A Tale Of Two Solvers: Eager and Lazy Approaches to Bit-vectors[*]

Liana Hadarean[1], Kshitij Bansal[1], Dejan Jovanović[3],
Clark Barrett[1], and Cesare Tinelli[2]

[1] New York University
[2] The University of Iowa
[3] SRI International

Abstract. The standard method for deciding bit-vector constraints is via eager reduction to propositional logic. This is usually done after first applying powerful rewrite techniques. While often efficient in practice, this method does not scale on problems for which top-level rewrites cannot reduce the problem size sufficiently. A lazy solver can target such problems by doing many satisfiability checks, each of which only reasons about a small subset of the problem. In addition, the lazy approach enables a wide range of optimization techniques that are not available to the eager approach. In this paper we describe the architecture and features of our lazy solver (LBV). We provide a comparative analysis of the eager and lazy approaches, and show how they are complementary in terms of the types of problems they can efficiently solve. For this reason, we propose a portfolio approach that runs a lazy and eager solver in parallel. Our empirical evaluation shows that the lazy solver can solve problems none of the eager solvers can and that the portfolio solver outperforms other solvers both in terms of total number of problems solved and the time taken to solve them.

1 Introduction

Many software and hardware verification tasks require precise modeling of computer arithmetic and bit-level operations. The verification conditions coming from such applications can be expressed as satisfiability problems in the theory of fixed-width bit-vectors (T_{bv}). The standard technique for deciding the satisfiability in T_{bv} of a quantifier-free formula, vividly called *bit-blasting*, reduces the problem to a Boolean satisfiability (SAT) one, by replacing word-level operators with their bit-level circuit equivalents. Current state-of-the-art decision procedures for T_{bv} build on bit-blasting by applying powerful rewriting simplifications to the input formula before the final bit-blasting step. While often efficient in practice, this *eager* approach has several limitations: (i) the entire formula must be bit-blasted and solved at once, which may be difficult if the problem is too large; (ii) word-level structure and information can only be leveraged during preprocessing, not during solving; (iii) the complexity of the problem is a function of the bit-width; and (iv) eager solvers do not fit cleanly into theory combination frameworks.

[*] Work supported in part by the Semiconductor Research Corporation (SRC), tasks 1850.001 and 1850.002, and the National Science Foundation (grants 0644299, 1049495, and 1228768).

A *lazy* solver can address these limitations, explicitly targeting problems that are difficult for eager solvers and thus providing a complementary approach. The lazy approach for bit-vectors was first proposed in [8, 16]. In this paper, we revisit this approach, extending and improving it in several ways. Our lazy solver integrates algebraic, word-level reasoning with bit-blasting. Designed for easy plug-and-play combination with solvers for other theories, the procedure integrates an on-line lazy T_{bv} solver (LBV) into the DPLL(T) framework [20], separating theory-specific reasoning from the search over the Boolean structure of the input problem. This separation offers benefits orthogonal to those provided by eager bit-vector solvers but also poses interesting trade-offs. On one hand, it has the potential of incurring additional overhead and losing important connections between subproblems; on the other hand, depending on the Boolean structure of the problem, it often allows the T_{bv} solver to reason about much smaller problems at a time. We use a specialized decision heuristic to reduce the size of these sub-problems even further by considering only literals relevant to the current search context.

Our approach is particularly useful on problems whose subproblems fall into one of the efficiently decidable fragment of the bit-vector theory (e.g., the core theory of concatenation and extraction [11], the theory of bit-vector inequalities, or fragments decidable using equational reasoning). To target such problems, our LBV solver is built as the combination of several algebraic solvers specialized for some of these fragments together with a complete bit-blasting solver. The bit-blasting solver uses a dedicated SAT solver SAT_{bb}, distinct from the DPLL(T) Boolean engine driving the main search ($\text{SAT}_{\mathsf{main}}$). The separation of the two SAT engines fits cleanly into the DPLL(T) framework and allows the solvers to be tuned independently.

Experiments (described in Section 6) confirm our claim that the lazy approach is complementary to the eager approach, as the lazy solver efficiently solves problems that are either impossible or very difficult for eager solvers. At the same time, it is not realistic to expect the lazy solver to do well on problems that are easy for eager solvers (and indeed it is often slower on these problems). For this reason we propose a portfolio approach that runs an eager solver and a lazy solver in parallel. Additional experiments show that our portfolio solver outperforms eager solvers both in terms of the number of problems solved and the time taken to solve them.

The rest of the paper is organized as follows. Section 2 frames our contributions in terms of related work. Sections 3 and 4 provide technical preliminaries and a brief overview of the DPLL(T) framework. Section 5 describes the components of our lazy solver LBV including some optimizations enabled by the lazy framework. We present an experimental evaluation of the solver followed by an in-depth analysis in Section 6. Finally, we conclude with future work in Section 7.

2 Related work

The predominant approach to solving bit-vector constraints is via reduction to SAT. Boolector, a specialized solver for bit-vectors and arrays, and the winner of the 2012 SMT-COMP for QF_BV logic, employs preprocessing before encoding the bit-vector formula into the AIG format [7]. Z3, a DPLL(T)-style SMT solver, applies bit-blasting to all bit-vector operators, but has specialized equality reasoning [13]. STP2 does simplifications and solving for linear modular arithmetic, and then uses an eager encoding into CNF for bit-vector reasoning as well as an abstraction refinement loop for array axiom instantiation [18].

Some solvers encode the problem into a different domain such as (linear and non-linear) modular arithmetic [1]. These solvers are efficient at dealing with large data-paths and arithmetic operations. However some constructs, such as bit-wise operations, cannot be encoded and have to be bit-blasted away, while others, such as selection and concatenation, are very expensive for arithmetic solvers. Yet another approach [6, 11], based on word-level reasoning, uses Shostak-style canonizers and solvers to compute a canonical form for bit-vector expressions. However, it is limited to a restricted set of operators: concatenation, extraction and linear equations over bit-vectors.

The framework for a lazy bit-vector solver was first introduced by Bruttomesso et al. [8]. They describe an implementation of a DPLL(T)-style lazy layered solver for T_{bv} in the SMT solver MathSAT [10]. Their approach lazily encodes the problem into linear integer constraints and uses word-level inference rules during solving. Later work by Franzen [16] moves from encoding the problem into linear integer arithmetic to bit-blasting the formula to the main SAT solver instead.

