Turbo-Charging Lemmas on Demand with Don’t Care Reasoning

Aina Niemetz, Mathias Preiner, and Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract—Lemmas on demand is an abstraction/refinement technique for procedures deciding Satisfiability Modulo Theories (SMT), which iteratively refines full candidate models of the formula abstraction until convergence. In this paper, we introduce a dual propagation-based technique for optimizing lemmas on demand by extracting partial candidate models via don’t care reasoning on full candidate models. Further, we compare our approach to a justification-based approach similar to techniques employed in the context of model checking. We implemented both optimizations in our SMT solver Boolector and provide an extensive experimental evaluation, which shows that by enhancing lemmas on demand with don’t care reasoning, the number of lemmas generated, and consequently the solver runtime, is reduced considerably.

I. INTRODUCTION

Procedures for deciding satisfiability of first order formulas w.r.t. first order theories, also known as Satisfiability Modulo Theories (SMT), are usually divided into so-called eager and lazy approaches. Eager SMT approaches eagerly encode an SMT formula into an equisatisfiable Boolean formula, which then serves as input for a SAT solver. Lazy approaches, on the other hand, are generally based on a tight integration of a SAT solver and one or more theory solvers. The SAT solver typically enumerates Boolean truth assignments satisfying a Boolean abstraction of the input formula, whereas the theory solver(s) not only check if those assignments are consistent w.r.t. the first order theory(s), but guide the SAT solver through its search. The majority of state-of-the-art SMT solvers employ lazy SMT approaches, where the lemmas on demand procedure as introduced for the extensional theory of arrays in [7] is one extreme variant thereof [20]. The core idea of lemmas on demand is similar to the Counterexample-Guided Abstraction Refinement (CEGAR) approach introduced in [9] and goes back to [11], while at the same time, a related technique was proposed in the context of bounded model checking, where all-different constraints are lazily encoded over bit vectors (see also [5]). Recently, in [19] we introduced a generalization of the lemmas on demand decision procedure in [7] to lazily handle $\lambda$ terms.

Similar to other lazy SMT approaches, lemmas on demand as in [7][19] enumerates truth assignments (so-called candidate models) of the bit vector abstraction of the (preprocessed) input formula and iteratively refines those assignments with lemmas until convergence. Each of these candidate models is a full truth assignment of the formula abstraction, which subsequently needs to be checked for consistency w.r.t. the theory of bit vectors with arrays. A full candidate model, however, includes parts of the formula abstraction irrelevant to its satisfiability under the current assignment and might therefore be over-determined.

In this paper we aim at exploiting a posteriori observability don’t cares, i.e., parts of the formula abstraction irrelevant under the current assignment. We show that don’t care reasoning on full candidate models to extract partial candidate models subsequently reduces the cost for consistency checking by focusing on the relevant parts of the formula, only. Motivated by dual propagation techniques in the context of quantified boolean formulas (QBF) [15][16], we propose an optimization of the lemmas on demand procedure in [19] and compare our approach to a technique based on justification heuristics in ATPG [18]. We implemented both techniques in our SMT solver Boolector and analyse the results in comparison to the version of Boolector that won the QF_AUFBV track of the SMT competition 2012.

Note that in this paper, our justification-based approach mainly serves as a basis for comparison to our dual propagation-based approach. In the context of SMT, Barrett and Donham [3] and De Moura and Bjørner [10] applied justification-based techniques to prune the search space of DPLL(T). In the context of model checking, justification-based techniques have been previously employed to identify a posteriori observability don’t cares. Bingham and Hu [6], e.g., prune the search space of their simulation-based bounded model checking engine by means of a justification-based generalization mechanism (skip cubes) similar to learning and non-chronological backtracking of conventional SAT procedures. Eén et al. [13] employ a related approach when generalizing proof obligations by ternary simulation for property directed reachability (PDR), whereas Chockler et al. [8] use a variant of offline dual propagation for SAT. The verification tool Reveal [2][1], on the other hand, employs a CEGAR approach for model checking complex hardware designs and generalizes candidate counter examples by justification techniques similar to our justification-based method. Their (and our) justification-based approach, however, is only applicable on structural (non-axiomatic) problems. In contrast, our dual propagation-based approach generalizes full candidate models by exploiting the duality of the Boolean layer of the input formula and is not restricted to structural formula abstractions.
Fig. 1: The workflow of the lemmas on demand decision procedure $\text{DP}_{\text{LOD}}$ in Boolector. The original procedure $\text{DP}_{\text{LOD}}$ (indicated by the dashed line) works on full candidate models, whereas the optimized procedure $\text{DP}_{\text{LODopt}}$ extracts partial candidate models prior to consistency checking.

