation into the BDD package. Our second solver PGPBS (for "proof-generating pseudo-Boolean solver") augments the SAT solver with a *pseudo-Boolean constraint* solver, enabling it to generate DRAT proofs of unsatisfiability for problems where the input formula, described in conjunctive normal form (CNF), encodes parity and cardinality constraints [17]. PGPBS relies on the constraint solver to detect that the formula is unsatisfiable. BDDs serve only as a mechanism to prove that 1) each of the extracted constraints is implied by the input formula, and 2) each step of the solver preserves satisfiability. These two solvers achieved polynomial scaling while generating unsatisfiability proofs for a number of challenging SAT problems.

The prototype solvers PGBDD and PGPBS demonstrated that BDDs can provide a useful framework for proof-generating automated reasoning tools, but their performance, in terms of both speed and capacity, was limited by their Python implementations. In this work, we describe TBUDDY, a high performance library for constructing and manipulating trusted BDDs. TBUDDY builds on BUDDY, a BDD package written by Jørn Lind-Nielsen while he was a PhD student at the Technical University of Denmark in the late 1990s [18]. It has subsequently been used and modified by a number of others, although the current version (2.4) has been unchanged on Sourceforge since 2014. BUDDY is written in C but has a C++ interface that provides more convenient memory management. These features were carried over to the implementation of TBUDDY.

Although there are a number of BDD packages available, we chose to implement our proof-generating library by extending BUDDY for several reasons:

- Multiple studies have shown that BUDDY generally performs as well as other BDD packages [19]–[21].
- BUDDY references nodes as integer indices into an array, rather than as pointers to a node data structure. As a result, it can manage BDDs with up to two billion (2^{31})

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_10

nodes using four-byte references, rather than the eight-byte pointers required for modern, 64-bit machines.
- BUDDY does *not* use complement pointers [22], [23] to denote Boolean negation. Although these can reduce BDD sizes and enable constant-time complementation, they would greatly complicate adding proof generation. Complement pointers rely on a symmetry between True and False that is not present in clausal representations.
- The BUDDY code is clear and concise. The complete package, prior to our modifications, consists of around 13,000 lines of code. By contrast, the core of the popular CUDD package [24] has over 72,000 lines of code. CUDD includes many features that are not relevant for this work but would requiring updating as the core data structures are changed.
- BUDDY supports dynamic variable ordering [25]. We do not use that feature directly, since it would be challenging to keep the proof information updated as variables are swapped in the BDD. However, it enables maintaining a distinction between the numbering of variables in the input file and the ordering of those variables within the BDD. We have found this capability vital for achieving good performance on some benchmarks.

This paper describes the design and implementation of TBUDDY, as well as TBSAT, a proof-generating SAT solver implemented using TBUDDY. It presents experimental results for several scalable benchmarks that are intractable for current CDCL solvers. A complete version of the code is available at https://github.com/rebryant/tbuddy-artifact.

II. Proof Generation with BDDs

Our immediate goal is to support the operations of a BDD-based SAT solver, generating one or more solutions when the formula is satisfiable and an unsatisfiability proof when it is not. Future uses of a proof-generating BDD package include a variety of automated reasoning tasks that would benefit from the assurances provided by checkable proofs of correctness.

A. Notation

Formulas are defined over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. The symbols u, v and w also denote Boolean variables, possibly with subscripts. The notation $\overline{u}$ denotes complement of variable u. A *literal* ℓ is either a variable or its complement. A clause C consists of a set of literals, and a formula ϕ consists of a set of clauses. We denote a clause as a disjunction of literals, enclosed in square brackets, e.g., $[\overline{u} \vee \overline{v} \vee w]$. A clause consisting of a single literal ℓ, denoted $[\ell]$, is a *unit clause*.

An assignment α is a mapping from the input variables X to the set $\{0, 1\}$, where 0 represents false, and 1 represents true. Assignment α is said to satisfy clause C if there is some literal $\ell \in C$ such that $\ell = x$ and $\alpha(x) = 1$, or $\ell = \overline{x}$ and $\alpha(x) = 0$. Assignment α satisfies formula ϕ if it satisfies every clause in ϕ. A formula ϕ is said to be satisfiable if it has a satisfying assignment and to be unsatisfiable if no satisfying assignment exists. A formula containing the empty clause $[\,]$ cannot be satisfied.

TABLE I
Defining Clauses for Extension Variable u Representing BDD node $\boldsymbol{u}$

Notation	Formula	Clausal Representation		
		Nonterm. child	Child is 1	Child is 0
$\mathsf{HD}(u)$	$x \rightarrow (u \rightarrow u_1)$	$[\overline{x} \vee \overline{u} \vee u_1]$	1	$[\overline{x} \vee \overline{u}]$
$\mathsf{LD}(u)$	$\overline{x} \rightarrow (u \rightarrow u_0)$	$[x \vee \overline{u} \vee u_0]$	1	$[x \vee \overline{u}]$
$\mathsf{HU}(u)$	$x \rightarrow (u_1 \rightarrow u)$	$[\overline{x} \vee \overline{u}_1 \vee u]$	$[\overline{x} \vee u]$	1
$\mathsf{LU}(u)$	$\overline{x} \rightarrow (u_0 \rightarrow u)$	$[x \vee \overline{u}_0 \vee u]$	$[x \vee u]$	1

A clausal proof consists of a sequence of clauses $C_1, C_2, \ldots, C_m, C_{m+1}, \ldots, C_t$ where the first m clauses are those of the input formula ϕ, while the subsequent clauses have the property that they preserve the satisfiability of the preceding clauses. That is, for all $m \leq i < t$, if the formula consisting of clauses $\{C_1, \ldots, C_i\}$ is satisfiable, then so is the formula $\{C_1, \ldots, C_i, C_{i+1}\}$. A proof of unsatisfiability has an empty clause as its final clause. The fact that this clause can be derived via a sequence of the steps from the input formula proves that the formula is unsatisfiable.

B. BDD Extension Variables and Defining Clauses

The BDD package maintains a directed acyclic graph consisting of a set of nodes, where each node $\boldsymbol{u}$ is either *terminal* or *nonterminal*. There are just two terminal nodes: $\boldsymbol{T}_0$, representing false, and $\boldsymbol{T}_1$, representing true. Nonterminal node $\boldsymbol{u}$ has an associated variable $\mathsf{Var}(\boldsymbol{u}) \in X$ as well as child nodes $\mathsf{Low}(\boldsymbol{u})$ and $\mathsf{High}(\boldsymbol{u})$. Each BDD node $\boldsymbol{u}$ represents a Boolean function, denoted $[\![\boldsymbol{u}]\!]$. Terminal nodes represent constant functions: $[\![\boldsymbol{T}_0]\!] = 0$, and $[\![\boldsymbol{T}_1]\!] = 1$. The function for nonterminal node $\boldsymbol{u}$ is defined recursively using the *ITE* operator (short for "if-then-else"), where $\mathit{ITE}(u, v, w) = (u \wedge v) \vee (\neg u \wedge w)$:

$$[\![\boldsymbol{u}]\!] \;=\; \mathit{ITE}\Big(\mathsf{Var}(\boldsymbol{u}), [\![\mathsf{High}(\boldsymbol{u})]\!], [\![\mathsf{Low}(\boldsymbol{u})]\!]\Big) \qquad (1)$$

The DRAT proof system supports an *extension* rule, similar to that of extended resolution [11], [12]. That is, the proof can define and reference *extension variables* serving as abbreviations for Boolean formulas over input variables and previous extension variables. Extension variable u encoding Boolean formula F is introduced by including a set of *defining clauses* in the proof encoding the formula $u \leftrightarrow F$. This capability is key to proof generation with BDDs, with an extension variable defined for every nonterminal node in the BDD.

