Blocked Clause Decomposition

Marijn J. H. Heule* and Armin Biere**

The University of Texas at Austin and Johannes Kepler University Linz

Abstract. We demonstrate that it is fairly easy to decompose any propositional formula into two subsets such that both can be solved by blocked clause elimination. Such a blocked clause decomposition is useful to cheaply detect backbone variables and equivalent literals. Blocked clause decompositions are especially useful when they are unbalanced, i.e., one subset is much larger in size than the other one. We present algorithms and heuristics to obtain unbalanced decompositions efficiently. Our techniques have been implemented in the state-of-the-art solver Lingeling. Experiments show that the performance of Lingeling is clearly improved due to these techniques on application benchmarks of the SAT Competition 2013.

1 Introduction

Random simulation is a useful technique to find patterns in Boolean circuits, such as equivalent gates and gates that are always true or false [1]. It works as follows: random values are assigned to the input gates and propagated through a given Boolean circuit. In case two gates always have the same value in many simulations, they are potentially equivalent. SAT sweeping [2] can be used to determine whether a potentially equivalent pair is indeed equivalent.

We want to lift random simulation to the domain of Boolean formulas. Yet even computing a single solution is hard for most interesting Boolean formulas. Therefore we focus on computing solutions for a satisfiable subset of a Boolean formula. The main question that arises is: which subset? If the subset is too large, then solving the formula is still hard. Hence, computing many solutions to observe patterns is too costly. On the other hand, if the subset is too small, the patterns get obscured and therefore hard to detect.

We propose to obtain a useful subset by *blocked clause decomposition*. A set of clauses is called *blocked* if and only if *blocked clause elimination* (BCE) [3] is able to remove it completely. We show that any Boolean formula can be decomposed in polynomial time into two blocked sets such that one subset is maximal. On average, the maximal subset contains about 90% of the clauses of a given formula. A major advantage of our approach is that multiple solutions for blocked sets can be computed using a linear number of steps in the size of the set. We conjecture that *all* solutions of a blocked set can be computed in a time polynomial in the number of solutions.

* Supported by DARPA contract number N66001-10-2-4087.
** Supported by Austrian Science Foundation (FWF) NFN Grant S11408-N23 (RiSE).

We want to find *backbone variables* [4] and *implied binary equivalences*. To detect these patterns, we decompose a formula into two blocked sets of which one is maximal. Afterwards, many solutions for the large subset are obtained by applying a linear time algorithm. These solutions partition the literals of the formula into equivalence classes. Literals in the same class are potentially equivalent. SAT sweeping is used to compute the backbone and equivalences of the large subset which are used to simplify the original formula. Experimental results show that this approach helps to solve hard application benchmarks.

Detection of these patterns has been studied in earlier work as well. Instead of using a subset of a formula to detect backbone variables, [5] proposes to use local minima computed by a local search solver. However, local search solvers perform poorly on most hard real-world SAT problems. For random formulas, the backbone of a formula is fragile [6]: i.e., removal of a few clauses reduces the size of the backbone. *Hyper binary resolution* (HBR) can be used to detect binary equivalences [7]. Yet HBR can only find "easy" equivalences, i.e., those that can be detected by unit propagation.

The remainder of this paper is structured as follows: first we briefly discuss the preliminaries in Section 2 and some definitions is Section 3. Section 4 deals with the theoretical results regarding blocked clause decompositions. We present in Section 5 heuristics and optimizations for decomposition algorithms. Section 6 explains how decompositions can be used to find backbone variables and binary equivalences. Experimental results are shown in Section 7 and we draw some conclusions in Section 8.

2 Preliminaries

In this section we review necessary background concepts: conjunctive normal form level Boolean satisfiability, resolution and blocked clause elimination.

CNF For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a disjunction of literals and a conjunctive normal form (CNF) formula a conjunction of clauses. A clause can be seen as a finite set of literals and a CNF formula as a finite set of clauses. A *unit clause* contains exactly one literal. A clause is a *tautology* if it contains both x and $\bar{x}$ for some x. The sets of variables and literals occurring in a formula F are denoted by $\mathsf{vars}(F)$ and $\mathsf{lits}(F)$, respectively. A literal l is *pure* within a formula F if and only if $\bar{l} \notin \mathsf{lits}(F)$.

A truth assignment for a CNF formula F is a function τ that maps variables in F to $\{1, 0\}$. If $\tau(x) = v$, then $\tau(\bar{x}) = \neg v$, where $\neg 1 = 0$ and $\neg 0 = 1$. A clause C is satisfied by τ if $\tau(l) = 1$ for some $l \in C$. An assignment satisfies F if it satisfies every clause in F. An assignment falsifies a clause C if it assigns all literals that occur in C to 0. Formulas are *logically equivalent* if they have the same set of satisfying assignments over the common variables.

A variable is said to be in the *backbone* of a formula if it is assigned to the same truth value in all satisfying assignments.

Resolution and Blocked Clauses The *resolution rule* states that, given two clauses $C_1 = (l \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{l} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, called the *resolvent* of C_1 and C_2, can be inferred by *resolving* on the literal l. This is denoted by $C = C_1 \otimes_l C_2$.