II. LEMMAS ON DEMAND AT A GLANCE

The lemmas on demand decision procedure as implemented in Boolector is an iterative abstraction/refinement approach for the quantifier-free theory of fixed-sized bit vectors and arrays. Figure 1 gives a high-level view of the procedure and introduces both the original, unoptimized procedure $\text{DP}_{\text{LOD}}$ and our optimized approach $\text{DP}_{\text{LODopt}}$ as follows.

Given a formula $\phi$, $\text{DP}_{\text{LOD}}$ uses a bit vector skeleton of the preprocessed formula $\pi$ as formula abstraction $\alpha(\pi)$. In each iteration, an underlying decision procedure $\text{DP}_B$ determines the satisfiability of the refined formula abstraction $\Gamma \equiv \alpha(\pi) \land \xi$ by encoding $\Gamma$ to SAT and determining its satisfiability by means of a SAT solver. Note that initially, formula refinement $\xi$ is $\top$. As $\Gamma$ is an overapproximation of $\phi$, $\text{DP}_{\text{LOD}}$ immediately concludes with unsat if $\Gamma$ is unsatisfiable. If $\Gamma$ is satisfiable, the current (full) candidate model $\sigma(\alpha(\pi) \land \xi)$ is checked for consistency w.r.t. the preprocessed input formula $\pi$. If $\sigma(\alpha(\pi) \land \xi)$ is consistent, $\text{DP}_{\text{LOD}}$ immediately concludes with sat. Otherwise, $\sigma(\alpha(\pi) \land \xi)$ is spurious and a lemma $l$ is added to formula refinement $\xi$.

As indicated in Fig. 1, $\text{DP}_{\text{LOD}}$ iteratively refines $\alpha(\pi)$ by consistency checking full candidate models, which usually include parts of the bit vector skeleton irrelevant to its satisfiability under the current assignment. In the following section, we will introduce an optimization to extract a partial candidate model $\sigma_p(\alpha(\pi) \land \xi)$ from the full candidate model $\sigma(\alpha(\pi) \land \xi)$ in order to guide the consistency check towards the relevant parts of $\alpha(\pi)$ only.

III. PARTIAL MODEL EXTRACTION

In terms of runtime, abstraction refinement usually is the most costly part of the lemmas on demand procedure $\text{DP}_{\text{LOD}}$, where cost generally correlates with the number of lemmas generated. During refinement, procedure $\text{DP}_B$ (and consequently the call to the underlying SAT solver) constitutes the majority of the overall runtime per iteration, which adds up when a great number of refinement iterations is needed. Hence, optimizing $\text{DP}_{\text{LOD}}$ in terms of runtime directly translates to reducing the number of lemmas generated.

In contrast to other lazy SMT approaches [20], formula abstraction in $\text{DP}_{\text{LOD}}$ does not produce a pure Boolean skeleton, but a bit vector skeleton, where each function application $f(a_0,\ldots,a_n)$ in the preprocessed formula $\pi$ is mapped to a fresh bit vector variable. Consequently, consistency checking in $\text{DP}_{\text{LOD}}$ is performed on all function applications in the bit vector skeleton (for details see [19]).

A high level view of the consistency checking algorithm consistent in $\text{DP}_{\text{LOD}}$ is given in Fig. 2 and proceeds as follows. Given the refined formula abstraction $\Gamma$ and the full candidate model $\sigma$, search_initial_applications collects all function applications in $\Gamma$ that need to be checked for consistency (line 2) and iteratively checks each APPLY $f(a_0,\ldots,a_n)$ w.r.t. the current assignment $\sigma$ (lines 4-5). If check_consistency encounters an inconsistency, consistent immediately returns with $\bot$. Else, search_applications_for_consistency_check instantiates function $f$ with arguments $a_0,\ldots,a_n$, which yields term $t$, and subsequently collects all function applications in formula abstraction $\alpha(t)$ for consistency checking (lines 7-8). If all applies in $\Gamma$ have been checked without inconsistencies, procedure consistent concludes that current candidate model $\sigma$ is consistent and returns $\top$.