Our work explores significant new ideas within this lazy framework, with the following contributions: (i) a dedicated SAT solver for T_{bv} that supports bit-blasting-based propagation with lazy explanations; (ii) specialized T_{bv} sub-solvers that reason about fragments of T_{bv}; (iii) inprocessing techniques to reduce the size of the bit-blasted formula when possible; and (iv) decision heuristics to minimize the number of literals sent to the bit-vector solver by the main SAT engine.

These new features greatly improve performance: our solver solves 450 more problems in roughly one third of the time compared to the only other lazy bit-vector solver. This brings the lazy framework from a niche player to a serious contender.

3 Formal Preliminaries

We assume familiarity with standard notions from many-sorted first-order logic. A *signature* Σ is a non-empty set of sort symbols together with a set of function symbols and a set of predicate symbols, each equipped with their respective arity and sorts. We call 0-arity function symbols *constants*.

A *constraint* is a conjunction of literals. We are concerned with the *constraint satisfiability* problem for a theory T with signature Σ_T, which consists of deciding whether a Σ_T-constraint is T-*satisfiable*, that is, satisfiable in a model of T. We will use $\models$ to denote propositional satisfiability and $vars(F)$ for the set of variables of a propositional formula F.

A bit-vector is a finite vector over the set $\{0, 1\}$ of binary digits. We consider the theory T_{bv} of bit-vectors with signature $\Sigma_{\mathsf{bv}} = \Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{con}} \cup \Sigma_{\mathsf{ineq}} \cup \Sigma_{\mathsf{ari}} \cup \Sigma_{\mathsf{bool}} \cup \Sigma_{\mathsf{shift}}$, consisting of infinitely many sort symbols $[n]$ with $n > 0$, and the function and predicate symbols listed in Table 1 together with their type (given after the symbol ::). Each sort $[n]$ denotes the set of bit-vectors of width n.

Table 1: T_{bv} signature Σ_{bv}

Σ_{eq}	sorts	$[n]$	$n > 0$	constants	$0, 1$	$::[1]$
	equal	$\_ = \_$	$::[n] \times [n]$		$\dots$	
Σ_{con}	concat	$\_ \circ \_$	$::[m] \times [n] \to [m+n]$	extract	$\_[i:j]$	$::[m] \to [i-j+1]$
Σ_{ineq}	less	$\_ < \_$	$::[n] \times [n]$	less-eq	$\_ \leq \_$	$::[n] \times [n]$
Σ_{ari}	plus	$\_ + \_$	$::[n] \times [n] \to [n]$	neg	$- \_$	$::[n] \to [n]$
	times	$\_ \times \_$	$::[n] \times [n] \to [n]$	div	$\_ / \_$	$::[n] \times [n] \to [n]$
	rem	$\_ \% \_$	$::[n] \times [n] \to [n]$			
Σ_{bool}	and	$\_ \& \_$	$::[n] \times [n] \to [n]$	or	$\_ \mid \_$	$::[n] \times [n] \to [n]$
	not	$\sim \_$	$::[n] \to [n]$	xor	$\_ \oplus \_$	$::[n] \times [n] \to [n]$
Σ_{shift}	left shift	$\_ << \_$	$::[n] \times [n] \to [n]$	right shift	$\_ >> \_$	$::[n] \times [n] \to [n]$

We will write $t_{[n]}$ for some fixed n to denote that t is a Σ_{bv}-term of sort $[n]$. Note that except for the constants, the function and predicate symbols in Table 1 are overloaded; for example, $+$ stands for any of the symbols in the infinite family $\{+ :: [n], [n] \to [n]\}_{n>0}$. For simplicity, we restrict our attention to a subset of the bit-vector operators described in the SMT-LIB v2.0 standard [4]; the missing ones can easily be expressed in terms of those given here.

The T_{bv}-satisfiability of conjunctions of equalities between terms over the *core* subsignature $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{con}}$ is decidable in polynomial time [9, 11]. However, adding almost any of the additional operators, or allowing for arbitrary Boolean structure, makes the T_{bv}-satisfiability problem NP-hard [6].

4 The DPLL(T) Framework

State-of-the-art SMT solvers efficiently decide the satisfiability of quantifier-free first-order formulas with respect to a background theory T by using the DPLL(T) framework [20]. The framework extends the Davis-Putnam-Logemann-Loveland (DPLL) decision procedure for SAT to handle reasoning in a theory T by relying on a *theory solver* (T-solver): a decision procedure for the T-satisfiability of Σ_T-constraints. Algorithm 1 gives a simplified algorithmic view of the DPLL(T) framework with a generalized theory interface. The algorithm takes as input a T-formula ψ and returns sat if ψ is T-satisfiable and $unsat$ otherwise. Variable C stores the set of working clauses and A the current truth assignment for C as a sequence of T-literals.We use $[]$ for the empty assignment and ; for the concatenation of two assignments. Initially, A is empty and C is simply the set of clauses obtained by converting ψ to Conjunctive Normal Form (CNF). We say that a pair $\langle \mathsf{A}, \mathsf{C} \rangle$ is *inconsistent* if the assignment A falsifies some clause in C; it is *consistent* otherwise. An assignment A *propositionally satisfies* ψ if ψ is satisfied by every full assignment extending A.

In Algorithm 1, the SAT and theory solver work together to augment A and C via `SatSolve` and `TheoryCheck`, respectively. The input to `SatSolve` is an assignment and a set of clauses $\langle \mathsf{A}, \mathsf{C} \rangle$. The return value is a new pair $\langle \mathsf{A}', \mathsf{C}' \rangle$ derived from the

```
Algorithm 1: DPLL($T$)
    Input: $\psi$ input formula
    $\mathsf{A} \leftarrow []$;
    $\mathsf{C} \leftarrow$ toCNF($\psi$);
    while true do
        $\langle \mathsf{A}, \mathsf{C} \rangle \leftarrow$ SatSolve($\mathsf{A}$, $\mathsf{C}$);
        if $\bot \in \mathsf{C}$ then
            return unsat;
        $\mathsf{final} \leftarrow$ Satisfies($\mathsf{A}$, $\psi$);
        $\langle \mathsf{P}, \mathsf{L} \rangle \leftarrow$ TheoryCheck($\mathsf{A}, \mathsf{final}$);
        if $\mathsf{L} = \emptyset$ and $\mathsf{final}$ then
            return sat;
        $\langle \mathsf{A}, \mathsf{C} \rangle \leftarrow \langle \mathsf{A}; \mathsf{P}, \mathsf{C} \cup \mathsf{L} \rangle$
```