Given a CNF formula F, a clause C, and a literal $l \in C$, l *blocks* C w.r.t. F if (i) for each clause $C' \in F$ with $\bar{l} \in C'$, $C \otimes_l C'$ is a tautology, or (ii) $\bar{l} \in C$, i.e., C is itself a tautology[1]. A pure literal blocks the clauses in which it occurs. *Pure literal elimination* removes clauses with pure literals until fixpoint.

A clause C is *blocked* w.r.t. a given formula F if there is a literal that blocks C w.r.t. F. Removal of blocked clauses preserves satisfiability [8]. For a CNF formula F, *blocked clause elimination* (BCE) repeats the following until fixpoint:

If there is a blocked clause $C \in F$ w.r.t. F, let $F := F \setminus \{C\}$.

BCE is confluent and does not preserve logical equivalence [9]. The CNF formula resulting from applying BCE on F is denoted by $\mathrm{BCE}(F)$. We say that BCE can *solve* a formula F if and only if $\mathrm{BCE}(F) = \emptyset$. Also note the following *monotonicity* property, which immediately follows from the definitions (also see Lemma 1 in [3]). It is a crucial observation for the rest of the paper.

Proposition 1. *If $G \subseteq F$ and C is blocked w.r.t. F, then C is blocked w.r.t. G.*

3 Definitions

Let F be a formula in CNF represented as a set of clauses. A subset $G \subseteq F$ is called a *satisfiable subset* (SS) of F, if it satisfiable. If in addition G is maximal, i.e., there is no other SS H with $G \subset H \subseteq F$, then G is called a *maximal satisfiable subset* (MSS) of F.

Note that maximality of G does not require that G is an SS of F with the largest cardinality (a solution to the MaxSAT problem). Actually if G is an MSS then the complement $F \setminus G$ is a minimal correcting subset (MCS). See [10] for more details on the relation between the notions of MSS, MCS, as well as the minimal unsatisfiable subset (MUS), and the MaxSAT problem. Similar to these standard definitions we propose the following new characterizations.

Definition 1. *Let $G \subseteq F$ be a subset of F for which* $\mathrm{BCE}(G) = \emptyset$. *Then G is called a* Blocked Subset (BS) *of F.*

Definition 2. *Let $\mathcal{BS}$ be the set of all formulas that can be solved by* BCE.

Hence all blocked subsets of any formula occur in $\mathcal{BS}$. Lemma 1 in [3] can be reformulated as follows.

[1] Here $\bar{l} \in C$ is included in order to handle the special case that for any tautological *binary* clause $(l \vee \bar{l})$, both l and $\bar{l}$ block the clause. Notice that, even without this addition, every *non-binary* tautological clause contains at least one literal that blocks the clause.

Proposition 2 ($\mathcal{BS}$ monotonicity). *If $F \in \mathcal{BS}$ and $G \subseteq F$ then $G \in \mathcal{BS}$.*

If $G \in \mathcal{BS}$, $G \subseteq F$, and maximal then G is called a *maximal blocked subset* (MBS) of F. Obviously an MBS is also an MSS, but there are of course MSSs, which are not an MBS, since all satisfiable formulas have itself as MSS, but in general can not be solved by BCE. For example, consider the CNF formula $F = (a \vee \bar{b}) \wedge (b \vee \bar{c}) \wedge (c \vee \bar{a})$. F is satisfiable, but cannot be solved by BCE. We define MaxBS of a given CNF formula F to be the problem of finding an MBS of F with the largest cardinality.

4 Decompositions

One key observation in this paper is that every CNF formula can be decomposed into two subsets that both can be solved by BCE. Throughout the paper we will present procedures how to compute such decompositions. We will use the symbols L and R to denote the two subsets. Set L refers to the *left* or *large* subset as some algorithms aim to make one subset as large as possible. Set R refers to the *right* or *remainder* subset.

4.1 Symmetric Decompositions

A blocked clause decomposition of a CNF formula F is called *symmetric* if both subsets can be solved by BCE. A decomposition is *asymmetric* if only one of the subsets can be solved by BCE. It is easy to compute a symmetric decomposition for a given formula.

Consider the *PureDecompose* algorithm shown in Fig. 1. When this algorithm terminates, L and $R := F \setminus L$ can be solved by pure literal elimination and hence both L and R are blocked subsets of F. Note, that BCE simulates pure literal elimination [3]. Following the construction method, $|L| \geq |R|$. The runtime of *PureDecompose* can be made linear in the size of F using a standard implementation of occurrence lists.

```
    PureDecompose (formula F)
PD1   let L := ∅
PD2   while F not empty do
PD3     select a variable x ∈ vars(F)
PD4     if |F_x| ≥ |F_x̄| then L := L ∪ F_x
PD5     else L := L ∪ F_x̄
PD6     F := F \ (F_x ∪ F_x̄)
PD7   return L
```

Fig. 1. Pseudo-code of *PureDecompose* algorithm, with F_l the set of clauses with l.

Lemma 1. *The PureDecompose algorithm will produce a symmetric blocked clause decomposition for any CNF formula.*

Proof. Follows from the observation that L and $F \setminus L$ are blocked sets of F. □

Theorem 1. *Any CNF formula F can be decomposed into two subsets $L, R \subseteq F$ such that $F = L \cup R$ and $L, R \in \mathcal{BS}$, in a time linear in the size of F.*

Proof. Follows from the observation that the PureDecompose algorithm produces a symmetric blocked clause decomposition in linear time. □

The PureDecompose algorithm can be made more unbalanced (i.e., produce a larger L) by applying BCE on F in between lines PD2 and PD3 and move eliminated clauses to L. We decided against this "optimization" in the remainder of this paper, after observing that it is too costly for some huge formulas. As *post-processing*, after PureDecompose terminates, one can increase unbalancedness by looping over the clauses $C \in F \setminus L$ and add C to L if C is blocked with respect to L. Notice that blockedness of C has to be checked with the latest L.