Consistency checking all function applications in formula abstraction $\Gamma$ corresponds to checking the full candidate model $\sigma$ for consistency, with the order in which applies are checked as the only way to positively influence the number of refinement iterations (by coincidentally finding lemmas that shortcut the search, early on). Checking the full candidate model, however, is often not required, as only a small subset of the full candidate model is responsible for actually satisfying the formula abstraction. As a consequence, parts of the formula abstraction irrelevant to its satisfiability under the current assignment are checked, which subsequently produces lemmas that do not necessarily prune the search space and therefore mainly cost runtime.

Example 1: As a running example, consider the formula

$$\psi_1 \equiv i \neq k \land (f(i) = e \lor f(k) = v) \land v = \text{ite}(i = j, e, g(j))$$

given as in Fig. 3. Its initial formula abstraction $\Gamma_{\psi_1} \equiv \alpha(\psi_1)$...
and a (possible) initial full candidate model $\sigma(\Gamma_\psi_1)$ (indicated in red) is given in Fig. 4. In the following, we assume that all variables in $\psi_1$ are bit vector variables of size 2 and $\Gamma_\psi_1$ is a bit vector skeleton. For the sake of simplicity, we further assume that functions $f$ and $g$ represent uninterpreted functions, i.e., we concentrate on consistency checking of the full versus a partial candidate model (via procedure search_initial_applies) and do not bother with details of the internals of the actual consistency check (for details, see [19]). Procedure search_initial_applies initially collects all function applications in $\Gamma_\psi_1$ (apply1, apply2, apply3) to be checked for consistency. During consistency checking, however, no further applications are identified as required to be checked (procedure search_applies_for_consistency_check) as both $f$ and $g$ do not make subsequent calls to other functions. Note that given $\sigma(\Gamma_\psi_1)$, instead of checking all applications in $\psi_1$, either checking apply1 or apply2 would be sufficient.

In the following, we consider two techniques for identifying irrelevant parts of the formula abstraction by extracting partial candidate models, which subsequently reduces the number of refinement iterations, and therefore, the overall runtime of the lemmas on demand procedure.

A. Justification-Based Partial Model Extraction

In the context of ATPG [18], sets of don’t care conditions are usually divided into observability don’t cares (ODC) and controllability don’t cares (CDC). The former denotes lines that do not influence the primary outputs (independent from the current assignment to the primary inputs), and the latter identifies line values that can not be justified and are therefore illegal under any assignment to the primary inputs. Given a concrete assignment to the primary inputs, however, we can determine what we call a posteriori observability don’t cares, i.e., lines that do not influence the output of a gate under its current assignment. In the context of model checking, such a posteriori ODC have already been exploited by Bingham and Hu [6], Eén et al. [13], and Andraus et al. [2][1].

In this section, we introduce a technique similar to [2][1] and extract partial candidate models by identifying parts of the formula abstraction $\Gamma$ that are irrelevant to its satisfiability under the current assignment $\sigma$. As indicated above, this directly translates to collecting and checking function applications in relevant parts of $\Gamma$ only. In the following, we assume that $\Gamma$ is represented as a directed acyclic graph (DAG) with exactly one root, where all Boolean operations are expressed by means of NOT and (two-input) AND gates. In place of procedure search_initial_applies, we introduce search_initial_appliesjust (Fig. 5), which collects function applications while traversing all relevant paths in $\Gamma$ as follows.

Given $\Gamma$ and a full candidate model $\sigma$, starting from the root, search_initial_appliesjust iteratively traverses $\Gamma$ towards its primary inputs (bit vector variables and function applications) in depth first search (DFS) order. That is, initially, root ($\Gamma$) is pushed onto stack $X$ (line 2) and for each node $x \in X$ we determine the paths to be skipped as follows. If a node $x$ is an AND node and its output is assigned to $\bot$, we follow (one of) its controlling input(s), i.e., one of its inputs with controlling value ($\bot$ for an AND) [18], and skip the other (lines 7-14). Similarly, if $x$ is an IF-THEN-ELSE (ITE) node and its condition is assigned to $\top$ (resp. $\bot$), we follow both its condition and its then (resp. else) branch (lines 15-20). In any other case where $x$ is not an APPLY node, we follow all inputs of node $x$ (line 22). However, if $x$ is an APPLY node, we collect $x$ (line 6) and cut off the traversal, as function applications are treated as fresh bit vector variables in the formula abstraction.