input one such that either $\langle \mathsf{A}', \mathsf{C}' \rangle$ is consistent or $\bot \in C'$.[4] If the input pair $\langle \mathsf{A}, \mathsf{C} \rangle$ is consistent, `SatSolve` can extend A with *implied literals*, deduced by Boolean Constraint Propagation (BCP), or with one *decision literal*, chosen non-deterministically from the currently unassigned ones. On the other hand, suppose that $\langle \mathsf{A}, \mathsf{C} \rangle$ is inconsistent. If A contains no decision literals, then the search is complete (no satisfying assignment can be found) and `SatSolve` indicates this by extending C with the empty clause $\bot$. Otherwise, it resolves the conflict in $\langle \mathsf{A}, \mathsf{C} \rangle$ by doing CDCL-style conflict analysis [19], popping literals from A until $\langle \mathsf{A}, \mathsf{C} \rangle$ becomes consistent, and then adding at least one new implied literal.

The function `Satisfies` checks whether A propositionally satisfies the input formula ψ, setting final to true if so, and to false otherwise. An efficient implementation of `Satisfies` is described in Section 5.2.

The function `TheoryCheck` implements a T-solver and returns a sequence P of propagations and a set L of theory lemmas that are used to update $\langle \mathsf{A}, \mathsf{C} \rangle$ as follows:

1. If `TheoryCheck` finds A to be T-unsatisfiable it identifies a T-unsatisfiable subset $\{l_1, \ldots, l_n\}$ of literals in A and returns $\langle [], \{\neg l_1 \vee \cdots \vee \neg l_n\} \rangle$. Adding this clause to C forces `SatSolve` to backtrack and search for a different assignment.
2. If A is T-satisfiable, `TheoryCheck` computes a (possibly empty) sequence P of *theory-propagated* literals (unassigned literals in C that are T-entailed by A), returning $\langle \mathsf{P}, \emptyset \rangle$. P is added to A, which helps guide the SAT search in the right direction by avoiding unnecessary decisions.
3. `TheoryCheck` may not be able to efficiently determine the T-satisfiability of A as this may require reasoning by cases. `TheoryCheck` can request case splits by returning a set L of clauses encoding a T-valid formula. This effectively delegates the case splitting to the main Boolean engine.[5]

[4] `SatSolve`, which encapsulates the SAT solver, also manages the mapping between atoms and their propositional abstractions and vice versa.

[5] For `Satisfies` to work correctly, it is then necessary to update the current formula ψ to $\psi \wedge \bigwedge_{\varphi \in \mathsf{L}} \varphi$.

We say a call to `TheoryCheck` is *final* when the parameter final is set to true. Final calls to `TheoryCheck` must either ensure that A is T-satisfiable, or return one or more theory lemmas.

Two important aspects of theory solvers are not captured here. The first is that actual implementations of `TheoryCheck` are stateful: they store a copy of the assignment A internally and are instructed to push and pop literals from it as A is modified by the main loop. In practice, it is crucial that the theory solver be able to backtrack efficiently when A is shrunk, and reason incrementally when it is extended. The second aspect is that a theory solver must be able to provide an *explanation* for each theory-propagated literal p. This is a clause of the form $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ for some subset $\{l_1, \ldots, l_n\}$ of A, explaining why the literal was entailed. Explanations are needed by `SatSolve` during its conflict analysis. It is important for efficiency that the theory solver be able to compute explanations lazily, only as needed by `SatSolve`.

5 A Lazy Bit-vector Solver

We now proceed to give the details of our lazy bit-vector solver LBV, designed to fulfill the requirements of the `TheoryCheck` interface described above.

5.1 Subsolvers

The LBV solver consists of four sub-solvers: the equality solver $\mathsf{LBV_{eq}}$, the core solver $\mathsf{LBV_{core}}$, the inequality solver $\mathsf{LBV_{ineq}}$ and the bit-blasting solver $\mathsf{LBV_{bb}}$. Each subsolver is incremental and provides the theory solver functionalities described in Section 4. The architecture of LBV was designed to be modular and extensible: all the bit-vector reasoning is confined within the solver, and it is easy to enhance it by adding more sub-solvers.

Algorithm 2: `LBVCheck`

```
Input: $\langle \mathsf{A}, \mathsf{final} \rangle$
$\langle \mathsf{P_{eq}}, \mathsf{L_{eq}}, \mathsf{complete} \rangle \leftarrow \mathtt{LBVCheck_{eq}}\,(\mathsf{A}, \mathsf{final})$ ;
if complete then
    return $\langle \mathsf{P_{eq}}, \mathsf{L_{eq}} \rangle$ ;
$\langle \mathsf{P_{ineq}}, \mathsf{L_{ineq}}, \mathsf{complete} \rangle \leftarrow \mathtt{LBVCheck_{ineq}}\,(\mathsf{A}; \mathsf{P_{eq}}, \mathsf{final})$ ;
if complete then
    return $\langle \mathsf{P_{eq}}; \mathsf{P_{ineq}}, \mathsf{L_{eq}} \cup \mathsf{L_{ineq}} \rangle$ ;
$\langle \mathsf{P_{bb}}, \mathsf{L_{bb}} \rangle \leftarrow \mathtt{LBVCheck_{bb}}\,(\mathsf{A}; \mathsf{P_{eq}}; \mathsf{P_{ineq}}, \mathsf{final})$ ;
return $\langle \mathsf{P_{eq}}; \mathsf{P_{ineq}}; \mathsf{P_{bb}}, \mathsf{L_{eq}} \cup \mathsf{L_{ineq}} \cup \mathsf{L_{bb}} \rangle$
```

Algorithm 2 shows the implementation of `LBVCheck`, the `TheoryCheck` from Algorithm 1 corresponding to the LBV solver. `LBVCheck` calls the subsolvers in increasing order of computational cost. For each $i \in \{\mathsf{eq}, \mathsf{ineq}, \mathsf{bb}\}$, `LBVCheck`$_\mathtt{i}$ returns a sequence P_i of literals, a set L_i of clauses, and a Boolean value indicating whether the solver is complete or not. A solver i is *complete* if `LBVCheck_i` detects an inconsistency or if it determines that A is consistent (which it can only do if all the literals in A fall into the sub-solver's fragment of T_{bv}). If none of the solvers detect an inconsistency, `LBVCheck` returns the collection of all the propagated literals and lemmas generated by the individual sub-solvers.