4.2 Maximal Blocked Sets

This subsection discusses two favorable properties of maximal blocked sets. First, given an MBS M of a formula F, both F and M contain the same set of variables. Second, given a formula F one can compute an MBS of F in polynomial time.

Lemma 2. *Given a CNF formula F and an* MBS *M of F, $\mathsf{vars}(F) = \mathsf{vars}(M)$.*

Proof. Assume that $\mathsf{vars}(F) \neq \mathsf{vars}(M)$. There must be a clause $C \in F \setminus M$ containing a literal l corresponding to a variable $x \in \mathsf{vars}(F) \setminus \mathsf{vars}(M)$. Because $\bar{l}$ does not occur in $\mathsf{lits}(M)$, C is blocked on l w.r.t. M. Hence, $M \cup C$ is a blocked subset of F. However this contradicts that M is a maximal blocked subset. □

Consider the ConstructiveDecompose algorithm shown in Fig. 2 which moves clauses from F to L using BCE. The number of BCE calls is at most $|F|$ and each of those calls has a polynomial runtime in the size of F.

```
    ConstructiveDecompose (formula F)
CD1   let L := ∅
CD2   forall C ∈ F do
CD3     if BCE(L ∪ {C}) = ∅ then L := L ∪ {C}
CD4   return L
```

Fig. 2. Pseudo-code of the ConstructiveDecompose algorithm.

Lemma 3. *ConstructiveDecompose returns an* MBS *for a CNF formula F.*

Proof. Given a CNF formula F and the blocked subset M returned by the algorithm *ConstructiveDecompose*. Assume that M is not an MBS of M. In other words, there exists a clause $C \in F \setminus M$ such that $\mathrm{BCE}(M \cup C) = \emptyset$. This is not possible because when C was evaluated in the algorithm, the current L of *ConstructiveDecompose* must have been a subset of M. If $\mathrm{BCE}(M \cup C) = \emptyset$, then due to monotonicity of BCE for all $L \subseteq M$ it holds that $\mathrm{BCE}(L \cup C) = \emptyset$. Hence, C should have been in M. □

Theorem 2. *Computing a maximal blocked subset of a given CNF formula F can be realized in a time polynomial in the size of F.*

Proof. Follows from the observations that *ConstructiveDecompose* produces an MBS for a given formula F and requires polynomial time in the size of F. □

Lemma 4. *There exists a CNF formula for which the* ConstructiveDecompose *algorithm produces an asymmetric decomposition.*

Proof. Consider the following formula:

$$A := (a \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (b \vee \bar{c}) \wedge (\bar{b} \vee c) \wedge (c \vee \bar{d}) \wedge (\bar{c} \vee d) \wedge (d \vee \bar{e}) \wedge (\bar{d} \vee e) \wedge (\bar{a} \vee c) \wedge (a \vee \bar{e}) \wedge (\bar{c} \vee e)$$

Assume that the *ConstructiveDecompose* algorithm adds the clauses to L in the order in which they occur in A. This means the first eight clauses, lets call them A', are added to L because A' is a blocked set. However, BCE cannot solve any $A' \cup C$ with $C \in A \setminus A'$. Additionally, $A \setminus A'$ is not a blocked set. Hence, *ConstructiveDecompose* produces an asymmetric decomposition of A. □

We can obtain an algorithm that produces a symmetric maximal blocked clause decomposition by combining the *PureDecompose* and *ConstructiveDecompose* algorithms. Instead of $L := \emptyset$ in *ConstructiveDecompose*, change the initialization to $L :=$ *PureDecompose* (F).

An alternative approach is a *destructive* algorithm. Initially all clauses are in the large set and one by one a clause is eliminated. Algorithm 3 shows this approach for BS extraction. On the notion of "destructive" and "constructive" minimization algorithms, particularly in the context of minimal unsatisfiable subset (MUS) extraction, see [11].

```
    DestructiveDecompose (formula F)
DD1   let L := F
DD2   while BCE(L) is not empty do
DD3     remove a clause C ∈ BCE(L) from L
DD4   return L
```

Fig. 3. Pseudo-code of the *DestructiveDecompose* algorithm.

In contrast to the *ConstructiveDecompose* algorithm, the *DestructiveDecompose* algorithm might not produce an MBS.

Lemma 5. *There is a CNF formula for which the* DestructiveDecompose *algorithm produces an asymmetric decomposition and a non-maximal blocked set.*

Proof. Consider the following formula:

$$D := (a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a \vee c) \wedge (\bar{a} \vee c) \wedge (b \vee \bar{c}) \wedge (\bar{b} \vee \bar{c})$$

Let's assume that the *DestructiveDecompose* algorithm removes clauses based on their order in D. This means that first the clauses $(a \vee b)$, $(a \vee \bar{b})$, $(\bar{a} \vee b)$, and $(\bar{a} \vee \bar{b})$ will be removed, because BCE cannot eliminate any clause from D before that point. Now $F \setminus L$ is unsatisfiable and hence cannot be solved by BCE. In contrast, *PureDecompose* will produce a symmetric decomposition of D resulting in $L = (a \vee b), (a \vee \bar{b}), (a \vee c), (b \vee c)$ and $R := F \setminus L = (\bar{a} \vee b), (\bar{a} \vee \bar{b}), (\bar{a} \vee c), (\bar{b} \vee \bar{c})$.