Note that in the case that both inputs of an AND node are controlling, we can skip either one of them (lines 9-10). Hence, we choose to follow the input with minimum cost in terms of consistency checking, where the cost of a node $x$ is defined as the minimum number of (unique) applies along a path from $x$ to the primary inputs in the preprocessed formula $\pi$. Similar as controllability measures in ATPG [18],
procedure search_initial_applesjust (Γ, σ)
S ← ∅, X ← {root (Γ)}
while X ̸= ∅
x ← pop (X)
if is_apply (x)
   push (S, x)
   e l i f is_and (x) and σ (x) = ⊥
      l ← left_input (x), r ← right_input (x)
      if is_controlling (l) and is_controlling (r)
         push (X, choose (l, r))
      e l i f is_controlling (l)
         push (X, l)
      e l s e
         push (X, r)
   e l i f is ITE (x)
      push (x, condition (x))
      if σ (condition (x)) = ⊤
         push (X, then (x))
      e l s e
         push (X, else (x))
      e l s e
   push (X, i ∈ inputs (x))
return S

Fig. 5: Procedure search_initial_applesjust in pseudo-code.

we recursively define a cost function cost (x) as follows.

cost (x) = \begin{cases} 
0 & \text{if is_var (x)} \\
\min \{ \text{cost} (i) \mid i ∈ \text{inputs} (x) \} & \text{if is_and (x)} \\
\sum \{ \text{cost} (i) \mid i ∈ \text{inputs} (x) \} + 1 & \text{if is_apply (x)} \\
\sum \{ \text{cost} (i) \mid i ∈ \text{inputs} (x) \} & \text{otherwise}
\end{cases} \quad (1)

Given formula π, a bit vector variable is a primary input, hence its cost is defined as 0. Function applications, on the other hand, are not primary inputs but define the cost of a path from input π to the primary inputs. Hence, the cost of an APPLY is defined as the sum of the costs of its inputs increased by one. In case of an AND node, we want to choose the input with minimum cost if both inputs are controlling, hence cost is defined as the minimum cost of its inputs. In any other case, all input paths have to be followed and cost (x) is defined as the sum of the costs of all inputs of x.

Example 2: Consider formula ψ₁, formula abstraction Γψ₁, and a full candidate model σ(Γψ₁) as given in Example 1. Starting from the root (and₁), procedure search_initial_applesjust traverses Γψ₁ in DFS order while identifying (and skipping) all paths irrelevant w.r.t. assignment σ(Γψ₁). Note that in Fig. 3 and 4, inverted nodes are indicated by black dots. In the following, however, we will interpret an inverted node as two distinct nodes (with resp. distinct assignments), i.e., ¬and₁ with σ(¬and₁) = ⊤ in Fig. 4, for example, is treated as a NOT (assigned to ⊤) in front of an AND (assigned to ⊥). Starting with root and₁, which is assigned to ⊤, neither of its inputs may be skipped and we first travel down towards eq1, whose inputs are both bit vector variables. Hence, we immediately continue with and₁ (also assigned to ⊤) and follow its input eq₁, where we encounter an ITE with its condition assigned to ⊤. We skip the else branch, no APPLY is collected, and we continue down the input path leading to and₃, which is assigned to ⊥. Both inputs of and₃ are controlling (i.e., assigned to ⊤), hence we choose one of them heuristically. The minimum cost for both paths, however, is 0 (as the body of function f does not contain any further applies), hence we may choose either. We decide on the path to apply and conclude with S = {apply₁}, which corresponds to the partial model to be subsequently checked for consistency.

B. Dual Propagation-Based Partial Model Extraction

Exploiting the duality of QBF by propagating a dual set of values through a QBF ψ and its negation ¬ψ, also referred to as dual propagation, has successfully been employed in [15] to significantly prune, and therefore speed up the search in circuit-based QBF solvers. The core idea of dual propagation, however, is neither restricted to circuit-based representations [16] nor to QBF and is based on the fact that assignments satisfying an input formula ψ (the primal channel), falsify its negation ¬ψ (the dual channel) and vice versa. Given a Boolean formula ψ₂ ≡ (a ∨ b) ∨ (c ∧ d), for example, assignment \{σ(a) = ⊤, σ(b) = ⊤, σ(c) = ⊤, σ(d) = ⊤\} satisfies ψ₂, but falsifies its negation ¬ψ₂ ≡ (¬a ∨ ¬b) ∧ (¬c ∨ ¬d).