The sub-solvers process all literals in A. However, except for $\mathsf{LBV}_{\mathsf{bb}}$, they reason on an abstraction of the literals. In particular, $\mathsf{LBV}_{\mathsf{eq}}$ treats all function and predicate symbols other than $=$ as uninterpreted, while $\mathsf{LBV}_{\mathsf{ineq}}$ (as well as $\mathsf{LBV}_{\mathsf{core}}$) treats as fresh variables any terms or predicates whose top symbol does not belong to its signature.

Equality Solver The equality solver $\mathsf{LBV}_{\mathsf{eq}}$, corresponding to `LBVCheck_eq`, uses a variant of well-known polynomial-time congruence-closure (CC) algorithms [14] to decide the satisfiability of constraints in Σ_{eq}. Standard CC algorithms assume that sorts have an unbounded cardinality. This makes them incomplete for reasoning about equality and disequality constraints in T_{bv}. For example, the formula $x_{[1]} \neq y_{[1]} \wedge x_{[1]} \neq z_{[1]} \wedge y_{[1]} \neq z_{[1]}$ is not satisfiable: there are only two distinct bit-vectors of width 1.

We handle the finite cardinality of the bit-vector sorts by trying to build a satisfying valuation for all the terms in a given Σ_{eq}-constraint. In final calls to check, once the CC algorithm is done and has not detected any inconsistency, $\mathsf{LBV}_{\mathsf{eq}}$ attempts to assign a distinct constant value to each congruence class $c^0_{[n]}, \ldots, c^k_{[n]}$ for each sort $[n]$ in the input problem. If this is not possible (because $k > 2^n$), it returns a lemma of the form:

$$\bigvee_{0 \leq i < j \leq 2^n} r^i_{[n]} = r^j_{[n]}$$

where r_i is a representative for class $c^i_{[n]}$, stating that at least two of the first $2^n + 1$ congruence classes must be merged.

This process continues until either the splits lead to an inconsistency or the sub-solver finds a satisfying valuation. The cardinality lemmas are currently generated only if the congruence classes consist just of bit-vector constants and variables (otherwise the solver reports that it is incomplete).

Core Solver The core solver is based on the slicing algorithms presented in [9, 11]. It decides conjunctions of equalities over $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{con}}$ in polynomial time, by reducing the problem to just equality reasoning. The key idea of the algorithm is expressing each variable as a concatenation of disjoint slices. The coarsest such decomposition that guarantees that none of the slices overlap given the input set of equalities, is called the *coarsest base*. In our experience, the core solver is most efficient on problems involving only core theory terms, and thus it is heuristically turned on for such instances.

Inequality Solver The inequality solver $\mathsf{LBV}_{\mathsf{ineq}}$ can decide the satisfiability of $(\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{ineq}})$-constraints by using an incremental special-purpose algorithm.[6] $\mathsf{LBV}_{\mathsf{ineq}}$ only

[6] This problem is a special case of modular difference logic that can be reduced to integer difference logic, as there is no wrap-around behavior due to overflows.

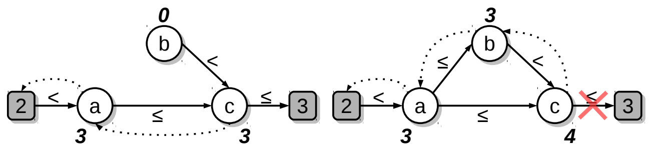
needs to reason about $<$ and $\leq$ since equality can be expressed in terms of $\leq$, and disequalities can be reasoned about by requesting a splitting lemma. For the rest of this section we assume all inequalities are unsigned (the signed case is analogous). We will use $\lhd$ to denote either $<$ or $\leq$, and $\lhd^*$ for the transitive closure of $\lhd$. A *valuation* M is a mapping from bit-vector variables $v_{[n]}$ to constant values in $[n]$. For convenience, we extend M to map constants to themselves, and to map other bit-vector terms and formulas to the constants obtained by mapping their sub-expressions and simplifying. A valuation M *satisfies* a bit-vector constraint ϕ if $\mathsf{M}(\phi) = true$.

Definition 1. Let I be a conjunction of inequality constraints over variables and constants of the same sort $[n]$. A valuation M is the *least model* of I if M satisfies I and for all valuations M' satisfying I and all terms t in I, $\mathsf{M}(t) \leq \mathsf{M}'(t)$.

It can be shown that every such satisfiable constraint I has a least model. Given I, $\mathsf{LBV_{ineq}}$ builds the least model by incrementally processing the inequalities. We will use $\mathcal{I}$ to refer to the already processed inequalities, and define the starting model M as:

$$\mathsf{M}(t_{[n]}) := \begin{cases} t_{[n]} & \text{if } t_{[n]} \text{ is a constant,} \\ 0_{[n]} & \text{otherwise} \end{cases}$$

where $0_{[n]}$ is the binary representation of 0 in n bits. We maintain the invariant that M is the least model of $\mathcal{I}$. Given a new inequality $a \lhd b$, we want to extend M to a least model of $\mathcal{I} \cup \{a \lhd b\}$, or discover that the problem is unsatisfiable. If $\mathsf{M}(a) \lhd \mathsf{M}(b)$ already holds, we are done. Otherwise, the least model property guarantees that terms a and b have the least possible values. Therefore, to satisfy $a \lhd b$ we must increase b's value, if possible, to match that of a. The update cannot violate previously satisfied inequalities of the form $\{t_1 \lhd t_2 \mid t_2 \lhd^* b\}$. The only terms whose values may need to be updated further are terms t such that $b \lhd^* t$. We reach a conflict when: (i) we try to update the model value of a constant, (ii) increasing the model value leads to an overflow or (iii) we detect an inequality cycle. The algorithm can be efficiently implemented using a priority queue that prioritizes updating the value of terms with lower model values.