DestructiveDecompose produces $L = (\bar{a} \vee c), (b \vee \bar{c}), (\bar{b} \vee \bar{c})$ which is not an MBS of F, because $(\bar{a} \vee b), (\bar{a} \vee \bar{b}) \in D$ are blocked w.r.t. L. □

Theorem 3. *The* MaxBS *problem is NP-hard.*

Proof. We show that the theorem holds by converting the NP-complete problem of Maximum Independent Set into MaxBS. The conversion works as follows. Given a graph $G = (V, E)$, we construct a CNF formula that contains Boolean variables v for each vertex $v \in V$. For each vertex $v \in V$, the formula contains the unit clause (v), while for each edge $uv \in E$ the formula contains the binary clause $(\bar{u} \vee \bar{v})$.

$$F_{\text{MIS}} := \bigwedge_{v \in V} (v) \wedge \bigwedge_{uv \in E} (\bar{u} \vee \bar{v})$$

Now we will show that a graph $G = (V, E)$ has an independent set of size k if and only if the corresponding F_{MIS} contains a blocking set of size $k + |E|$.

($\Rightarrow$) Let $S \subseteq V$ be an independent set of size k of G. The formula F' containing all binary clauses of F_{MIS} and unit clauses (v) for $v \in S$ is a blocking set of size $k + |E|$. To see that BCE can solve F', notice that all binary clauses are blocked on the literals $\bar{u}$ for $u \in V \setminus S$. After eliminating all these binary clauses, the unit clauses (v) for $v \in S$ have become blocked (pure literals).

($\Leftarrow$) Given a blocked subset B of F_{MIS} of size $k + |E|$. If B contains all the binary clauses in F_{MIS}, then the independent set is represented by the unit clauses in B: since B contains all binary clauses, it cannot contain both vertices of an edge because the clauses $(u), (v), (\bar{u} \vee \bar{v})$ together are unsatisfiable and hence not solvable by BCE.

If B does not contain all binary clauses, we will make another blocked subset B' of F_{MIS} that contains all binary clauses of F_{MIS} by exchanging unit clauses in B with the missing binary clauses. Let $(\bar{u} \vee \bar{v})$ be a missing binary clause in $F_{\text{MIS}} \setminus B$. In case $(\bar{u} \vee \bar{v})$ is blocked w.r.t. B, simply add $(\bar{u} \vee \bar{v})$ to B and remove an arbitrary unit clause from B. In case $(\bar{u} \vee \bar{v})$ is *not* blocked w.r.t. B, then both $(u), (v) \in B$. Now, add $(\bar{u} \vee \bar{v})$ and remove either (u) or (v) from B. By removing (u) or (v) from B, $(\bar{u} \vee \bar{v})$ becomes blocked on $\bar{u}$ or $\bar{v}$, respectively. □

4.3 Computing solutions in polynomial time

Given a blocked set B, one can compute a solution for B in polynomial time [3]. A procedure to obtain a solution uses the *reconstruction stack*. This stack is a sorted list of the clauses in B based on the order in which BCE can eliminate them. Given a reconstruction stack S of B, one can compute a solution as follows. Generate a random truth assignment τ of the variables in B. Pop the clauses from S one by one. If τ falsifies a clause C with blocking literal l, flip the truth value of l in τ to 1. Fig. 4 shows how to compute a reconstruction stack and demonstrates how to use the stack to obtain satisfying assignments.

```
    ReconstructionStack (blocked set B)
RS1   let S be an empty stack
RS2   while B not empty do
RS3     let C ∈ B be a clause that is blocked w.r.t. B
RS4     B := B \ C
RS5     S.push(C)
RS6   return S

    GetSolution (blocked set B)
GS1   let τ be a random truth assignment of the variables in B
GS2   S := ReconstructionStack (B)
GS3   while S not empty do
GS4     C := S.pop() and let l ∈ C be the blocking literal
GS5     if τ falsifies C then τ(l) = 1
GS6   return τ

  GetMultipleSolutions (blocked set B, bit-width w)
GMS1  let T be a set of assignments of random bit-vectors with width w for x ∈ vars(B)
GMS2  S := ReconstructionStack (B)
GMS3  while S not empty do
GMS4    C := S.pop()
GMS5    let b be an all zero bit-vector of width w
GMS6    forall l ∈ C do b := b OR T(l)
GMS7    let l' ∈ C be the blocking literal w.r.t. B
GMS8    T(l') := T(l') XOR NOT(b)
GMS9  return T                                  // set of w satisfying assignments
```

Fig. 4. Pseudo-code *ReconstructionStack*, *GetSolution*, *GetMultipleSolutions* algorithms.