The duality of formula ψ₂, however, can be further exploited. Assume, for example, that given ψ₂ and σ(ψ₂) as above, we fix the values of all input variables assigned in σ(ψ₂) by making assumptions \{a = ⊤, b = ⊤, c = ⊤, d = ⊤\} to a SAT solver maintaining its negation ¬ψ₂. All assumptions inconsistent with ¬ψ₂, also called failed assumptions [14], identify all input assignments sufficient to falsify ¬ψ₂, hence sufficient to satisfy ψ₂. This set of failed assumptions, for example \{a = ⊤, b = ⊤\}, therefore represents a partial model satisfying ψ₂. Note that our approach does not require a structural SAT solver—structural don’t care reasoning is simulated via the dual solver, which maintains ¬ψ₂ in CNF. Consequently, given a CNF representation of ψ₂ (where structural information of ψ₂ is essentially lost), we extract a partial model (disregarding structural don’t cares w.r.t. assignment σ) that satisfies ψ₂ but not necessarily its encoding to CNF. Consider, for example, the Tseitin encoding CNF(ψ₂) ≡ (¬a ∨ x ∨ y) ∧ (¬x ∨ a) ∧ (¬y ∨ a) ∧ (¬x ∨ x) ∧ (¬x ∨ b) ∧ (¬a ∨ ¬b ∨ x) ∧ (¬y ∨ a) ∧ (¬y ∨ b) ∧ (¬c ∨ d) ∧ (¬c ∨ ¬d ∨ y).

Our previous partial model \{a = ⊤, b = ⊤\} satisfies ψ₂ (and therefore identifies those parts of ψ₂ relevant to its satisfiability) but does not satisfy all clauses in CNF(ψ₂). This is in contrast to other partial model extraction techniques based on iterative removal of unnecessary assignments on the CNF level (e.g. [12]), which do not enable structural don’t care reasoning and therefore need to satisfy all clauses in CNF(ψ₂).

In this section, we lift the approach sketched above to the word level by means of a dual SMT solver and introduce a technique to extract partial candidate models via dual propagation-based don’t care reasoning. Given a formula abstraction Γ = α(π) ∧ ξ, we use a single dual solver instance to maintain ¬Γ over all refinement iterations in combination with the primal (or main) solver. However, since in each iteration i a new lemma lᵢ is added to ξ ≡ l₁ ∧ ... ∧ lᵢ₋₁, we set up the dual solver to maintain ¬Γ ≡ ¬(α(π) ∧ l₁ ∧ ... ∧ lᵢ₋₁ ∧ lᵢ)
procedure search_initial_appliesdp (Γ, σ)
1 S ← ∅, A ← ∅
2 assume (dual_solver, ¬Γ)
3 X ← collect_primary_inputs (Γ)
4 for x ∈ X
5 a ← x = σ(x), A ← A ∪ a
6 assume (dual_solver, a)
7 res ← DPB (dual_solver)
8 assert res = UNSAT
9 for a ∈ A
10 x, σ(x) ← a
11 if is_failed (a) and is_apply (x)
12 push (S, x)
13 return S

Fig. 6: Procedure search_initial_appliesdp in pseudo-code. Solver instance dual Solver simulates the dual channel and is maintained globally.

as assumption rather than assertion. As illustrated in Fig. 6, we introduce search_initial_appliesdp in place of procedure search_initial_applies as follows.

Given Γ and a full candidate model σ, procedure search_initial_appliesdp initializes the dual solver by assuming ¬Γ (line 3). The value of all primary inputs in ¬Γ is then fixed by making assumptions of the form x = σ(x), where x is either a bit vector variable or an abstracted function application, and σ(x) is its assignment in the current full candidate model σ (lines 4-7). Candidate model σ represents a satisfying assignment for Γ, hence decision procedure DPB must conclude that assuming σ, ¬Γ is unsatisfiable (lines 8-9).

The resulting set of failed assumptions identifies all relevant parts of Γ w.r.t. assignment σ, and all function applications in the set of failed assumptions are subsequently collected for consistency checking (lines 10-13).