(a) (b)

Fig. 1: The nodes are bit-vector terms; gray nodes are constants and white ones variables. Each node has an associated constant, its M value. The continuous edges represent inequalities. The dotted edges are *reason* edges: they point to the node that forced the last update to the current node's value.

Example 1. Consider the following set of inequalities over bit-vector terms of bit-width 8 where, for brevity, we use decimal numerals to denote bit-vector constants: $\mathcal{I} = \{2 < a, a \leq c, b < c, c \leq 3\}$. Figure 1a shows the least satisfying model for $\mathcal{I}$. To process the new inequality $a \leq b$, we add the corresponding inequality edge, and update the value of b to $\mathsf{M}(a)$. This in turn requires increasing the value of c to $\mathsf{M}(b) + 1$. We identify a conflict while revisiting $c \leq 3$: 3 is a constant and $M(c) \leq 3$ does not hold (Figure 1b). Because c has the lowest possible value, $\mathcal{I}$ must be unsatisfiable. We build the following conflict by including $c \leq 3$ and traversing back along the constraints that force the value of c to be 4: $\{2 < a, a \leq b, b < c, c \leq 3\}$.

Bit-blasting Solver Finally, the bit-blasting solver $\mathsf{LBV}_{\mathsf{bb}}$ can decide the satisfiability of bit-vector constraints over the entire Σ_{bv} signature. At its heart is a second SAT solver SAT$_{\mathsf{bb}}$ distinct from the DPLL(T) Boolean engine. Our implementation uses the open source MiniSAT solver [15]. We instrumented MiniSAT to efficiently implement the main requirements on a T-solver: incrementality, conflict detection and propagation of entailed literals.

Incrementality. Most SAT solvers do not have full support for incremental solving.[7] Incrementality can be simulated through a feature known as *solve with assumptions* [15]: given a fixed set C of input clauses, the SAT solver can check their satisfiability with respect to the assumption that some of the variables appearing in C are assigned to be true or false. We exploit this feature by creating a *marker variable* a_{BB} for each atom a in the formula being checked. When a appears in an assertion, instead of bit-blasting a, we bit-blast $a_{BB} \Leftrightarrow a$. We can then call solve with assumptions with the set of literals $A_{BB} := \{a_{BB} \mid a \in \mathsf{A}\} \cup \{\neg a_{BB} \mid \neg a \in \mathsf{A}\}$.

Conflict generation. If A is unsatisfiable, we use SAT$_{\mathsf{bb}}$ to determine an inconsistent subset of A_{BB} via resolution and return the corresponding subset of A as a conflict.

Propagation. On a non-final call to `LBVCheck`$_{\mathsf{bb}}$, we want to be able to determine whether any theory literals can be propagated without doing a full SAT check. To do this, we again use solve with assumptions but only allow the SAT solver to do Boolean Constraint Propagation (BCP), stopping it before any decisions are made. If BCP succeeds in deducing the value of a marker variable a_{BB}, the corresponding atom a can be propagated to have the same value that BCP assigned to a_{BB}. The explanation for the propagation can be computed using the SAT solver's conflict resolution infrastructure. As mentioned in Section 4, it is important to compute propagation explanations lazily as not all propagated literals may need to be explained. Unfortunately, the interaction between the SAT solver's solve-with-assumptions feature and non-chronological backtracking can cause the solver to lose the explanation for a propagated literal. To overcome this problem, we implemented a simple check that detects when backtracking can lead to the loss of explanations, and in such cases backtrack to a more conservative level instead. Algorithm 3 shows the implementation of `LBVCheck`$_{\mathsf{bb}}$. `BvSatBCP` implements the call to SAT$_{\mathsf{bb}}$ limited to BCP, while `BvSatSolve` is a normal full call to SAT$_{\mathsf{bb}}$.

[7] More input clauses can be added during solving, but the main challenge of removing problem clauses remains.

Algorithm 3: LBVCheck$_{\mathtt{bb}}$

```
Input: $\langle \mathsf{A}, \mathsf{final} \rangle$
$\langle \mathsf{P}, \mathsf{L} \rangle \leftarrow$ BvSatBCP($\mathsf{A}$) ;
if final and $\mathsf{L} = \emptyset$ then
    $\mathsf{L} \leftarrow$ BvSatSolve($\mathsf{A}$) ;
return $\langle \mathsf{P}, \mathsf{L} \rangle$ ;
```

5.2 Lazy Techniques

The lazy DPLL(T) framework enables several techniques that are difficult or impossible to use with eager solvers. In this section we discuss two of these techniques: applying word-level rewrites during solving (*inprocessing*) and reducing the problem size by only reasoning about atoms relevant in the current search context (*relevancy-based decision heuristics*).

Inprocessing Techniques Before engaging in potentially expensive SAT reasoning, $\mathsf{LBV}_{\mathsf{bb}}$ relies on the inprocessing module to check if the problem can be solved or significantly simplified by word-level simplification techniques. This is done by a process, described in Algorithm 4, that has the flavor of Gaussian elimination. It works by iterating over a worklist of theory literals W while maintaining a substitution map σ.

Initially, W is initialized to the set of literals A assigned to true in the current search context. For each worklist assertion $w \in W$, we first apply the substitution map, and then rewrite it using word-level simplification techniques (`Simplify`). The `SolveEq` procedure then attempts to solve the updated assertion w to obtain a new substitution. Alternatively, it can also learn new equalities entailed by w and add these to the working list.[8] The working list W and the substitution map σ are updated with this new information, and the process is repeated to a fixpoint.[9]

If any of the assertions in W reduces to false, we have a conflict. If there are no such obvious inconsistencies we can run the LBVCheck$_{\mathtt{bb}}$ routine on the simplified set of assertions W. We do this heuristically, if the problem has been reduced enough in terms of the circuit size. We found checking the simplified assertions when they are less than 50% of the size of the original assertions to be a good heuristic.

Relevancy-Aware Decision Heuristics The idea of *relevancy* is best understood with a simple example. Let $\psi = \neg a \wedge (b \vee \varphi)$ with assignment $\mathsf{A} = [\neg a; b]$. Note that A propositionally satisfies ψ regardless of how many unassigned literals are in φ. The literals in φ are *irrelevant*.