An assignment α over the input variables can be uniquely extended to assign values to the extension variables. Extension variable u is assigned the value resulting from applying its defining formula F to the values assigned to the input and previous extension variables. For assignment α and extension variable u, we therefore have $\alpha(u) \in \{1, 0\}$.

As with the approach of Biere, Sinz, and Jussila [9], [10], each nonterminal BDD node has an associated extension variable. Nodes are denoted by boldface letters, possibly with subscripts, e.g., $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{v}_1$, while their corresponding extension

variables are denoted with a normal face, e.g., u, v, and v_1. The extension variables associated with the nonterminal nodes of the BDD provide the proof with a semantic definition of how BDDs encode Boolean functions according to Equation 1. More precisely, for nonterminal node $\boldsymbol{v}$, let $\mathsf{Ex}(\boldsymbol{v}) = v$ be the extension variable associated with the node. For the two terminal nodes, define $\mathsf{Ex}(\boldsymbol{T}_0) = 0$ and $\mathsf{Ex}(\boldsymbol{T}_1) = 1$. For nonterminal node $\boldsymbol{u}$, let $x = \mathsf{Var}(\boldsymbol{u})$, $u_1 = \mathsf{Ex}(\mathsf{High}(\boldsymbol{u}))$, and $u_0 = \mathsf{Ex}(\mathsf{Low}(\boldsymbol{u}))$. Then the defining clauses for u encode the formula $u \leftrightarrow \mathit{ITE}(x, u_1, u_0)$. These clauses are shown in Table I. As can be seen, when both children are nonterminal, there will be four clauses, each containing three literals. When one or more children are terminal nodes, some of the formulas for the defining clauses degenerate into tautologies (indicated by table entry 1.) These are not included among the defining clauses. Others have just two literals. For BDD node $\boldsymbol{u}$, we let $\mathsf{Def}(\boldsymbol{u})$ denote the set of defining clauses for all nodes in the subgraph with root $\boldsymbol{u}$.

Consider assignment α over the input variables extended to assign values to the extension variables. We will say that assignment α satisfies BDD root $\boldsymbol{u}$ with associated extension variable u if $\alpha(u) = 1$. This will occur precisely for those assignments where $[\![\boldsymbol{u}]\!]$, the Boolean function associated with $\boldsymbol{u}$, evaluates to 1.

C. RUP Proof Steps

Each logical inference for the subset of the DRAT proof system we use is based on an application of the *reverse unit propagation* (RUP) rule [26], [27]. RUP provides an easily checkable way to combine a linear sequence of resolution steps with subsumption. Let $C = [\ell_1 \vee \ell_2 \vee \cdots \vee \ell_p]$ be a clause to be proved and let clauses $D_1, D_2, \ldots, D_k$ be a sequence of supporting *antecedent* clauses occurring earlier in the proof. The RUP step proves that $\bigwedge_{1 \leq i \leq k} D_i \rightarrow C$ by showing that a combination of the antecedents plus the negation of C leads to a contradiction. The negation of C is the formula $\overline{\ell}_1 \wedge \overline{\ell}_2 \wedge \cdots \wedge \overline{\ell}_p$ having a CNF representation consisting of unit clauses $[\overline{\ell}_i]$ for $1 \leq i \leq p$. A RUP check processes the clauses of the antecedent in sequence, inferring additional unit clauses. In processing clause D_i, if all but one literal in the clause is the negation of one of the accumulated unit clauses, then we can add this literal to the accumulated set. The final step, with clause D_k, must cause a contradiction, i.e., all of its literals are falsified by the accumulated unit clauses.

D. The Trusted BDD API

The TBUDDY package supports the generation of *trusted* BDDs (TBDDs). These are ones that have been formally certified to be implied by the input formula. More precisely, for a trusted BDD with root node $\boldsymbol{u}$ and associated extension variable u, any assignment α to the input variables that satisfies the input formula must also assign 1 to u. This can be written as $\phi, \mathsf{Def}(\boldsymbol{u}) \models u$. This property is proved by generating a sequence of proof clauses leading to a proof of the *validating clause*, consisting of unit clause $[u]$. We use the notation $\dot{\boldsymbol{u}}$ to indicate that node $\boldsymbol{u}$ is trusted.

```
/* Generate TBDD from input clause */
tbdd tbdd_from_clause_id(int i);

/* Form conjunction of two TBDDs */
tbdd tbdd_and(tbdd u, tbdd v);

/* Upgrade BDD v to TBDD */
tbdd tbdd_validate(bdd v, tbdd u);

/* Generate proof of clause */
int tbdd_validate_clause(ilist lits, tbdd u);
```

Fig. 1. Trusted BDD API Function Prototypes

The TBUDDY API provides several procedures that enable the generation of TBDDs. Their prototypes are shown in Figure 1. In these, data types `bdd` and `tbdd` represent BDDs and TBDDs, respectively, as is described in Section III-A. Data type `ilist` is the API's representation of integer lists.

The `tbdd_from_clause_id` operation generates the BDD representation $\boldsymbol{u}_i$ of input clause C_i, as well as a proof of unit clause $[u_i]$. The BDD representation of a clause is a linear chain. The proof that $C_i, \mathsf{Def}(\boldsymbol{u}) \models u_i$ consists of a single RUP step, with C_i plus a subset of the defining clauses for the nodes in the chain as antecedents [10].

Given trusted BDDs $\dot{\boldsymbol{u}}$ and $\dot{\boldsymbol{v}}$, the `tbdd_and` operation first generates the BDD representation $\boldsymbol{w}$ of their conjunction. It also generates a proof that $u \wedge v \rightarrow w$, given by the clause $[\overline{u} \vee \overline{v} \vee w]$. It then uses a RUP step with this clause plus unit clauses $[u]$ and $[v]$ to prove the unit clause $[w]$, upgrading node $\boldsymbol{w}$ to $\dot{\boldsymbol{w}}$. As is described below, the BDD construction and the proof generation are performed by a version of the BDD APPLYAND operation that generates both a BDD node and a sequence of proof steps [15], [16].