Example 3: Again, consider formula ψ₁, its initial formula abstraction Γ₂ψ₁ ≡ α(ψ₁), and a (possible) full candidate model σ(ψ₁) as given in Example 1. Procedure search_initial_appliesdp initializes the dual solver by assuming ¬Γ₂ψ₁ ≡ ¬(i ≠ k ∧ (α(apply₁) = e ∨ α(apply₂) = v) ∧ v = ite(i = j, e, α(apply₃)), and subsequently collects all bit vector variables i, j, k, e, v and abstracted function applications α(apply₁), α(apply₂), α(apply₃) in Γₐ onto stack X. All primary inputs x ∈ X are then fixed by making assumptions {i = 0, j = 0, k = 0, e = 0, v = 0, α(apply₁) = 0, α(apply₂) = 0, α(apply₃) = 0} to the dual SMT solver instance, which concludes that under the current set of assumptions, ¬Γ₂ψ₁ is unsatisfiable. Assumption α(apply₁) = 00 is identified as failed assumption and we conclude with S = {apply₁} to be subsequently checked for consistency.

Note that in a sense, our dual propagation-based approach as discussed above simulates dual propagation as introduced in the context of QBF [15][16] rather than literally lifting it to bit vectors with arrays. Dual propagation as in [15][16] is done eagerly by means of one single solver instance maintaining a primal and a dual channel without additional overhead. Primary inputs are shared between both channels, which enables symmetric propagation between the primal and dual channel and allows to detect partial models early—even before a full assignment has been generated. In our approach, however, propagation is not interleaved, but consecutive—the primal solver generates a full assignment before the dual solver enables partial model extraction based on the primal full assignment. Further, primary inputs are not physically shared as the dual solver discretely maintains ¬φ (while mapping primary inputs back to the primal solver and vice versa). Hence we have to simulate shared inputs via fixing input values by means of assumptions to the dual solver, which simply acts as “slave” for partial model extraction to the primal solver. In order to adopt a more eager approach to enable early partial model extraction while reducing the dual solver overhead, interleaved execution between the primal and dual solver similar to “SAT modulo SAT” [4] would be required. Integrating such an interleaved decision process into an existing SMT solver has high potential, however, is rather involved to implement and left to future work.

IV. EXPERIMENTAL EVALUATION
We implemented justification-based and dual propagation-based partial model extraction in our SMT solver Boolector and provide a comparison of the following four configurations:

- **Boolector_sc**: The version that won the QF_AUFBV track of the SMT competition 2012.
- **Boolector_ba**: Our current base version of Boolector, a slightly optimized version of [19], with partial model extraction disabled.
- **Boolector_ju**: Our base version of Boolector with justification-based partial model extraction enabled.
- **Boolector_dp**: Our base version of Boolector with dual propagation-based partial model extraction enabled.

We compiled two benchmarks sets for our experimental evaluation: (1) Selected (149 instances), which consists of all non-extensional benchmarks used for the SMT competition 2012 and (2) Selected (173 instances), which includes all non-extensional benchmarks from the QF_AUFBV category of SMT-LIBf for which Boolector_required at least 10 seconds (CPU time) for solving (incl. timeouts and memouts). Note that we had to exclude extensional benchmarks as Boolector requires its optimized versions Boolector_ju and Boolector_dp do not yet support extensionality on arrays. Further note that 58 instances of the benchmark set Selected are included in Selected. All experiments were performed on 2.83Ghz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 12.04.
The memory and time limits for each solver instance were set to 7GB and 1200 seconds, respectively.

A. Results Overview

The overall results of all four solver configurations on both benchmark sets SMT'12 and Selected are shown in Table I, which summarizes the number of solved instances (Solved), timeouts (TO), memouts (MO), total CPU time (Time), and the overhead produced by the dual solver in terms of CPU time (DS). Note that the overhead introduced by our justification-based approach is negligible. Further note that in case of a timeout or memout, a penalty of 1200 seconds was added to the total CPU time. On the SMT'12 benchmark set, in terms of solved instances, Boolector_{sc}, Boolector_{ju}, and Boolector_{dp} perform slightly better than Boolector_{sa}. In terms of runtime, however, only Boolector_{ju} shows a significant improvement (of about 20%), while Boolector_{dp} appears to even perform worse than Boolector_{sa}, which is mainly due to the runtime overhead introduced by the dual solver. If we disregard this overhead, the overall runtime of Boolector_{dp} is competitive with the runtime of Boolector_{ju}. It is conceivable that an eager implementation of dual propagation would perform equally well, i.e., at least as fast as Boolector_{dp} without the overhead.