The DPLL(T) framework makes it easy to add a decision heuristic that avoids splitting on irrelevant literals. In particular, we can (i) detect when an assignment A becomes propositionally satisfying and *stop early* in order to reduce the number of literals

[8] In our implementation, we solve xor equations and slice equations between concatenation expressions to get new equalities.

[9] The data-structures are enhanced with extra book-keeping information to keep track of explanations. We omit these details for simplicity.

Algorithm 4: IN-PROCESSING

```
Input: $\mathsf{A}$
$\langle \mathsf{W}, \sigma \rangle \leftarrow \langle \mathsf{A}, [] \rangle$;
$\mathsf{changed} \leftarrow$ true;
while $\mathsf{changed}$ do
    $\mathsf{changed} \leftarrow$ false;
    for $\mathsf{w} \in \mathsf{W}$ do
        $\mathsf{w} \leftarrow$ Simplify($\sigma(\mathsf{w})$) ;
        $\langle \mathsf{W}', \sigma' \rangle \leftarrow$ SolveEq($\mathsf{w}$);
        if $\mathsf{W}' \neq \emptyset$ or $\sigma \neq []$ then
            $\mathsf{changed} \leftarrow$ true ;
        $\langle \mathsf{W}, \sigma \rangle \leftarrow \langle \mathsf{W} \cup \mathsf{W}', \sigma; \sigma' \rangle$;
if false $\in \mathsf{W}$ then
    return Conflict;
return BvSatSolve($\mathsf{W}$);
```

sent to theory solvers and (ii) employ *decision heuristics* that allow the SAT solver to decide only on literals relevant in the current search context. We use circuit-based techniques of maintaining *justification frontiers* [2, 17] to track which literals are relevant in each context.[10] The call to `Satisfies` in Algorithm 1 examines the Boolean structure of ψ and determines whether the current assignment A is sufficient to propositionally satisfy it. It does so by incrementally computing the *justification frontier* as the assignment A changes.

This heuristic, which we also call the *justification* heuristic, has a significant performance impact on bit-vector benchmarks, as shown in Section 6.

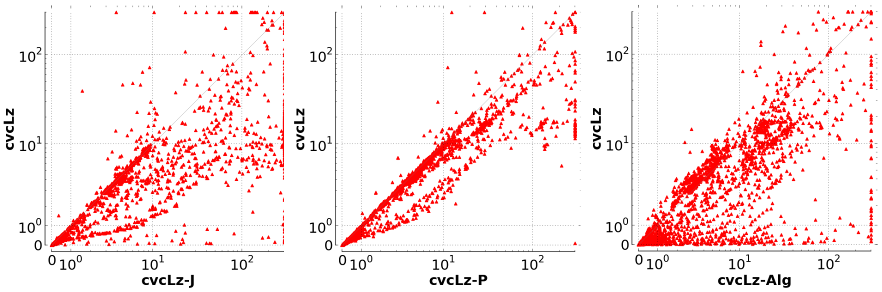
6 Experimental Results

In this section we present a comparative experimental evaluation of the eager and lazy approaches[11]. To this end we implemented both the lazy theory solver LBV as well as an eager theory solver within the SMT solver CVC4. After applying the same preprocessing steps as the lazy solver, the eager solver uses standard bit-blasting techniques to assert the formula to its MiniSAT backend. To gauge the complementarity of the two approaches we used CVC4's portfolio infrastructure which allows us to run the two solvers in different parallel threads. In this setup, CVC4 waits for the first thread that finishes with an answer and then kills the other, thus getting the best performance between the two theory solvers each time (modulo memory usage).

All experiments were performed on AMD Opteron 250 2.4GHz machines with a time limit of 5 minutes and memory limit of 3GB. We evaluate our solvers' performance on a large selection of SMT-LIB v2.0 benchmarks from the $\mathsf{QF\_BV}$ logic [5]. Because of time constraints, we could not include all 31K $\mathsf{QF\_BV}$ benchmarks from SMT-LIB v2.0. Instead, we selected 3786 of them by focusing on examples coming

[10] A different technique to reduce the number of literals sent to theory solvers is proposed in [12].

[11] Source code and binaries at `http://cvc4.cs.nyu.edu/papers/CAV2014-bitvectors/`



(a) cvcLz vs cvcLz-J (b) cvcLz vs cvcLz-P (c) cvcLz vs cvcLz-Alg

Fig. 2: Impact of various features of the lazy solver. All plots are on a logarithmic scale.

from verification applications: we excluded the answer-set programming asp family as well as the check2 and crafted families that contain toy examples. To prevent very large families such as sage (26K) and spear (1694) from dominating the results, we used a randomized process to select a representative fraction of the benchmarks from them. Because many of the sage problems are very easy, we considered only benchmarks that take more than 10 seconds to solve. From the spear family we included all small sub-families, and randomly selected a fraction of the largest subfamily. For brevity, we merge here the four families with a brummayerbiere prefix into brummayerbiere*, uclid and uclid-contrib-smtcomp09 into uclid*, and stp and stp-samples into stp*.

We use cvcE to refer to the implementation of the eager solver in CVC4, cvcLz for the lazy LBV solver and cvcPll for the parallel solver. The letters preceeded by a minus sign represent which feature of cvcLz has been *turned off*: J for the justification heuristic, P for $\mathsf{LBV}_{\mathsf{bb}}$ propagation, Alg for all of the algebraic sub-solvers ($\mathsf{LBV}_{\mathsf{eq}}$, $\mathsf{LBV}_{\mathsf{core}}$, $\mathsf{LBV}_{\mathsf{ineq}}$) plus the word-level in-processing techniques.

The scatter plots in Figure 2 compare the runtime performance of the full featured lazy solver with a version without one of the features above. Figure 2a shows the impact of the justification heuristic. While overall the justification heuristic improves performance, it has a negative impact on benchmarks in the mcm family. These problems consist of conjunctions of large disjunctions. On such problems the justification heuristic forces $\mathrm{SAT}_{\mathsf{main}}$ to choose a naive pattern of decisions by always initially deciding on the first disjunct of each conjunct. Figure 2b shows that $\mathsf{LBV}_{\mathsf{bb}}$ propagation is essential to solving difficult benchmarks, although it adds some overhead to the easier ones. Figure 2c shows the impact of all the word-level techniques enabled by the lazy approach. The plot shows a relatively small overhead when these techniques do not help, but dramatic improvements when they do apply.