The standard version of the APPLYAND procedure recursively traverses the nodes for the two arguments and generates intermediate result nodes [14]. It maintains an *operation table* of previously computed results to ensure polynomial complexity. Given arguments $\boldsymbol{u}$ and $\boldsymbol{v}$, it directly handles the cases where one argument is a terminal node. Failing this, it looks in the table with key $\langle \boldsymbol{u}, \boldsymbol{v}, \mathit{And} \rangle$ and returns any stored result. Otherwise, a set of recursive calls is required. The program chooses variable x as the least (in the BDD variable ordering) among variables $\mathsf{Var}(\boldsymbol{u})$ and $\mathsf{Var}(\boldsymbol{v})$ and splits into two cases, given by nodes $\boldsymbol{u}_1$ and $\boldsymbol{v}_1$, and nodes $\boldsymbol{u}_0$ and $\boldsymbol{v}_0$. It recursively computes nodes $\boldsymbol{w}_1$ and $\boldsymbol{w}_0$ as the conjunctions of $\boldsymbol{u}_1$ and $\boldsymbol{v}_1$, and of $\boldsymbol{u}_0$ and $\boldsymbol{v}_0$, respectively. When $\boldsymbol{w}_1 = \boldsymbol{w}_0$, this becomes the returned result $\boldsymbol{w}$. Otherwise node $\boldsymbol{w}$ is created having $\mathsf{Var}(\boldsymbol{w}) = x$, $\mathsf{High}(\boldsymbol{w}) = \boldsymbol{w}_1$, and $\mathsf{Low}(\boldsymbol{w}) = \boldsymbol{w}_0$. Before returning, an entry with key $\langle \boldsymbol{u}, \boldsymbol{v}, \mathit{And} \rangle$ and result $\boldsymbol{w}$ is added to the table.

The modified version of APPLYAND operation follows this recursive structure, such that a recursive call generating node $\boldsymbol{w}$ as the conjunction for nodes $\boldsymbol{u}$ and $\boldsymbol{v}$ also generates a proof of the clause $[\overline{u} \vee \overline{v} \vee w]$, i.e., that $u \wedge v \rightarrow w$. We refer to this proof step as the *justifying clause* for the operation. The recursive calls will have generated proofs of the clauses

$[\overline{u}_1 \vee \overline{v}_1 \vee w_1]$ and $[\overline{u}_0 \vee \overline{v}_0 \vee w_0]$. In general, the desired result can require two RUP steps. The first generates a proof of the intermediate result $x \rightarrow (u \wedge v \rightarrow w)$ given by clause $[\overline{x} \vee \overline{u} \vee \overline{v} \vee w]$ using as antecedents the defining clauses $\mathsf{HD}(\boldsymbol{u})$, $\mathsf{HD}(\boldsymbol{v})$, and $\mathsf{HU}(\boldsymbol{w})$, as well as the recursive result $[\overline{u}_1 \vee \overline{v}_1 \vee w_1]$. The second step proves the target clause using as antecedents the intermediate result, defining clauses $\mathsf{LD}(\boldsymbol{u})$, $\mathsf{LD}(\boldsymbol{v})$, and $\mathsf{LU}(\boldsymbol{w})$, and the recursive result $[\overline{u}_0 \vee \overline{v}_0 \vee w_0]$. For special cases, such as when some of the arguments are terminal nodes, only a subset of these antecedents is required. In some cases, the desired proof degenerates to a single proof step. The proof generation code in TBUDDY attempts to generate a single-step proof when one of the recursive results is a tautology. When this fails, or for the more general case, it generates a two-step proof. A built-in RUP checker determines which clauses to use as antecedents and can detect whether the proof succeeds or fails. The intermediate clause generated in a two-step proof can be deleted immediately after the second clause is added, and therefore there is a single justifying clause associated with each recursive operation.

Observe that to reuse results from the operation table, the program needs to reference its justifying clause. This requires augmenting the table entry with a field to hold an identifier for the justifying clause, as is discussed in Section III-A.

The `tbdd_validate` operation enables an ordinary BDD with root $\boldsymbol{v}$ to be upgraded to trusted node $\dot{\boldsymbol{v}}$ based on trusted node $\dot{\boldsymbol{u}}$. When called, the program first generates a proof of the implication $u \rightarrow v$, given by the clause $[\overline{u} \vee v]$. It then uses a RUP step with this clause plus unit clause $[u]$ to prove the unit clause $[v]$. The implication proof is generated by PROVEIMPLICATION [15], an operation that traverses the BDD and generates proof steps without adding any nodes. At each step on arguments $\boldsymbol{u}'$ and $\boldsymbol{v}'$, it generates a proof of the justifying clause $[\overline{u}' \vee v']$, i.e., that $u' \rightarrow v'$, using a simplified version of the proof structure used for the conjunction operation.

Some applications of TBDDs combine BDD and clausal reasoning, alternating between the two forms. The `tbdd_validate_clause` operation transfers the trust embodied in TBDD node $\dot{\boldsymbol{u}}$ to a clause C, generating a proof of $\mathsf{Def}(\boldsymbol{u}), u \models C$: This function requires TBUDDY to generate a sequence of proof steps, concluding with a RUP step with the specified clause. In some cases, the step can be performed directly by tracing a path in the BDD from $\boldsymbol{u}$ down to node $\boldsymbol{T}_0$ and listing some of the defining clauses along the way as antecedents. In cases where the path is not unique, the prover must first generate a BDD representation $\boldsymbol{v}$ of the clause, validate $\boldsymbol{v}$, and then trace the path from $\boldsymbol{v}$ to $\boldsymbol{T}_0$.

E. Proof File Format

There are several different file formats for encoding a DRAT proof, representing different trade-offs between the level of detail that must be supplied by the proof generator, versus the effort required to check the validity of the proof. With the *LRAT* format [28], each proof step must be accompanied by a *hint*. For a RUP step, the hint specifies the sequence of antecedent clauses. These proofs can be checked efficiently by the program LRAT-CHECK. There are also several formally verified checkers for LRAT proofs [28], [29]. By contrast, no hints are given with the *DRAT* format [2]. For each RUP step, the checker must identify a sequence of prior clauses that can serve as the antecedent. This format is accepted by the widely used DRAT-TRIM checker. Internally, DRAT-TRIM operates by adding the hints and then invoking an LRAT checker. The *FRAT* format [30] spans the two extremes of hints versus no hints by making the hints optional. It also operates by adding hints and invoking an LRAT checker. TBUDDY can generate proofs in any of these formats. Here we describe properties of the LRAT file format that influence how BUDDY encodes and stores proof information. Generating proofs in other formats requires storing additional information. For long executions, the proofs can range up to one billion clauses. These would be far too long for the DRAT-TRIM checker, due to the high cost of generating hints. In practice, therefore, it is best to either generate LRAT proofs or to generate FRAT proofs where the steps involving BDD operations include hints.