Interestingly, Boolector_{sc} clearly outperforms all other three solver configurations on the benchmark family “platania strcmp” (9 instances). Boolector_{sc} solved these benchmarks in about 31 seconds, whereas the other solvers required 4416 seconds (Boolector_{ba}), 2308 seconds (Boolector_{ju}), and 4527 seconds (Boolector_{dp}), respectively. The base version Boolector_{sa}, and consequently both Boolector_{ju} and Boolector_{dp}, obviously struggle on these benchmarks, which needs further investigation.

Note that benchmark set SMT'12 is not necessarily representative for lemmas on demand in Boolector, as 79 (53%) out of a total of 149 instances are immediately solved by Boolector_{sc} without a single refinement iteration. Benchmark set Selected, on the other hand, has been compiled based on the runtime performance of the SMT competition 2012 winner Boolector_{sc} (incl. timeouts and memouts) and represents a set considered to be “harder” for Boolector. As indicated in Table I, on set Selected both Boolector_{ju} and Boolector_{dp} clearly outperform their base version Boolector_{sa} as well as the competition configuration Boolector_{sc}. More precisely, both our justification-based and dual propagation-based optimizations considerably reduce the overall runtime while solving 14 (9) additional instances compared to Boolector_{sa} (Boolector_{ba}), where 13 (9) out of 14 (9) are satisfiable instances. Again, Boolector_{dp} is slowed down by the dual solver overhead, but still manages to solve as many instances as Boolector_{ju}. Disregarding the dual solver overhead, Boolector_{dp} even outperforms Boolector_{ju} in terms of runtime. Note that the dual solver overhead in general correlates with the number of lemmas generated. This is due to the fact that in each refinement iteration a partial candidate model is extracted from the full candidate model, which requires an additional call to the dual solver. On set Selected, for 10 out of 130 instances, the dual solver overhead constitutes about 50-70% of the total runtime per instance, whereas for 83 instances it does not exceed 10%.

B. Results Commonly Solved Instances

Table II summarizes all instances in each benchmark set that could be solved by all four solver configurations and gives an overview of the runtime required for solving (Time), the runtime required by the underlying SAT solver (Sat), the dual solver overhead (DS), the number of lemmas generated (LOD), and the size of the array models for satisfiable instances (Array Model Size). For all four solver configurations, we identified 139 common instances (82 sat, 57 unsat) on benchmark set SMT'12 and 113 common instances (70 sat, 43 unsat) on benchmark set Selected. Array model size is measured in terms of the number of index/value pairs identified by each solver with model generation enabled. However, unlike Boolector_{sa} (and consequently Boolector_{ju} and Boolector_{dp}), Boolector_{sc} requires additional overhead for model generation, which has a negative impact on the overall number of solved instances. As a consequence, Boolector_{sc} effectively “lost” 1 (resp. 3) satisfiable instance(s) on set SMT'12 (resp. Selected). We therefore compiled all columns except column Array Model Size with model generation disabled.

On the 139 common instances in the SMT'12 benchmark set, Boolector_{sc} is still the fastest solver, albeit only due to the “platania strcmp” benchmarks mentioned above—on those nine instances, Boolector_{sc}, Boolector_{ju}, and Boolector_{dp} spent 50%, 35% and 45% of the overall runtime, respectively. A similar picture emerges when comparing the number of refinement iterations required for these nine instances, which constitutes 59%, 47%, and 60% of the total number of lemmas generated by Boolector_{sa}, Boolector_{ju}, and Boolector_{dp}, respectively. In comparison to the base version Boolector_{ba}, however, Boolector_{dp} shows the most notable
Fig. 7: Runtime comparison of Boolector$_{dp}$ vs. Boolector$_{sc}$ (8a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (8b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (8c) on benchmark set _Selected_ with 1200 seconds timeout, dual solver overhead included.

Fig. 8: Runtime comparison of Boolector$_{dp}$ vs. Boolector$_{sc}$ (8a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (8b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (8c) on benchmark set _Selected_ with 1200 seconds timeout, dual solver overhead _not_ included.

Fig. 9: Comparison of the number of lemmas generated by Boolector$_{dp}$ vs. Boolector$_{sc}$ (9a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (9b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (9c) on benchmark set _Selected_.