Table 2 compares the performance of cvcE, cvcLz and that of the only other bit-vector solver that supports lazy bit-blasting: mathsatL (smtcomp2012 version with lazy solving enabled). The eager solver cvcE performs better on families that involve bit-level manipulations, such as the brummayerebiere* families. The lazy solver cvcLz excels on families calypto, tacas07, lfsr, core and simple_processors that benefit from al-

Table 2: Eager vs Lazy

	cvcE		cvcLz		mathsatL	
set	solved	time (s)	solved	time (s)	solved	time (s)
vs (VS3,11)	**0**	**0.0**	0	0.0	0	0.0
be (bench-ab,285)	285	57.5	285	2.4	**285**	**2.4**
br (brummayerbiere*,206)	**138**	**3732.3**	112	2923.2	100	3937.5
co (core,672)	132	3208.4	**672**	**596.4**	509	22345.5
lf (lfsr,240)	186	9451.9	**240**	**2286.3**	177	12412.2
si (simple-processor,64)	33	1566.4	**64**	**48.7**	18	845.6
ca (calypto,23)	10	9.2	**15**	**100.7**	11	233.4
dw (dwp-formulas,332)	332	68.2	**332**	**5.5**	332	5.9
ga (galois,4)	1	0.4	**1**	**0.4**	1	2.5
gu (gulwani-pldi08,6)	**6**	**49.1**	6	63.9	6	73.8
mc (mcm,185)	**64**	**3937.7**	13	392.9	2	278.9
pi (pipe,1)	**0**	**0.0**	0	0.0	0	0.0
ru (rubik,7)	5	157.9	2	110.6	**6**	**313.4**
sa (sage,189)	188	205.0	188	174.9	**189**	**51.2**
sp (spear,680)	**675**	**24057.0**	648	9347.0	478	14579.5
st (stp*,427)	424	170.3	424	108.6	**425**	**70.5**
ta (tacas07,5)	3	19.3	5	294.4	**5**	**136.8**
uc (uclid*,423)	414	2651.5	420	3148.9	**420**	**1132.5**
uu (uum,8)	**2**	**33.9**	1	1.5	1	0.3
wi (wienand-cav2008,18)	**14**	**32.2**	14	34.7	14	37.5
	2912	49408.4	**3442**	**19641.2**	2979	56459.6
us (unique-solve)	4		6		0	

Table 3: Comparison with other solvers

	cvcPll		yices2		stp2		z3		boolector		sonolar		mathsat	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
vs (11)	0	0.0	0	0.0	1	270.3	**3**	**341.7**	2	258.7	0	0.0	0	0.0
be (285)	285	39.1	**285**	**0.0**	285	0.2	285	8.5	285	3.0	285	0.1	285	2.5
br (206)	137	3024.0	113	1718.1	143	3188.5	115	4005.1	**155**	**4060.8**	125	1858.9	123	3741.9
co (672)	**672**	**726.6**	326	5717.9	191	3126.4	672	798.4	656	32176.8	266	2796.8	587	21791.1
lf (240)	**240**	**2481.3**	181	8394.7	196	8896.3	232	12183.3	213	15939.2	219	3385.1	139	7644.1
si (64)	**64**	**57.8**	35	824.3	54	1911.1	60	1134.6	60	2377.2	37	1038.4	25	1283.3
ca (23)	**15**	**349.1**	9	6.1	11	3.5	11	50.8	9	45.0	9	20.4	11	56.2
dw (332)	332	47.4	**332**	**0.0**	332	0.9	332	10.0	332	0.0	332	0.2	332	4.2
ga (4)	1	0.5	1	0.1	**1**	**0.1**	1	0.2	1	0.3	1	0.1	1	0.6
gu (6)	6	44.8	**6**	**25.5**	6	26.7	6	31.2	6	42.1	6	39.3	6	56.5
mc (185)	**63**	**6152.2**	54	5308.3	44	3616.9	55	4302.8	45	3452.2	50	3592.0	42	3429.4
pi (1)	**0**	**0.0**	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
ru (7)	5	142.7	5	99.5	7	323.4	6	148.2	7	343.5	**7**	**190.1**	6	342.8
sa (189)	188	215.5	**189**	**9.9**	189	35.2	189	49.5	189	706.9	189	39.9	189	49.1
sp (680)	677	11028.4	**680**	**400.5**	679	1756.6	675	7546.6	676	5360.9	677	6910.1	676	13175.0
st (427)	424	168.0	**425**	**5.1**	425	41.9	425	58.8	425	22.9	425	46.7	425	47.1
ta (5)	5	249.8	3	1.5	5	348.4	3	7.2	5	465.6	5	410.4	**5**	**54.9**
uc (423)	419	3315.6	416	58.6	422	902.0	421	1856.4	422	1368.0	**423**	**1207.7**	423	1226.6
uu (8)	2	605.9	2	30.4	2	29.1	**2**	**11.1**	2	11.5	2	17.7	2	64.5
wi (18)	14	30.8	14	68.6	14	64.6	14	41.4	**14**	**23.3**	9	36.1	14	36.6
	3549	**28679.7**	3076	22669.3	3007	24542.1	3507	32585.9	3504	66658.1	3067	21590.1	3291	53006.6
us	*		3		1		2		10		0		1	

gebraic reasoning. Furthermore, cvcLz solves 6 problems that none of the other solvers we considered could solve in the given time limit. The unique-solve row at the bottom of Table 2 and Table 3 shows this figure for all other solvers.

Finally, in Table 3 we compare cvcPll with other state-of-the-art bit-vector solvers: yices (2.1.1), stp2 (r1673), z3 (r0e74362), boolector (1.6), sonolar (smtcomp2012) and mathsat (smtcomp2012 with eager solver). For the parallel solver cvcPll we report wall clock time. The portfolio solver cvcPll solves the largest number of problems. We attribute this increase in performance to the complementary nature of the two approaches. To illustrate that the lazy cvcLz approach complements eager solvers, we also simulated running cvcLz in parallel with two of the most efficent eager bit-vector solvers: boolector and z3. We did this by chosing the best result from either solver for each problem. Even for these solvers, cvcLz greatly improves on their performance: the combined boolector+cvc4L solves 57 more problems in a quarter of the original boolector total time and z3+cvcL solves 42 more problems in just over half the total time.