One can use the reconstruction set to compute multiple solutions in linear time of the size of the blocked set using bit-vectors. The bottom part of Fig. 4 shows the algorithm. Each variable is assigned a random bit-vector of width w. Positive literals have the bit-vector assignment of the corresponding variable, while negative literals have a bit-vector assignment which complements the one of the corresponding variable. For each clause C that is popped from the stack,

a bit-vector b is obtained by computing the logical OR of all the bit-vectors of the literals $l \in C$. If b contains zeros, then those bits are flipped in the bit-vector assignment of the literal that blocks C. The result of the algorithm is a set of w satisfying assignments — some of them might be equivalent.

The complexity of computing *all* solutions of a blocked set is unknown, but we conjecture below that they can be computed in polynomial time in the number of solutions. It is not clear whether one can use the reconstruction stack to enumerate the solutions of blocked sets.

Conjecture 1. Given a blocked set B with k satisfying assignments. Computing all satisfying assignments of B requires at most k polynomial-time computations.

Below some intuition why we believe that the conjecture might hold. Consider a Boolean circuit BC with unrestricted output gates and a CNF formula F_{BC} being the Tseitin translation of BC. Let n be the number of input gates of BC. The number of solutions of F_{BC} is 2^n – exactly one solution for each assignment to the input gates. We showed that BCE can eliminate all clauses from a Boolean circuit for which the output gates are not restricted [3]. Hence F_{BC} is a blocked set. The variables in F_{BC} corresponding to the input gates occur in the last clauses that BCE will eliminate. Assigning variables in the reverse order that BCE eliminates them, will enumerate the solutions of F_{BC}. We observed this for other blocked sets as well, although we also found some counter-examples. We expect that a more sophisticated procedure could work for any blocked set.

In case the conjecture holds, blocked sets are useful when they have few and many solutions. Given a maximal blocked set M of a CNF formula F, F is satisfiable if and only if a solution of M exists which satisfies F — because M is a subset of F and $\mathsf{vars}(F) = \mathsf{vars}(M)$. So in case M has few solutions we can compute them all in polynomial time to solve F. If M has many solutions, then we can generate a lot of them in linear time to search for patterns.

5 Heuristics and Efficiency

For the applications of blocked clause decomposition that we have in mind, one wants to have the decomposition as unbalanced as possible. Ideally, one subset contains only one clause while the large subset contains all the other clauses. In this section we discuss heuristics to obtain unbalanced decompositions.

In order to make a decomposition useful, one must be able to compute it efficiently. This section offers several ideas we came up with to improve the performance. A fast implementation of BCE is crucial for all the algorithms. An important optimization is a literal-based priority queue. Details about this and other BCE optimizations are presented in Section 10 of [3].

The QuickDecompose Algorithm If a formula is partitioned arbitrarily it is not unlikely that one of its part can be solved by BCE. In this case we can add all its clauses to the MBS, which we want to construct. Otherwise, the partition

should be refined. This idea leads to the *QuickDecompose* algorithm shown in Fig. 5, which is similar in spirit to the *QuickXplain* algorithm [12].

QuickDecompose is a more efficient variant of *ConstructiveDecompose*. Hence, it will always produce a maximal blocked set (Lemma 3), but decompositions can be asymmetric (for example on the CNF formula A in Lemma 4). In order to make all decompositions symmetric, the initialization at line `QD1` should be changed to $L :=$ *PureDecompose* (F).

The advantage of this algorithm is that it only needs $\mathcal{O}(\log|F|)$ calls to BCE to zero in on an MBS, if the formula F has exactly one MSB, which in addition also is assumed to contain a single clause. We conjecture that $\mathcal{O}(m + \log|F|)$ calls are needed in general, where m is the maximum size of an MBS of F. Thus this algorithm is particularly useful if m is small. However, for practically all benchmarks from the SAT competitions, we observed that m is close to $|F|$.

```
    QuickDecomposeRecursive (formula F)
QDR1  if BCE(L ∪ F) = ∅ then L := L ∪ F
QDR2  else if |F| ≠ 1 then        // partition F in non-empty sets G and H
QDR3    let F = G ∪ H with G, H ≠ ∅ and G ∩ H = ∅
QDR4    QuickDecomposeRecursive (G)
QDR5    QuickDecomposeRecursive (H)

    QuickDecompose (formula F)
QD1   let L := ∅                  // visible in QuickDecomposeRecursive
QD2   QuickDecomposeRecursive (F)
QD3   return L
```

Fig. 5. Pseudo-code of the *QuickDecompose* algorithm.

Optimizations The most important optimization is to replace the recursive simple depth-first search by a prioritized search, where larger subsets are tried first. Further, many instances are encodings from circuit SAT problems [3], where the circuit is encoded via Tseitin encoding and zero/one constraints on circuit nodes and outputs are added as additional unit clauses. In this situation, removing the units from the CNF results in a blocked set. Thus we added a pre-processing algorithm, which removes N from F, where $N \subseteq F$ is the set of non-unit clauses of F, and then initializes L to N, if $\mathsf{BCE}(N) = \emptyset$.

We also observed that it can be useful to check whether removing at most 50% of the longest clauses would result in a blocked set. If this is the case we proceed with the shorter clauses and initialize L accordingly.

Finally, redundant BCE calls might occur, for which it has already determined previously that the formula is not solvable through BCE. Thus we maintain a cache of formulas F for which $\mathsf{BCE}(F)$ was not successful and produced a non-empty set as result. Thus the call to the BCE procedure at line `QDR1` would first check whether its argument is not already in the cache.