Following the conventions of the DIMACS format for encoding CNF formulas, the proof clauses for a formula with n variables and m clauses are encoded using signed integers to represent literals, where variable x_i is represented as the value i, and its complement as $-i$. Each clause in the proof is assigned a numeric *clause ID*, with the first m of these corresponding to the input clauses (which are not included in the proof file). Clause IDs must be in ascending order, but they need not be consecutive. Extension variables are represented by integers with values greater than n. RUP proof steps are encoded by giving the clause ID, the literals of the clause, and a list of the antecedent clause IDs. LRAT also supports *clause deletion*, where a list of clause IDs is provided, indicating that the proof will no longer use these clauses as antecedents. Deleting clauses whenever possible is critical for the proof checker, since it must retain copies of all *active* clauses, i.e., those that have been added but not yet deleted.

III. IMPLEMENTATION

With this as background, we can now describe how the BUDDY BDD package was modified to support proof generation. As we have seen, the key requirements are:

- Each time a new BDD node is created, it must be assigned an extension variable and its defining clauses must be added to the proof.
- For each input clause C_i, its BDD representation $\boldsymbol{u}_i$ must be generated, along with a proof of validating clause $[u_i]$.
- Every recursive step of the APPLYAND and PROVEIMPLICATION operations must generate one or two proof steps.
- The result nodes and proof steps generated by BDD operations must be stored for later reuse.
- A RUP step is required to prove validating clause $[u]$ when BDD root $\boldsymbol{u}$ is generated by conjunction or implication testing.

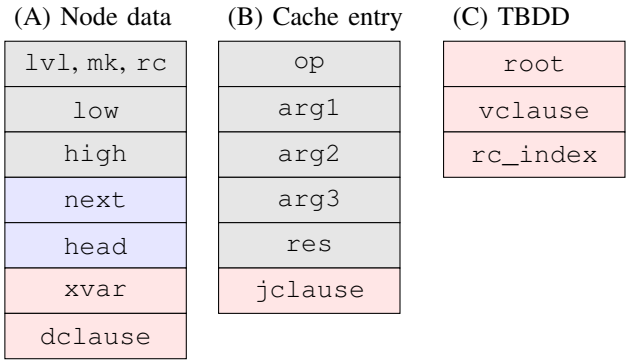

Fig. 2. Data structures for nodes (A) cache entries (B), and TBDDs (C). Each rectangle represents four bytes. Proof generation requires adding the fields shown in red.

- The defining clauses for the nodes and the clauses generated by RUP steps should be deleted when they are no longer required for subsequent proof steps.

These capabilities can all be incorporated into the basic BDD operations, as well as the supporting operations to manage the data structures.

A. Data Structures

Figure 2(A) and (B) show the fields in the two major data structures for BUDDY, with added fields (shown in red) to support proof generation. It also shows the representation for a TBDD (C). A BDD node in BUDDY is indicated by an integer, providing an index into an array of node structures, each having the fields shown in (A). Nodes $\boldsymbol{T}_0$ and $\boldsymbol{T}_1$ are represented by indices 0 and 1, respectively. Each rectangle in the figure represents four bytes. The node array integrates the set of BDD nodes with the *unique table*, providing a mapping from the children and variable for each node to the node itself. In the node data structure (A), the fields indicated in gray encode the node. Three values are packed into the first four-byte word: `lvl`, encoding the position of the node variable in the BDD variable ordering, `rc`, a reference count used to track external references to the node, and `mk`, a single bit used to support mark-sweep garbage collection. The indices for the two children `low` and `high` occupy the second and third words. The fields shown in blue encode the unique table, with the `next` field forming a link in the linked list implementing a hash table bucket, and the `head` field providing the head of the linked list for all nodes that hash to this index.

As mentioned earlier, to support dynamic variable ordering, BUDDY distinguishes between the level of a variable, giving its position in the BDD variable ordering, and the integer representation of the variable, with permutation vectors providing the mapping between these two. We use this feature to allow the BDD variable ordering to be independent of the numbering of variables in the input file.

Supporting proof generation requires adding two fields to the node data structure. The `xvar` field gives the associated extension variable, encoded as an integer having a value greater than the number of input variables n. When a node is created, the next four clause IDs are assigned to its defining clauses, even if only some subset of these is added to the proof. The `dclause` field stores the first of these—the remaining three can be computed as offsets from this field. In skipping some possible clause IDs, we add some sparseness to the ID space. Considering that we can only encode around two billion ($2^{31} - 1$) clause IDs, and proofs can routinely reach one billion clauses, this might seem wasteful. However, only a small fraction of the nodes in large BDDs will have terminal nodes as children, and so the vast majority of nodes will require the full complement of four defining clauses.

Like other BDD packages [22], BUDDY stores its table of previously computed results as a direct-mapped cache indexed by a hash of the operation and arguments.[1] Before performing the recursive steps of an APPLY operation, the table is first referenced to see if a suitable result has already been generated. When a new result is added to the table, any previous result that hashes to the same position is overwritten. The entries in the cache are shown in Figure 2(B). The standard entries (shown in gray) encode the operation, arguments (up to three), and the result node, each given as a four-byte integer. In the event the operation is either APPLYAND or PROVEIMPLICATION, reusing the cached result also requires the ID of the justifying clause. This is stored in the field `jclause`.

The added fields enable TBUDDY to track the clause IDs of the defining clauses for the active BDD nodes and the justifying clauses of the cache entries. Significantly, TBUDDY need not keep copies of the clauses themselves. When actual clauses are required to support proof generation, they can be recreated based on other information stored with the node or the cache entry.

We can see that the node data structure expands from 20 bytes to 28 in order to support proof generation. Cache entries require 24 bytes with or without proof generation, since an eight-byte field is used to store results for operations that return floating-point numbers. We configured the program to maintain a cache size that has 1/8 the number of entries as the node array. Therefore, adding proof generation required growing these two data structures from combined total of 23 bytes per node to 31 bytes per node, an increase of $1.35\times$. These are the only two data structures that grow in proportion to the number of BDD nodes.

Figure 2(C) shows the representation of a TBDD. It consists of three integers. The first identifies the root node and the second gives the clause identifier for the validating clause. The third field, labeled `rc_index`, supports reference counting of TBDDs. This count is distinct from the reference count for the root node, since there may be references to a BDD node that are independent of its use in a TBDD. The reference count for a TBDD tracks references to possible uses of the validating clause in proof generation. Once the count drops to zero, the clause can be deleted. Since TBDD structures

[1]The standard BUDDY package maintains seven separate caches to support different operations. We combined these into a single, unified cache.

are passed by value, they cannot hold actual reference counts. Instead, a separate table of reference counts is maintained, with the `rc_index` field providing an index into this table. In typical applications, fewer than 1% of the BDD nodes serve as TBDD roots, and so the space required by this table is negligible.

As can be seen, the modifications to support proof generation are fairly modest. In terms of code, the original BUDDY package contains 13,186 lines of source code. The TBUDDY package expands this to 18,030, with 1,061 lines added to existing files, 2,715 lines in new files to support proof generation and TBDDs, and 1,068 in new files to support parity reasoning. As noted above, the memory used increases by around $1.35\times$. The impact on runtime is more variable; we show experimental results in Section V.