ISBN: 978-0-9835678-4-4. Copyright owned jointly by the authors and FMCAD Inc.
improvement (about 26%) in terms of runtime required by the underlying SAT solver on the 139 common instances in SMT'12. Disregarding the dual solver overhead, Boolector$_{dp}$ even outperforms Boolector$_{ju}$ in terms of overall runtime. Interestingly, in terms of the number of lemmas generated, Boolector$_{dp}$ requires slightly more lemmas than the base version, which is in stark contrast to Boolector$_{ju}$. However, in case of Boolector$_{dp}$, this can be contributed to a relative small number of instances. On 14 instances, Boolector$_{dp}$ generates 1.5 to 2.6 times more lemmas than Boolector$_{ra}$, whereas on all other instances, Boolector$_{ra}$ requires considerably more refinement iterations than Boolector$_{dp}$. This might indicate that in some cases, Boolector$_{ra}$ coincidentally generates lemmas that shortcut the search early on. In terms of array model size, both optimized configurations Boolector$_{ju}$ and Boolector$_{dp}$ clearly show a reduction in the number of array index/value pairs compared to the base version Boolector$_{ra}$.

Note that the considerable difference in array model size between Boolector$_{sc}$ and Boolector$_{ra}$ is due to an optimization of procedure search_appplies_for_consistency_check (see Section III) introduced subsequent to [19]. In essence, given a function application $f(a)$, this optimization aims at consistency checking APPLY nodes reachable while traversing in DFS order from $f(a)$ to the primary inputs, only. In contrast, prior to that optimization it was possible that function applications irrelevant to consistency checking $f(a)$ were pulled in. The effect of this optimization is even more notable on the Selected benchmark set, where Boolector$_{ra}$ clearly outperforms Boolector$_{sc}$ in every aspect.

On the 113 common instances in set Selected, Boolector$_{dp}$ clearly outperforms Boolector$_{ju}$ and Boolector$_{ra}$ not only in terms of runtime required by the underlying SAT solver, but in the number of lemmas generated. Disregarding the dual solver overhead, Boolector$_{dp}$ shows even more improvement in terms of overall runtime than Boolector$_{ju}$. Note that without the optimization of procedure search_applies_for_consistency_check mentioned above, the difference in terms of overall runtime between Boolector$_{ra}$ and both optimized versions Boolector$_{ju}$ and Boolector$_{dp}$ would be even greater, i.e., comparable to the difference between both optimized versions and Boolector$_{sc}$.

C. Results Dual Propagation-Based Optimization

A more detailed overview of the instance-based results of our dual propagation-based approach Boolector$_{dp}$ on benchmark set Selected is given in Fig. 7-9. Figure 7 compares the overall runtime of Boolector$_{dp}$ (incl. the overhead introduced by the dual solver) with the runtime of Boolector$_{sc}$ (7a), Boolector$_{ra}$ (7b), and Boolector$_{ju}$ (7c). Even though the dual solver overhead constitutes 31% of the total runtime of Boolector$_{dp}$, it still outperforms Boolector$_{sc}$ and Boolector$_{ra}$ on a majority of the instances and is even competitive with Boolector$_{ju}$. Disregarding the overhead of the dual solver (Fig. 8), Boolector$_{dp}$ even outperforms Boolector$_{ju}$ on a majority of the instances (Fig. 8c). In terms of the number of lemmas generated (Fig. 9), in comparison to all three solver configurations Boolector$_{sc}$, Boolector$_{ra}$, and Boolector$_{ju}$, our dual propagation-based solver Boolector$_{dp}$ clearly shows the most notable improvement.

V. Conclusion

In this paper we introduced a dual propagation-based optimization of the lemmas on demand procedure for bit vectors with arrays as implemented in Boolector. We compared our approach with a justification-based approach similar to [2][1]. We showed that don’t care reasoning on full candidate models improves the performance of lemmas on demand considerably. Our current simulation of dual propagation is competitive with our justification-based optimization and clearly outperforms the winner of the SMT competition 2012, even though the dual solver introduces a considerable amount of overhead to the overall runtime. Adopting a more eager dual propagation approach promises to render the dual solver overhead negligible, while further improving the overall performance by enabling partial model extraction even before a full candidate model has been generated. However, this would require an interleaved execution between the primal and the dual solver, which is rather involved to implement and subject of future work. Further, our current version of dual propagation-based partial model extraction heavily relies on incremental SAT solving under assumptions, which can benefit from dedicated data structures [17]. The integration of such SAT solver level optimization techniques is also left to future work.

Binaries of Boolector and all log files of our experimental evaluation can be found at http://fmv.jku.at/dpjust.

REFERENCES