Results We developed two decomposition tools. The first tool, called BCDD, implements *PureDecompose*. BCDD comes in two variants: the one shown in Fig. 1 (default) and one with the optimization discussed in the last sentences of Section 4.1 (post-processing). The second tool, called SBLITTER, implements *QuickDecompose* and includes the optimizations described above. The results are shown in Table 1. Observe that the tools can help each other by providing the symmetric decomposition of BCDD to SBLITTER. Although the runtime of SBLITTER is polynomial in the size of its input, it was not able to finish (obtain a maximal blocked set) on most benchmarks within 100 seconds.

tool	mode	A	B	R	O	T	TO	M
BCDD|SBLITTER	post-processing	85%	371	69	55	25	218	71
BCDD|SBLITTER	default	84%	367	73	55	25	218	71
BCDD	post-processing	82%	358	82	0	2	44	41
BCDD	default	80%	349	91	0	2	21	39
SBLITTER	default	33%	143	298	55	24	218	72

Table 1. Comparing the decomposition tools on 299 benchmarks from the SAT Competition 2013 application track. We removed a huge instance (esawn_uw3.debugged) with 54 million clauses which caused a memory out. Column 'A' shows the *average* fraction of the large subset. The sum of the sizes of the *blocked* subset L is shown in column 'B', the sum of the *remaining* clauses in 'R', both in millions of clauses. The number of benchmarks with exactly *one* remaining clause is listed in column 'O'. Then the sum of the time taken follows in column 'T'. The number of times the time-out of 100 seconds was hit for SBLITTER and 10 seconds for BCDD is shown in column 'TO'. In those $81 = 299 - 218$ cases where SBLITTER (actually all versions) finished before the time-out an MBS was found. The last column 'M' lists the sum of the maximum memory used in all the runs in GB.

6 SAT Sweeping, Equivalence Checking and Extraction

SAT sweeping is a well-known and very effective preprocessing technique for satisfiability problems represented as circuits. It is based on techniques used in formal equivalence checking of circuits. See [2,13] for a complete list of references, and further the independently derived results in [14,15]. Similar techniques have been used in the context of computing backbones, see for instance [16]. Related approaches for sequential equivalence checking [17] are used for preprocessing model checking problems and resemble refinement techniques in fast algorithms for minimizing automata [18].

One variant of SAT sweeping starts by assigning random bit-vectors to the input gates of a given circuit. Afterwards, these values are propagated. Gates with the same bit-vector value are potentially equivalent. Next, a pair of potentially equivalent gates is selected and a SAT formula is generated stating that

these gates are not equivalent. In case the formula is satisfiable, the set of potentially equivalent gates is refined. Otherwise, the two gates are merged. This process continues until there are no potentially equivalent gates left.

SAT sweeping might also be a useful preprocessing technique for SAT solving. However, porting this technique to SAT solving is not trivial. Unlike a circuit, a SAT formula has no input gates. Consequently, assigning variables to random values followed by propagation will typically result in a conflict. Hence, it is much harder to obtain a list of potentially backbone or equivalent variables.

In order to use SAT sweeping as preprocessing technique for a CNF formula F, we need to compute a large satisfiable subset L of F, which is easy to satisfy. This is exactly what blocked clause decomposition gives us. Further, it is easy to find a solution for a blocked set, i.e., linear in the size of the subset. This gives a fast way to initialize the partition of potentially equivalent literals. We used the *GetMultipleSolutions* algorithm in Fig. 4 to efficiently generate many random solutions in linear time. Variables with the same bit-vector value are potentially equivalent, while potential backbone variables have either all true or all false bit-vectors.

EquivalenceExtraction (satisfiable set L)
EE1 let τ be a solution for L, hence $\tau(L) = 1$
EE2 let $P = \{\{l \in \mathsf{lits}(L) \mid \tau(l) = 1\}\}$ // partition of potentially equivalent literals
EE3 let $E = \emptyset$ // set of determined equivalences
EE4 **while** exists a class $\mathcal{C} \in P$ with $l, k \in \mathcal{C}$ and $l \neq k$ **do**
EE5 **if** $\mathrm{SAT}(L \cup \{(l)\} \cup \{(\bar{k})\})$ **then** refine P by returned solution τ
EE6 **else if** $\mathrm{SAT}(L \cup \{(\bar{l})\} \cup \{(k)\})$ **then** refine P by returned solution τ
EE7 **else** add equivalence $l = k$ to E and remove k from $\mathcal{C}$
EE8 **return** E

Fig. 6. Pseudo-code of the *EquivalenceExtraction* algorithm.

Given a *satisfiable* formula L, SAT sweeping can be implemented as shown in the algorithm in Fig. 6. As result it produces the strongest set of equivalences E, modulo transitivity and equivalent literal substitution, with $L \models E$. If one of the SAT calls in line `EE5` or `EE6` returns a solution τ then $\tau(l) \neq \tau(k)$ and thus the partition P is refined by splitting class $\mathcal{C}$ into $\mathcal{C}_0 = \{l \in \mathcal{C} \mid \tau(l) = 0\}$ and $\mathcal{C}_1 = \{l \in \mathcal{C} \mid \tau(l) = 1\}$. The result of the refinement is $(P \backslash \{\mathcal{C}\}) \cup \{\mathcal{C}_0, \mathcal{C}_1\}$.