B. BDD Management

BUDDY represents all of the nodes as a single array. This array starts with an initial allocation and is expanded as more nodes are added. Each expansion requires allocating a larger array, copying over existing nodes, and reconstructing the unique table and free list. Before expanding, it attempts to free existing nodes by performing garbage collection, reclaiming nodes that cannot be reached by any reference external to the data structure. Garbage collection is supported by 1) having each node store a reference count indicating the number of external references to the node, and 2) performing mark-sweep garbage collection to determine which nodes are unreachable. Nodes with nonzero reference counts provide the starting points of the marking phase. Both resizing the node array and performing garbage collection cause the entire cache to be flushed, with all entries marked as invalid. Garbage collection can occur at any point during the program operation, including in the middle of a series of recursive calls. To support this capability, a stack is maintained indicating intermediate nodes that may be required at future points in the outstanding calls. These nodes are also incorporated into the marking phase.

Garbage collection and cache flushing provide the means to manage the active clauses in a proof. That is, when a node is reclaimed during the sweep phase, its defining clauses are deleted. When a cache entry is evicted, either because it is overwritten or the cache is flushed, its justifying clause is deleted. To support the ability to perform garbage collection in the middle of a sequence of recursive calls, the deletion steps are not added to the proof directly. Rather, they are added to a list, which is cleared as the top-level of the recursion completes. As mentioned earlier, the validating clauses for TBDDs are managed via a separate set of reference counts. The C++ interfaces to the package automatically handle the reference counting for both BDDs and TBDDs.

IV. Capabilities Supported by TBUDDY

Building on the basic support for TBDDs, we have created several additional libraries and a BDD-based SAT solver. We describe these capabilities here and present some experimental results in Section V.

A. Parity Reasoning

Parity constraints arise in a variety of contexts, but they are not well handled by current CDCL solvers. A parity constraint is an equation of the form:

$$x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_k} \quad = \quad p \tag{2}$$

The variables in this constraint are a subset of the input variables, and the phase p is 1 for odd parity and 0 for even. Adding two parity constraints creates a new parity constraint. Gaussian or Gauss-Jordan elimination systematically adds constraints to yield a reduced set [31]. It can determine when the set of constraints cannot be satisfied. When the constraints are satisfiable, it can be used to derive a satisfying assignment.

Manipulating parity constraints is especially efficient for BDDs. The BDD representation of a constraint with k variables contains $2k+1$ nodes, independent of the BDD variable ordering. As we have demonstrated [17], a set of parity constraints encoded in CNF can be automatically extracted from an input formula, and BDD-based proofs of unsatisfiability can be generated using Gaussian elimination. The TBUDDY package provides the necessary support for the proof generation portion of this task.

Our constraint library represents a parity constraint as a list of integer variable IDs, a phase, and a TBDD giving the BDD representation of the constraint as well as the ID of a validating clause justifying that this constraint is implied by the input formula. An input constraint is converted into this representation by 1) forming the TBDD representations of the input clauses that encode it, 2) conjuncting them, and 3) using this TBDD to validate a BDD representation of the constraint. Each time constraints having TBDD representations $\dot{\boldsymbol{u}}$ and $\dot{\boldsymbol{v}}$ are summed to form a constraint with BDD representation $\mathbf{t}$, we use the conjunction operation to generate TBDD $\dot{\boldsymbol{w}}$ representing the conjunction of the constraints and validate the sum by calling `tbdd_validate(`$\mathbf{t}$`,` $\dot{\boldsymbol{w}}$`)`.

Applying Gaussian elimination requires first running a preprocessor to identify how the clauses encode parity constraints [17]. The program creates a *schedule* listing equations of the form of Equation 2 and identifying which clauses encode each of these. It also provides a list of the *internal* variables, i.e., those appearing only in parity constraints. Implicitly, all other variables are *external*. Gaussian elimination reduces the set of constraints to a smaller set over only the external variables. If the reduced set contains a constraint of the form $0 = 1$, then the original set cannot be satisfied. Otherwise, any solution of the reduced set can be expanded into a solution of the original set. In either case, the reduced constraints have TBDD representations and can therefore be used in proof generation.

Our Gaussian elimination routine attempts to preserve the sparseness found in typical parity constraint problems, where the number of variables in the constraints is far less than the total number of variables in the problem. Maintaining sparseness requires a successful strategy for *pivot selection*. Consider a set of parity constraints $P_1, P_2, \ldots, P_m$, each of the form of

Equation 2. Let the notation $x_j \in P_i$ indicate that constraint P_i contains variable x_j. Each elimination step requires selecting a pivot constraint P_s and a pivot variable $x_t \in P_s$. It then eliminates variable x_t from all other constraints P_i for which $x_t \in P_i$ by replacing P_i with the sum $P_i \oplus P_s$. Our routine uses a greedy pivot selection strategy attributed to Markowitz [32], [33]: Let c_s be the number of nonzero variables in constraint P_s and r_t be the number of constraints containing variable x_t. Then a constraint P_s and variable $x_t \in P_s$ are selected such that the cost function $(c_s - 1) \cdot (r_t - 1)$ is minimized. That cost is an upper bound on the net number of variables that will be added to the constraints when generating the sums $P_i \oplus P_s$.

B. The TBSAT SAT Solver

The TBSAT solver builds on the TBUDDY library. It can generate multiple solutions for satisfiable formulas and proofs of unsatisfiability for unsatisfiable formulas. It starts by reading the input clauses and forming their TBDD representations. The overall control flow is determined by the combination of an optional input schedule file and bucket elimination, expanding on the capabilities implemented in our prototype solvers PGBDD [15] and PGPBS [17]. The schedule file can serve two different roles. In one, it specifies a sequence of conjunction and existential quantification operations using a stack-based notation. This mode can be effective when the user has some problem-dependent strategy for solving a particular problem. In the other form, it identifies sets of clauses forming parity constraints. These constraints are converted into TBDDs and simplified using Gaussian elimination. In some cases, a TBDD with root node $\boldsymbol{T}_0$ will be generated while processing the schedule file. That indicates the formula is unsatisfiable and the proof of unsatisfiability will be complete. Otherwise, the TBDDs remaining, including those of unused input clauses, are processed using bucket elimination. When no schedule file is provided, all clauses are processed in this manner.

Bucket elimination [8], [9], [34] processes the TBDDs according to some ordering of the variables. Our implementation makes the simplifying assumption that buckets are ordered according to the BDD variable ordering, with bucket i associated with input variable x_i. Each TBDD is stored in a list (the "bucket") according to its root node variable. Buckets are processed from the least to the greatest. For bucket i, a conjunction of the TBDDs in the bucket is computed to yield TBDD $\dot{\boldsymbol{u}}_i$. A new BDD is computed as $\boldsymbol{v}_i = \mathsf{Low}(\boldsymbol{u}_i) \vee \mathsf{High}(\boldsymbol{u}_i)$, existentially quantifying x_i from $\boldsymbol{u}_i$. This BDD is validated using TBDD $\dot{\boldsymbol{u}}_i$, since any Boolean function f and variable x satisfies $f \rightarrow \exists x\, f$. The resulting TBDD $\dot{\boldsymbol{v}}_i$ is then placed in the bucket corresponding to its root node variable. This process continues until either 1) the TBDD $\dot{\boldsymbol{T}}_0$ is generated, or 2) all buckets are processed with the final step yielding $\boldsymbol{v}_n = \boldsymbol{T}_1$. In the former case, the formula is unsatisfiable and the unsatisfiability proof is complete. In the latter case, the formula is satisfiable and the next task is to generate one or more solutions.