Discussion We now provide a more detailed analysis of the tradeoffs between the two approaches, based on our experimental results.

The eager solver cvcE is particularly efficient on hardware equivalence checking benchmarks that verify the equivalence of a bit-level implementation to its word-level specification. In such cases the correctness of the proof often depends on bit-level properties that benefit from efficient propositional analysis more than the kind of algebraic reasoning done in the lazy solver. This is especially obvious in the difference in the performance of cvcE and cvcLz on the brummayerbiere* family, as can be seen in Table 2.

Maintaining the word-level structure during the computation in LBV requires establishing a common language between SAT$_{\mathsf{main}}$, the SAT solver driving the main DPLL(T) search, and SAT$_{\mathsf{bb}}$. In our approach, this language consists of the T_{bv}-atoms and represents a frontier that partitions the problem between the two solvers. LBV conflicts can be seen as interpolants between the part of the problem describing the control flow (the Boolean abstraction) and the datapath. Restricting the conflict language to T_{bv}-atoms limits the granularity of the conflicts: we cannot express bit-level conflicts. In some cases this can prove inefficient. Consider the following example.

Example 2. The following assertions are unsatisfiable. All paths through the disjunction force the last bit of the x_i variables to be $0_{[1]}$. Therefore their disjunction must also have the least significant bit equal to $0_{[i]}$ which makes the equality false.

$$\bigvee_{i=0}^{n} x_i = y \circ 1_{[1]} \wedge \bigwedge_{i=0}^{n} (x_i = t_i \circ 0_{[1]} \vee x_i = s_i \circ 0_{[1]})$$

In Example 2, an eager solver may potentially learn that the last bit of x_i has to be 0. The lazy solver on the other hand, will have to try all possible paths through the disjunction and learn a conflict for each one of them.

For problems with expensive arithmetic operators, the benefits of maintaining the word-level structure outweigh this limitation. While eager solvers have sophisticated rewrite techniques, such techniques are usually only applicable at the top level. Equivalence checking problems between higher level designs can require proving the equivalence of results obtained by taking different control-flow paths. These can be encoded as large ite (if-then-else) term trees with a similar structure, as in the following example.

Example 3. The formula below is unsatisfiable. The conditions on all paths through the ite trees force the leaves to be equal.

$$\begin{aligned} &ite(x_0 = y_0, x_0 * (ite(x_1 = y_1, 2 * x_1, 2)), 2) \neq \\ 2 * \, &ite(x_0 = y_0, y_0 * (ite(x_1 = y_1, y_1, 1)), 1) \end{aligned}$$

Collecting the assertions down any ite path in the example, and applying simple equality substitutions renders each such path trivially unsatisfiable. No multiplication reasoning is required. However, bitblasting this expression results in a difficult SAT problem as the large circuits required to model the products obscure the trivial inconsistency. The calypto, lfsr and simple_processors (Table 2) exhibit this type of structure. On these families, our LBV in-processing module can often simplify each call to `TheoryCheck` to false or a significantly simpler circuit. Other verification problems, such as checking the correctness of sorting algorithms, rely on the arithmetic properties of a total order. The equality, core and inequality subsolvers can decide such problems, often without any bit-level reasoning at all.

7 Future Work

For future work, we plan to both improve the performance of the lazy solver and investigate heuristics for automatically selecting between the eager and lazy solvers. In Section 6 we gave some intuition for which of the two approaches is best suited for which problem structure. It would be interesting to see if it is possible to statically determine which solver is likely to perform better.

The lazy solver can be improved by adding more sub-theory solvers, such as a subsolver complete for some fragment of modular arithmetic. The inprocessing module currently only handles equality reasoning, xor solving and slicing. Although it is already remarkably efficient, the `SolveEq` routine could be generalized to other types of equation solving.

Another way to improve the performance of the lazy solver is to minimize the conflicts obtained from the bit-blasting subsolver. The conflicts returned by that subsolver with assumptions infrastructure are not guaranteed to be minimal. Indeed, in our experience they are often non-minimal, in some cases larger than minimal ones by a factor of 10. The challenge here is to minimize the conflict in an efficiently since satisfiability queries in T_{bv} are potentially very expensive.

One way to expand the scope of the lazy bit-vector solver, and overcome some of its limitation, would be to increase the kind of conflicts it can return. Currently, the solver can only return conflicts in terms of bit-vector atoms. It would be interesting to experiment with expanding this vocabulary dynamically, by adding conflicts that refer to individual bits of the terms. This could potentially be supported by using the *splitting on demand* framework [3].

References

1. D. Babić and M. Musuvathi. Modular arithmetic decision procedure. *Microsoft Research Redmond, Tech. Rep. TR-2005-114*, 2005.
2. C. Barrett and J. Donham. Combining SAT methods with non-clausal decision heuristics. *Electronic Notes in Theoretical Computer Science*, 125(3):3–12, 2005.
3. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In M. Hermann and A. Voronkov, editors, *LPAR '06*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer-Verlag, Nov. 2006.
4. C. Barrett, A. Stump, and C. Tinelli. The smt-lib standard: Version 2.0. In *SMT*, volume 13, 2010.
5. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
6. C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC*, pages 522–527, 1998.
7. R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, pages 174–177. 2009.
8. R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT BV solver for hard industrial verification problems. In *CAV*, pages 547–560, 2007.
9. R. Bruttomesso and N. Sharygina. A scalable decision procedure for fixed-width bit-vectors. In *ICCAD 2009*, pages 13–20, 2009.
10. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *TACAS*, pages 93–107. 2013.
11. D. Cyrluk, O. Möller, and H. Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *CAV*, pages 60–71, 1997.
12. L. de Moura and N. Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.
13. L. De Moura and N. Bjrner. Z3: An efficient SMT solver. *TACAS*, page 337340, 2008.
14. D. Detlefs, G. Nelson, and J. Saxe. Simplify: a theorem prover for program checking. *JACM*, 52(3):365–473, 2005.
15. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.
16. A. Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.
17. H. Fujiwara, S. Member, T. Shimono, and S. Member. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, 32:1137–1144, 1983.
18. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.
19. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 131–153. IOS Press, February 2009.
20. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract DPLL procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.