In practice, several important optimizations are required (see also [13]). First, incremental SAT solving [19] should be used, adding L as fixed formula permanently, but treating the two unit clauses added in line `EE5` and `EE6` as assumptions [19]. This allows to reuse learned facts from one SAT call to the next, which is particularly important for learned equivalences: If both queries to the SAT solver in line `EE5` and `EE6` are unsatisfiable, the SAT solver will in essence learn the two clauses $(\bar{l} \vee k)$ and $(l \vee \bar{k})$, which implicitly record equivalence of l and k in the SAT solver as well.

The second most important optimization is to bound the time spent in each SAT solver call, by for instance posing a limit on the number of conflicts. Third, it is useful to simplify L by SAT based preprocessing. Note, however, that unrestricted satisfiability preserving preprocessing, such as unrestricted BCE, will just turn L into an empty CNF, which then will not have any equivalences. We propose to restrict preprocessing to those cases, where solutions to L projected on the common variables between L and $R = F \setminus L$ do not change. More concretely blocked clause addition is disabled, and common variables are "frozen", which means they can not be eliminated nor used as blocking literal etc. This technique will of course not preserve internal equivalences within L, but still proved to be useful in practice.

Regarding heuristics for choosing the pair of literals l and k in line `EE4`, which are tried to be merged next, we suggest to alternate between randomly picking literals, favoring large equivalence classes, and then for every second candidate pair pick two random literals from the next equivalence class in a round-robin fashion, smallest classes first.

The extraction algorithm will implicitly also produce many learned unit clauses of the backbone of the blocked set. In our current implementation we remove them immediately from the partition P. If at least one unit is kept in P our algorithm will actually produce the backbone of the blocked set. It can then be seen as an extension of the iterative backbone extraction algorithm in [16].

7 Results

The algorithms presented above have been implemented. Source code and the log files of the experiments are available at `http://fmv.jku.at/bcd`.

We evaluated the effectiveness of SAT sweeping on CNF formulas on some instances from the application track of the SAT Competition 2013 using an improved version of the SAT solver Lingeling [20], the winner of this track.

We observed that our equivalence extraction tool was only useful for those benchmarks for which our decompose tools were able to compute a maximal blocked subset. Therefore, we selected all 81 instances of the application track for which Bcdd | Sblitter (with post-processing) was able to compute an MBS in 100 seconds (see Table 1 for details). Our equivalence extraction tool outputs the backbone variables and equivalences or a subset in case the time limit of 2000 seconds was hit. The simplified CNF is finally given to Lingeling. The total running time is limited to 5000 seconds, both for the sequence of blocked clause decomposition, equivalence extraction and then Lingeling, as well as for plain SAT solving by Lingeling. This is the same time limit as used in the competition but on Intel Q9550 2.83GHz instead of Intel E5440 2.83 GHz processors.

For 16 out of 81 instances with an MBS, the extract part runs into the time limit of 2000 seconds. For the other 65 instances, extract was able to reduce all equivalence classes of the partition P to singletons. Altogether, the MBSs of all 81 instances consist of 10 355 344 clauses, from which 3 313 948 (32%) were still active (non-singletons) after SAT based preprocessing. From those active

variables the tool removed 66 267 backbone variables (2%) and found 343 716 equivalences (10%), due to succeeding implication checks at lines `EE5` and `EE6` in *EquivalenceExtraction*. Out of 397 228 SAT solver calls, 48 912 produced a solution (12%), while 241 790 were unsatisfiable (61%), and 106 526 calls (27%) used up the budget of 100 conflicts. The *GetMultipleSolutions* algorithm was called 52 834 times in an interleaved fashion with the main equivalence extraction loop, right after the SAT calls in line `EE5` and `EE6`, scheduled with a frequency of approximately every 4th SAT solver call. Each time it used bit-vectors of width 512 and produced altogether 27 051 008 solutions. These solutions were used to split 2 164 294 classes (90%), while the single solutions from the SAT solver calls in lines `EE5` or `EE6` returning a solution only split 237 943 classes (10%).

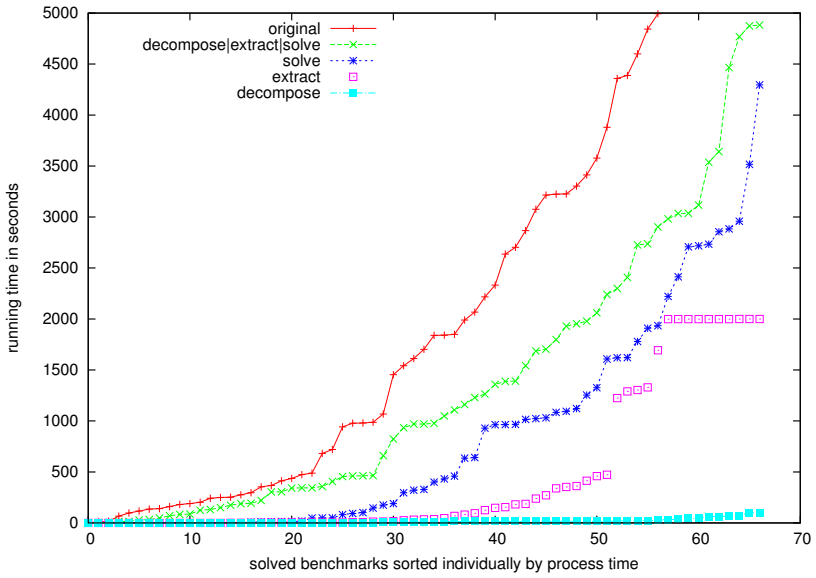

Fig. 7. Running plain SAT solving (original) and our new approach (decompose | extract | solve) on the 81 application track instances from the SAT Competition 2013 for which our decompose tools can compute a maximal blocked subset within 100 seconds. The total time limit is 5000 seconds (also used during the competition). For the anytime algorithms decompose and extract a fixed time budget was allowed of at most 100 and 2000 seconds, respectively. The rest of the time is used for solving (solve).