To generate a solution, the solver starts with an empty assignment and works in reverse order, adding assignments to variables x_n through x_1. Let $\alpha_{n+1} = \emptyset$. For bucket i, it can assume that α_{i+1} satisfies $\boldsymbol{v}_i$, and we must assign a value to x_i. Let $\boldsymbol{u}_1 = \mathsf{High}(\boldsymbol{u}_i)$ and $\boldsymbol{u}_0 = \mathsf{Low}(\boldsymbol{u}_i)$. Assignment α must satisfy at least one of these. In the event that just $\boldsymbol{u}_1$ is satisfied, assign 1 to x_i. If just $\boldsymbol{u}_0$ is satisfied, then assign 0 to x_i. Otherwise, x_i can be assigned an arbitrary value. No further BDD generation is required to find a solution.

To generate a solution where some of the variables have been eliminated by Gaussian elimination, the solver first continues the elimination process to simplify the intermediate parity constraints via Gauss-Jordan elimination [31]. It uses BDD representations of these constraints to generate assignments for the internal variables. To generate multiple solutions, a new clause is created as the negation of the generated assignment, and the buckets are reprocessed in forward order. If this processing yields BDD node $\boldsymbol{T}_0$, then no further solutions exist. Otherwise, the bottom-up generation of an assignment will be guaranteed to find a new solution.

V. Experimental Evaluation

As a general purpose SAT solver, TBSAT is no match for state-of-the-art CDCL solvers. Among benchmarks used in recent SAT competitions, it succeeds only on the Tseitin-Grid parity constraint problems [35]. On the other hand, it handles classes of problems for which CDCL solvers fare poorly. BDD-based approaches can best complement CDCL, rather than compete with it.

Table II shows the performance of proof-generating SAT solvers on several scalable, unsatisfiable challenge problems. It compares different operating modes of TBSAT to KISSAT, a state-of-the-art CDCL solver [36]. It shows a progression of problem sizes, with the most difficult benchmark for one approach becoming the starting point for the next. All experiments were performed on a 3.2 GHz Apple M1 Max processor with 64 GB of memory and running the OS X operating system. The proofs were checked using DRAT-TRIM for the proofs generated by KISSAT and LRAT-CHECK for those generated by TBSAT. For LRAT proofs over 500 million clauses, we used a modified version of LRAT-CHECK that better exploits the sparseness in the proof structure that arises when a large fraction of the clauses is deleted. The column labeled "SAT Time" indicates the time (in seconds) taken by the solver, and the column labeled "Check Time" indicates the time taken by the checker. The column labeled "Proof Clauses" indicates the number of clauses in the generated proof. Entries marked "—" indicate a failure by the program to complete. The following benchmark problems were evaluated:

- *Mutilated chessboard*: Tile an $n \times n$ chessboard with dominos. Two opposite corners are removed from the chessboard, making the task impossible [37]. The problem size, in terms of the number of variables and clauses, scales as $O(n^2)$.
- *Pigeonhole*: Assign $n+1$ pigeons to n holes such that no hole contains more than one pigeon [38]. The at-most-one constraints are encoded using auxiliary variables [39]. The problem size scales as $O(n^2)$.

TABLE II
PERFORMANCE OF KISSAT AND TBSAT ON UNSATISFIABLE CHALLENGE PROBLEMS

Solver	Method	Problem Size	Variables	Clauses	SAT Time	Proof Clauses	Check Time
Mutilated Chessboard							
KISSAT	CDCL	16	476	1,592	358.7	12,621,694	618.5
KISSAT	CDCL	18	608	2,044	1314.9	38,083,824	1295.8
TBSAT	Column scan	18	608	2,044	0.1	111,163	0.1
TBSAT	Column scan	368	270,108	943,544	898.2	568,261,363	568.8
Pigeonhole							
KISSAT	CDCL	13	351	508	1116.1	66,263,560	2041.8
KISSAT	CDCL	14	406	589	6077.2	331,858,919	—
TBSAT	Column scan	14	406	589	0.1	92,687	0.1
TBSAT	Column scan	254	129,286	193,549	898.5	898,819,648	993.5
Chew-Heule parity formulas							
KISSAT	CDCL	40	114	304	334.3	29,133,644	594.2
KISSAT	CDCL	44	126	336	3103.6	227,489,490	8254.9
TBSAT	Bucket elim.	44	126	336	0.1	24,492	0.1
TBSAT	Bucket elim.	8,666	25,992	69,312	894.7	505,637,209	523.4
TBSAT	Gauss. elim.	8,666	25,992	69,312	4.6	5,066,914	5.2
TBSAT	Gauss. elim.	699,051	2,097,147	5,592,392	645.3	575,600,179	656.1
Urquhart-Li parity formulas							
KISSAT	CDCL	3	153	408	—	—	—
TBSAT	Bucket elim.	3	153	408	0.1	38,598	0.1
TBSAT	Bucket elim.	35	25,305	67,480	784.6	349,400,890	230.8
TBSAT	Gauss. elim.	35	25,305	67,480	3.8	4,232,657	4.3
TBSAT	Gauss. elim.	316	2,093,184	5,581,824	529.3	484,548,938	346.9

- *Chew-Heule*: Enforce both odd and even parity constraints on the n input variables. Each constraint is encoded linearly using $n-1$ auxilliary variables, with the second constraint using a random permutation of the variables [40]. The problem size scales as $O(n)$.
- *Urquhart-Li*: A parity constraint problem devised by Urquhart [41], defined over a bipartite graph with $2m^2$ nodes. The problem size scales as $O(m^2)$. We use the benchmark generator implemented by Li [42].

The formulas were evaluated for different values of the scaling parameter n or m. Runs of TBSAT were limited to 900 seconds—longer runs generally produced proofs that exceeded the capacity of the proof checker. KISSAT was allowed to run for up to 7200 seconds.

The limitations of CDCL solvers for these problems are clearly indicated by the results for KISSAT. It can only handle relatively small instances. We also found that allowing longer run times does not have a significant effect, due to the exponential scaling. For example, KISSAT completes the mutilated chessboard problem for $n = 16$ in 360 seconds, but once it reaches $n = 20$, the solver runs for over two hours without completing. Similarly, KISSAT completes the pigeonhole problem for $n = 12$ in just 42 seconds, but once it reaches $n = 14$, it requires nearly 1.7 hours and generates a proof that is too large for DRAT-TRIM to check. For the Chew-Heule formulas, KISSAT can only complete $n \leq 44$ within the 7200-second time limit. We ran KISSAT for over 16 hours on the smallest instance of the Urquhart-Li benchmark, having $m = 3$, but it did not complete. It is remarkable that a problem with just 153 variables and 408 clauses could be so challenging for CDCL solvers.

By contrast, TBSAT achieves polynomial scaling for all four benchmarks. In earlier work [15], we presented *column scanning* to efficiently generate unsatisfiability proofs of the mutilated chessboard and pigeonhole problems. This approach performs a sequence of conjunction and quantification steps to effectively sweep through the columns of the chessboard or the pigeons in the pigeonhole problem in a manner inspired by symbolic model checking. TBSAT can also apply column scanning, easily handling the limiting instances for KISSAT. It can scale to $n = 368$ for the mutilated chessboard problem and to $n = 254$ for the pigeonhole problem within the 900-second time limit. Even though the generated proofs are very large, they can be verified by the modified version of LRAT-CHECK. It remains to be seen whether column scanning can be made more general and with automatic generation of the schedule and variable order.

TBSAT can apply bucket elimination to the two parity problems with good effect. It can easily handle the limiting instances for KISSAT, and it scales to the Chew-Heule benchmark for $n \leq 8666$ and the Urquhart-Li benchmark for $m \leq 35$ within a 900-second time limit.

Perhaps the most striking results are those using Gaussian elimination. By exploiting the sparse structure of the formulas, TBSAT can solve very large instances of the Chew-Heule and Urquhart benchmarks quickly. The limiting factor for both of these problems is that BUDDY allocates only 21 bits for the `level` field in each BDD node (Figure 2(A)), limiting it to to a maximum of $2^{21} - 1$ (2,097,151) input variables. This prevents it from going beyond $n = 699{,}051$ for Chew-Heule and $m = 316$ for Urquhart, each having over two million input variables and five million clauses. Obtaining these results

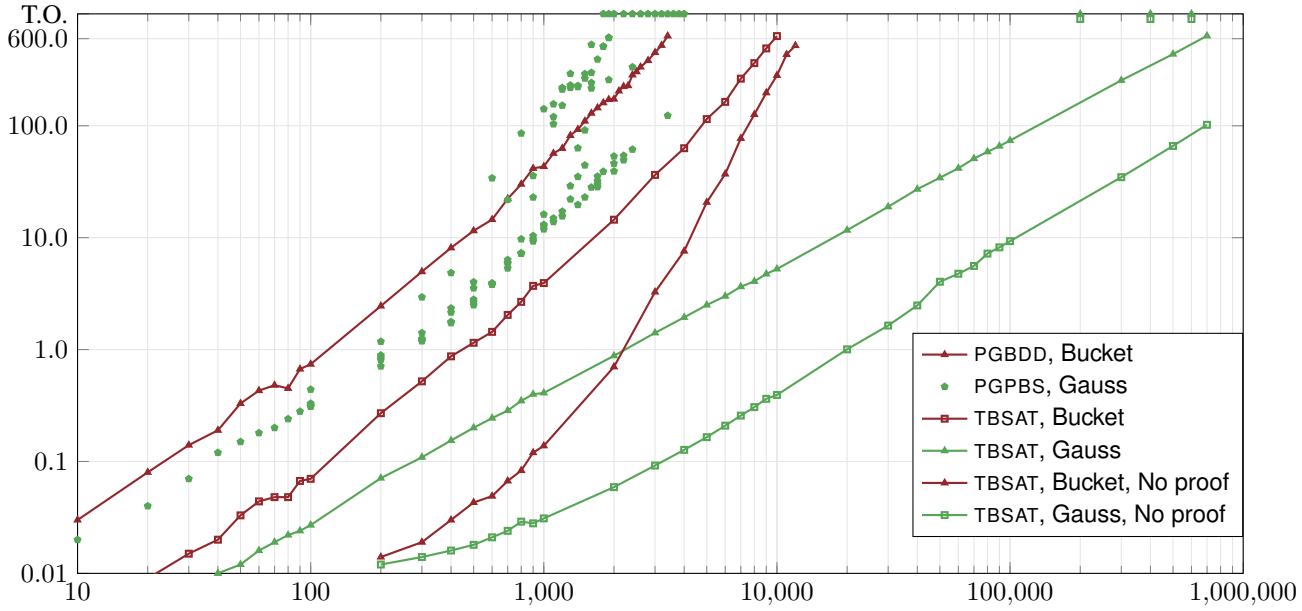

Fig. 3. Elapsed times (in seconds) for different solvers and solution methods on Chew-Heule parity formulas, as function of problem size n

requires no guidance for the user, and it is insensitive to the BDD variable ordering.

Figure 3 presents more runtime data for the Chew-Heule parity formula benchmark as a function of problem size n, enabling us to compare the relative performance and scaling of different solvers and solution methods. The red lines show three different versions of solving via bucket elimination. The top red line shows the performance of our prototype solver PGBDD, while the middle line shows the times for TBSAT. As can be seen, TBSAT consistently ran 10–12× faster. This can be attributed to the advantage of compiled C/C++ code versus interpreted Python. The lower red line shows the performance of TBSAT when proof generation is not required. This mode performs only the conjunction and quantification BDD operations, without generating proof clauses or writing them to a file. For smaller values of n, the runtime can be up to 33× faster, but this advantage drops to just a factor of 2× for larger values. For large values of n, the cost of garbage collection becomes a more dominant concern.

The data shown in green give results for three different versions of solving via Gaussian elimination. The data points at the top show the performance of our prototype pseudo-Boolean solver PGPBS. We found that the runtimes and generated proof sizes varied widely depending on the random permutation of the second parity constraint, and so the plot shows the raw data for five different random seeds for each value of n, including timeouts. The variation depends on whether or not the greedy pivot selections kept the constraints sparse. The middle green line shows the performance of TBSAT using Gaussian elimination. As noted before, it scales very well, nearly reaching its upper limit of $n = 699{,}051$ within the 600-second time limit. Compared to even the best data points for PGPBS, we see that TBSAT achieves much better scaling despite using very similar algorithms. However, like PGPBS, its ability to maintain sparseness depends on both the particular permutation of the second parity constraint, as well as the random tie breaking done during pivot selection. Consequently, some data points yielded timeouts. The lower green line shows the performance of TBSAT using Gaussian elimination, but without proof generation. In this mode, it need not perform any BDD operations and hence can be very fast, reaching a maximum of 15.3× faster for $n = 3{,}000$, but dropping off to 6.2× as n approaches its limiting value.

Overall, these measurements show that 1) TBSAT greatly outperforms the prototype implementations, 2) adding proof generation can slow performance considerably, but the penalty diminishes for larger benchmarks, 3) Gaussian elimination greatly increases the speed and capacity of the solver for parity constraint problems, and 4) careful pivot selection is required to maintain sparseness during Gaussian elimination.