Fig. 7 shows the results of our experiments. Notice that Lingeling contains many advanced equivalence reasoning engines which were enabled during all runs. Our new approach is able to solve ten instances more than plain SAT solving. It requires at most 100 seconds to determine whether a formula would benefit from our approach, i.e., whether decompose can compute an MBS. If we take this time into account on the other 218 instances, Lingeling solves one in-

stance less. So the total gain on the whole suite is nine benchmarks. Although the new approach is faster, quite some time is spent on equivalence extraction. We expect that a faster implementation of extract can further improve the results by bringing the decompose | extract | solve line closer to the solve line. Furthermore, by speeding up decompose, we can compute MBSs for more formulas thereby enlarging the number of benchmarks for which our approach is expected to be useful.

8 Conclusions

We introduced the concept of *blocked clause decompositions* and showed that any CNF formula can be decomposed into two blocked sets in polynomial time. Additionally, we showed how to obtain a maximal blocked set in polynomial time. The problem of finding a maximal blocked set with the largest cardinality is NP-hard. We presented several algorithms to obtain decompositions as well as heuristics and optimizations to make the procedures effective and efficient.

We implemented blocked clause decomposition and SAT sweeping for CNF formulas in Lingeling, winner of the application track of the SAT Competition 2013. We evaluated the proposed techniques on the benchmarks of this track. Lingeling with the new techniques was able to solve ten more instances for which our tools were able to compute a maximal blocked set within 100 seconds.

Future work will focus on improving the efficiency of our tools. In case we can obtain maximal blocked sets faster, SAT sweeping is expected to be useful for more benchmarks. Additionally, by reducing the costs of SAT sweeping, we can increase the benefit of this preprocessing technique. Finally, SAT sweeping can be implemented more effectively via inprocessing [21] — by interleaving detection of backbone variables and binary equivalences with conflict-driven search.

References

1. Krohm, F., Kuchlmann, A., Mets, A.: The use of random simulation in formal verification. In: Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on. (1996) 371–376
2. Kuehlmann, A.: Dynamic transition relation simplification for bounded property checking. In: ICCAD. (2004) 50–57
3. Järvisalo, M., Biere, A., Heule, M.J.H.: Simulating circuit-level simplifications on cnf. Journal of Automated Reasoning **49**(4) (2012) 583–619
4. Parkes, A.J.: Clustering at the phase transition. In: Proceedings of AAAI'97, AAAI Press (1997) 340–345
5. Zhang, W., Rangan, A., Looks, M.: Backbone guided local search for maximum satisfiability. In: Proceedings of IJCAI'03, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2003) 1179–1184
6. Singer, J., Gent, I.P., Smaill, A.: Backbone fragility and the local search cost peak. Journal of Artificial Intelligence Research **12** (2000) 235–270
7. Heule, M.J.H., Järvisalo, M., Biere, A.: Revisiting hyper binary resolution. In Gomes, C., Sellmann, M., eds.: Proceedings of CPAIOR 2013. Volume 7874 of LNCS. Springer Berlin Heidelberg (2013) 77–93

8. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics **96–97** (1999) 149–176
9. Järvisalo, M., Biere, A., Heule, M.J.H.: Blocked clause elimination. In Esparza, J., Majumdar, R., eds.: Proceedings TACAS 2010. Volume 6015 of LNCS., Springer (2010) 129–144
10. Liffiton, M., Sakallah, K.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning **40**(1) (2008) 1–33
11. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2) (2012) 97–116
12. Junker, U.: Quickxplain: Preferred explanations and relaxations for over-constrained problems. In: AAAI. (2004) 167–172
13. Khasidashvili, Z., Nadel, A.: Implicative simultaneous satisfiability and applications. In: HVC. Volume 7261 of LNCS. (2011) 66–79
14. Codish, M., Fekete, Y., Metodi, A.: Compiling finite domain constraints to SAT with BEE. In: POS. (2013)
15. Metodi, A., Codish, M., Stuckey, P.J.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. JAIR **46** (2013) 303–341
16. Janota, M., Lynce, I., Marques-Silva, J.: Experimental analysis of backbone computation algorithms. In: International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion (RCRA). (2012)
17. van Eijk, C.A.J.: Sequential equivalence checking based on structural similarities. IEEE Trans. on CAD of Integrated Circuits and Systems **19**(7) (2000) 814–819
18. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA (1971)
19. Eén, N., Sörensson, N.: An extensible sat-solver. In Giunchiglia, E., Tacchella, A., eds.: Theory and Applications of Satisfiability Testing. Volume 2919 of LNCS. Springer Berlin Heidelberg (2004) 502–518
20. Biere, A.: Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In: Proceedings of SAT Competition 2013. (2013)
21. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Proceedings of IJCAR 2012. Volume 7364 of LNCS., Springer (2012) 355–370