VI. Conclusions and Acknowledgements

The TBUDDY library provides a powerful framework for creating automated reasoning tools that generate proofs of correctness. Building on an established BDD package, it can generate clausal proofs justifying the correctness of each step in its recursive algorithms. The TBSAT solver is especially strong for handling problems with parity constraints. We have also incorporated its proof-generation capability into a CDCL solver that uses Gauss-Jordan elimination for parity reasoning [43]. We anticipate implementing other automated reasoning tools using TBUDDY.

Thanks to Marijn Heule for his continued advice and for creating a high capacity version of LRAT-CHECK. This work was supported by the U. S. National Science Foundation under grant CCF-2108521.

References

[1] M. J. H. Heule, W. A. Hunt, Jr., and N. D. Wetzler, "Verifying refutations with extended resolution," in *Conference on Automated Deduction (CADE)*, ser. LNCS, vol. 7898, 2013, pp. 345–359.

[2] N. D. Wetzler, M. J. H. Heule, and W. A. Hunt Jr., "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 8561, 2014, pp. 422–429.

[3] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J.ACM*, vol. 12, no. 1, pp. 23–41, January 1965.

[4] R. Damiano and J. Kukula, "Checking satisfiability of a conjunction of BDDs," in *Design Automation Conference (DAC)*, June 2003, pp. 818–923.

[5] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet, "SBSAT: a state-based, BDD-based satisfiability solver," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 2919, 2004, pp. 398–410.

[6] J. Huang and A. Darwiche, "Toward good elimination orders for symbolic SAT solving," in *International Conference on Tools for Artificial Intelligence (ICTAI)*, 2004, pp. 566–573.

[7] H. Jin and F. Somenzi, "CirCUs: A hybrid satisfiability solver," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. Lecture Notes in Computer Science, vol. 3542, 2005, pp. 211–223.

[8] G. Pan and M. Y. Vardi, "Search vs. symbolic techniques in satisfiability solving," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 3542, 2005, pp. 235–250.

[9] T. Jussila, C. Sinz, and A. Biere, "Extended resolution proofs for symbolic SAT solving with quantification," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 4121, 2006, pp. 54–60.

[10] C. Sinz and A. Biere, "Extended resolution proofs for conjoining BDDs," in *Computer Science Symposium in Russia (CSR)*, ser. LNCS, vol. 3967, 2006, pp. 600–611.

[11] O. Kullmann, "On a generalization of extended resolution," *Discrete Applied Mathematics*, vol. 96-97, pp. 149–176, 1999.

[12] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Springer, 1983, pp. 466–483.

[13] S. A. Cook, "A short proof of the pigeon hole principle using extended resolution," *SIGACT News*, vol. 8, no. 4, pp. 28–32, Oct. 1976.

[14] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[15] R. E. Bryant and M. J. H. Heule, "Generating extended resolution proofs with a BDD-based SAT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I*, ser. LNCS, vol. 12651, 2021, pp. 76–93.

[16] ——, "Generating extended resolution proofs with a BDD-based SAT solver," *CoRR*, vol. abs/2105.00885, 2021.

[17] R. E. Bryant, A. Biere, and M. J. H. Heule, "Clausal proofs for pseudo-Boolean reasoning," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, 2022.

[18] J. Lind-Nielsen, *BuDDy: a Binary Decision Diagram Package*. Department of Information Technology, Technical University of Denmark, 1996.

[19] R. M. Jensen, "A comparison study between the CUDD and BuDDy OBDD package applied to AI-planning problems," Carnegie Mellon University, Tech. Rep. CMU-CS-02-173, September 2002.

[20] R. Pohl, K. Lauenroth, and K. Pohl, "A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models," in *International Conference on Automated Software Engineering (ASE)*, 2011, pp. 313–322.

[21] T. van Dijk, E. M. Hahn, D. N. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini, and L. Zhang, "A comparative study of BDD packages for probabilistic symbolic model checking," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, ser. LNCS, vol. 9409, 2015, pp. 35–51.

[22] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Design Automation Conference (DAC)*, June 1990, pp. 40–45.

[23] S.-I. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation," in *Design Automation Conference (DAC)*, June 1990, pp. 52–57.

[24] F. Somenzi, "Efficient manipulation of decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 171–181, 2001.

[25] R. L. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *International Conference on Computer-Aided Design (ICCAD)*, November 1993, pp. 139–144.

[26] E. I. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Design, Automation and Test in Europe (DATE)*, 2003, pp. 886–891.

[27] A. Van Gelder, "Producing and verifying extremely large propositional refutations," *Annals of Mathematics and Artificial Intelligence*, vol. 65, no. 4, pp. 329–372, 2012.

[28] M. J. H. Heule, W. A. Hunt, M. Kaufmann, and N. D. Wetzler, "Efficient, verified checking of propositional proofs," in *Interactive Theorem Proving*, ser. LNCS, vol. 10499, 2017, pp. 269–284.

[29] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "cake_lpr: Verified propagation redundancy checking in CakeML," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II*, ser. LNCS, vol. 12652, 2021, pp. 223–241.

[30] S. Baek, M. Carneiro, and M. J. H. Heule, "A flexible proof format for SAT solver-elaborator communication," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I*, ser. LNCS, vol. 12651, 2021, pp. 59–75.

[31] T. Laitinen, T. Junttila, and I. Niemelä, "Extending clause learning SAT solvers with complete parity reasoning," in *International Conference on Tools with Artificial Intelligence*, 2012, pp. 65–72.

[32] I. S. Duff and J. K. Reid, "A comparison of sparsity orderings for obtaining a pivotal sequence in Gaussian elimination," *IMA Journal of Applied Mathematics*, vol. 14, no. 3, pp. 281–291, 1974.

[33] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, no. 3, pp. 213–284, 1957.

[34] R. Dechter, "Bucket elimination: A unifying framework for reasoning," *Artificial Intelligence*, vol. 113, no. 1–2, pp. 41–85, 1999.

[35] J. Ellfers and J. Nordström, "Documentation of some combinatorial benchmarks," in *Proceedings of the SAT Competition 2016*, 2016.

[36] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020—Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[37] M. Alekhnovich, "Mutilated chessboard problem is exponentially hard for resolution," *Theoretical Computer Science*, vol. 310, no. 1-3, pp. 513–525, Jan. 2004.

[38] A. Haken, "The intractability of resolution," *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985.

[39] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," in *Principles and Practice of Constraint Programming (CP)*, ser. LNCS, vol. 3709, 2005, pp. 827–831.

[40] L. Chew and M. J. H. Heule, "Sorting parity encodings by reusing variables," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 12178, 2020, pp. 1–10.

[41] A. Urquhart, "Hard examples for resolution," *J.ACM*, vol. 34, no. 1, pp. 209–219, 1987.

[42] C.-M. Li, "Equivalent literal propagation in the DLL procedure," *Discrete Applied Mathematics*, vol. 130, no. 2, pp. 251–276, 2003.

[43] R. E. Bryant and M. Soos, "Proof generation for CDCL solvers using Gauss-Jordan elimination," 2022.